UNIVERSITÀ DEGLI STUDI DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

SCUOLA DI DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

INDIRIZZO IN SCIENZA E TECNOLOGIA DELL'INFORMAZIONE

CICLO XXVIII

ON SPACE CONSTRAINED COMPUTATIONS

Direttore della Scuola

Ch.mo Prof. Matteo Bertocco

Coordinatore d'indirizzo

Ch.mo Prof. Carlo Ferrari

Supervisore

Ch.mo Prof. Gianfranco Bilardi

Dottorando : Lorenzo De Stefani

On space constrained computations

Lorenzo De Stefani

Advisor: Ch.mo Prof. Gianfranco Bilardi

Department of Information Engineering, University of Padova

January, 2016

Sommario

A partire dall'avvento del calcolatore digitale (computer), la sua tecnologia costitutiva è stata caratterizzata da un ritmo di sviluppo costante e impressionante. Sebbene la maggior parte dei parametri siano ancora in via di miglioramento, vi è un crescente consenso che i limiti fisici alla velocità di propagazione dei segnali ed alla dimensione dei dispositivi integrati stiano diventando sempre più significativi. In definitiva, il tempo di accesso alla memoria associata ad un calcolatore digitale è destinato ad aumentare con la dimensione della memoria.

Pertanto, quando è richiesta una memoria di grandi dimensioni, risulta conveniente organizzarla gerarchicamente in più livelli, caratterizzati da dimensione e tempo di accesso progressivamente crescenti. Tipicamente, i livelli di gerarchia di memoria comprendono i registri della CPU, due o tre livelli di cache, la memoria principale (RAM) e i dischi. Rispetto ai registri della CPU, le memorie cache sono qualche centinaia di volte più lente, la memoria RAM è qualche migliaio di volte più lenta, mentre i dischi sono qualche milione di volte più lenti. L'uso efficace dei livelli più veloci della gerarchia di memoria è pertanto un punto chiave nella progettazione ed implementazione di algoritmi. I limiti fisici sono un problema anche nel contesto delle architetture multiprocessore e del calcolo parallelo, a causa del ritardo introdotto dalla velocità dei segnali usati per la comunicazione tra le varie unità di elaborazione. Inoltre, nonostante la Legge di Moore predica un aumento di velocità esponenziale per l'hardware in generale, il tasso di miglioramento annuale del tempo-per-operazione-aritmetica (miglioramento CPU), nel corso degli anni, ha costantemente superato quella del tempo-per-lettura/scrittura-dato. Appare legittimo aspettarsi che la percentuale di tempo consumata per la comunicazione diventi sempre più rilevante, costituendo sempre più un collo di bottiglia per le prestazioni sia delle memorie multilivello che delle architetture di calcolo parallelo.

Nel valutare la complessità degli algoritmi, si devono pertanto considerare due tipi di costo: il costo aritmetico, che dipende dal numero di passi computazionali richiesti, ed il costo di comunicazione (I/O), che dipende dal movimento dei dati richiesto nell'ambito dell'esecuzione di un algoritmo, tra livelli diversi della gerarchia di memoria (nel caso sequenziale), o nella rete di collegamento tra diversi processori (nel caso parallelo). In entrambi questi scenari applicativi, la componente di I/O ha spesso un impatto sulle prestazioni dell'algoritmo molto più significativo del tempo della componente aritmetica.

È quindi di grande interesse indagare da un lato lo spazio di memoria minimo richiesto per il calcolo di un algoritmo (space complexity), e dall'altro il tradeoff tra lo spazio di memoria effettivamente utilizzato e il volume di comunicazione dei

dati necessari per l'esecuzione dell'algoritmo (I/O complexity). Oltre all'interesse puramente teorico di tale analisi, il perseguimento di buone tecniche per individuare limiti inferiori (lower bounds techniques) è anche fondamentale per il perseguimento di algoritmi ad alte prestazioni, in quanto consentono di valutare la distanza di una soluzione proposta dal livello ottimale.

Nel nostro studio ci focalizziamo sui calcoli eseguiti con programmi straight-line (in contrapposizione ai programmi branching) in modalità indipendente dai dati, dove quindi la successione delle operazioni da eseguire non è influenzata dal valore specifico di valori di ingresso (in contrapposizione alle computazioni data-dependent), che possono essere rappresentati grafi diretti aciclici computazionali (CDAG) $G(I \cup V, E)$, i cui vertici rappresentano operazioni (sia di input/output che di elaborazione) e i cui archi rappresentano dipendenze tra i dati (data dependencies) [58]. In questa tesi analizziamo vari aspetti dei calcoli di CDAG, tra cui i loro requisiti di memoria e la quantità di movimento di dati (tra diversi livelli di una gerarchia di memoria o tra diversi processori che eseguono un programma in parallelo) richiesti in situazioni in cui è disponibile solo una quantità limitata di spazio di memoria.

La tesi è organizzata come segue. Nel Capitolo 1, si introduce la notazione e le principali definizioni che verranno utilizzati nella presentazione.

Nel Capitolo 2, si studiano limiti inferiori (lower bounds) per le dimensioni dello spazio di memoria che è necessario per calcolare vari CDAGs. Introduciamo il Pebble Game, uno strumento concettuale utilizzato in letteratura per studiare i requisiti di spazio di memoria dei calcoli su CDAG. Successivamente si descrive l'approccio *Marking Rule* (Regola di Marcatura) originariamente introdotta da Bilardi et. al. [10], e lo si applica per ottenere limiti inferiori (lower bounds) per la lo spazio di memoria necessario per la valutazione dei CDAG Superconcentrators-Stack [68, 40] . Al fine di studiare i limiti dell'approccio con Marking Rule, introduciamo il concetto di *visita* di un CDAG, e dimostriamo vari limiti superiori (upper bounds) per lo spazio di memoria necessario per visitare un CDAG in condizioni appropriate.

Nel Capitolo 3, si studiano limiti superiori (upper bounds) per lo spazio di memoria minimo necessario per calcolare qualsiasi CDAG. Dopo aver esaminato i principali contributi della letteratura [35, 40, 43], si presenta un nuovo algoritmo che permette di valutare (*pebble*) qualsiasi CDAG con $|E| = m$ archi usando al massimo uno spazio di memoria $\mathcal{O}(m \log m)$.

Nel Capitolo 4, si rivolge l'attenzione al costo legato alla comunicazione *I/O cost* per le computazioni di CDAG, inteso come scambio di dati tra i diversi livelli di una gerarchia di memoria, o come la comunicazione di dati tra diversi processori che eseguono un programma in parallelo. In particolare, otteniamo limiti inferiori (lower bounds) per la complessità di ingresso-uscita (I/O) dei calcoli di CDAG, in relazione al modello classico di Hong e Kung [37]. Si studiano quindi le esecuzioni sequanziali dell' algoritmo di Strassen per la moltiplicazione di matrici quadrate su una piattaforma equipaggiata con una memoria gerarchica a due livelli. In tale modello, si ottiene quindi un limite inferiore (lower bound) per la complessità di I/O dell'algoritmo di Strassen, sotto il vincolo che nessun risultato intermedio venga calcolato più di una volta durante l'esecuzione dell'algoritmo. Sebbene il limite inferiore ottenuto sia già stati presentato in letteratura [7, 62], la nostra tecnica permette di ottenere dimostrazioni più pulite, basate sulla struttura ricorsiva degli algoritmi anziché sulle proprietà combinatorie dei CDAG che li rappresentano.

Sempre per il modello sequenziale, nel contributo principale del Capitolo 4, si fornisce un nuovo limite inferiore (lower bound) per la complessità di I/O dell'algoritmo di moltiplicazione matriciale di Strassen, che vale per tutte le possibili computazioni, senza vincoli sul numero di volte in cui un risultato immediato può essere calcolato.

Sfruttando la stessa tecnica usata per il risultato appena menzionato, si ottiene un limite inferiore per la complessità di I/O per computazioni dell'algoritmo di Strassen eseguite in parallelo da P, ciascuno equipaggiato con una memoria finita, processori connessi tra loro. Nessuna assunzione è necessaria riguardo l'iniziale distribuzione dei dati di input fra i P processori.

Nel Capitolo 5, si considera l'effetto di un utilizzo opportuno della memoria nel contesto di algoritmi resilienti agli errori (di memoria), che forniscono soluzioni (quasi) corrette anche quando si verificano errori di memoria "*silenziosi* (corruzioni di dati memorizzati), ovvero che non causano il blocco dell'esecuzione del programma. In particolare, si va a fornire una panoramica dei risultati ottenuti nell'articolo [22].

Abstract

Since the advent of the digital computer, its supporting technology has been characterized by steady and impressive growth. Although most parameters are still being improved, there is an emerging consensus that physical limitations to signal propagation speed and device size are becoming increasingly significant. Ultimately, the access time to the memory associated to a digital computer is bound to increase with the size of the memory. Therefore, when a large overall memory is required, it becomes convenient to organize it hierarchically into a sequence of levels whose size and access time increase progressively. Typically, the levels of memory hierarchy include the CPU registers, two or three cache levels, main memory and disks. Compared to the CPU registers, main memory is a few hundred times slower and disks are a few million times slower, hence, effective use of the fastest levels of the memory hierarchy is becoming a key concern in the design and implementation of algorithms.

Physical limitations are a concern also in the context of multiprocessor architectures and parallel computing, due to the delay introduced by the speed of the signals used for the communication between the various processing units. Furthermore, while Moore's Law predicts an exponential speedup of hardware in general, the annual improvement rate of time per-arithmetic-operation has, over the years, consistently exceeded that of time-per-word read/write. The fraction of running time spent on communication is thus expected to increase further, becoming more and more of a bottleneck for the performance of both multi-level memory and parallel computing architectures.

When considering the complexity of algorithms, two kinds of costs are therefore to be considered: the *arithmetic* cost which depends on the number of required computational steps, and the *communication* cost which depends on the required movement of data within the execution of an algorithm, either between levels of a memory hierarchy (in the sequential case), or over a network connecting processors (in the parallel case). In both of these applicative scenarios, the communication component of an algorithm often costs significantly more time than its arithmetic component.

It is therefore of interest to investigate the minimum memory space required for computation of algorithms on the one hand (the *space complexity*), and then the tradeoff between the memory space actually being used and the data communication needed for the algorithm execution (the *I/O complexity*).

In addition to a purely theoretic interest of such an analysis, the pursuit of good lower bounds techniques is also crucial for the pursuit of high performances

algorithms, since they enable to evaluate the distance from optimality of a proposed solution.

In our study we focus on computations done with straight-line programs (opposed to branching programs) in a data-independent fashion, where the succession of the operations to be executed is thus not influenced by the specific value of input values (opposed to data-dependent computations), which can be modeled as Computational Directed Acyclic Graph (CDAG) $G(I \cup V, E)$, whose set of vertices represents operations (of both input/output and processing type) and whose set of edges represents data dependencies [58]. In this thesis we investigate various aspects of CDAG computations, among which their memory requirements and the amount of data movement (either between levels of a memory hierarchy or between various processors executing a program in parallel) required in situations in which only a limited amount of memory space is available. This thesis is organized as follows. In Chapter 1, we introduce the notation and the main definitions which will be used through the presentation.

In Chapter 2, we study lower bounds on the size of the memory space which is necessary to compute various CDAGs. We introduce the *Pebble Game*, a theoretical device used in literature to study the space requirements of CDAGs computations. We then describe the *Marking Rule* approach originally introduced by Bilardi et. al. in [10] and we apply it in order to obtain lower bound on the space complexity of Superconcentrators-Stack CDAGs[68, 40]. In order to study the limits of the Marking Rule approach, we introduce the concept of *visit* of a CDAG, and we prove various upper bounds on the memory space required for visiting a CDAG under appropriate conditions.

In Chapter 3, we study upper bounds on the minimum memory space necessary to compute any CDAG. After reviewing the main contributions in literature [35, 43, 40], we present a novel algorithm which allows to pebble any CDAG with $|E| = m$ edges using at most $O(m \log m)$ memory space.

In Chapter 4, we direct our attention towards the "*inpu-output cost*" (I/O cost) of CDAGs computations, intended either as the data exchange between different levels of a memory hierarchy, or as the communication of data between various processors executing a program in parallel. In particular, we obtain lower bounds for the input-output (I/O) complexity of CDAGs computations with respect to the classical model by Hong and Kung [37]. We begin by studying the I/O complexity of Strassen's algorithm when executed sequentially on a machine equipped with a two level memory hierarchy. We provide an alternative technique to those in [4] and [62] to obtain a tight lower bound to the I/O complexity of Strassen's matrix multiplication algorithm for computations in which no intermediate result is ever recomputed. We then obtain the first asymptotically tight lower bound to the I/O complexity of Strassen's algorithm for general computations, that is, computations without any restriction on the recomputation of intermediate values. Our technique is based on a novel application of Gigoriev's *information flow* concept [33]. We also study the I/O complexity of Strassen's algorithm when executed in parallel by P processors each equipped with a finite memory. We obtain an lower bound which holds for any computation (no restriction on recomputation), without any assumption regarding the distribution of the input data among the P processors at the beginning of the computation.

Furthermore, in the main contribution of Chapter 4, we provide a novel lower bound for the I/O complexity of Strassen's matrix multiplication algorithm, which holds for all possible computations, without constraints on the number of times an immediate result can be computed.

In Chapter 5, we consider the effect of opportune memory utilization in the context of *error resilient algorithms*, which provide (almost) correct solutions even when silent memory errors occur. In particular, we provide a brief overview of the results published by the author in [22].

Acknowledgments

I would like to express my special appreciation and thanks to my advisor Professor Gianfranco Bilardi, which has been a tremendous mentor for me. I would like to thank him for encouraging my research and for allowing me to grow as a research scientist. His advice on both research as well as on my career have been priceless. I want to thank my academic siblings and collaborators at the University of Padova, Francesco Silvestri, Michele Shimd, Nicola Zago and Emanuele Milani for the many interesting conversations and exchanges regarding research. I would also like to thank Professor Eli Upfal form Brown University whit whom I had the chance to work during my period abroad doing stimulating and interesting research.

A special thanks to my family. Words cannot express how grateful I am to my mother and father for all the sacrifices that they have made on my behalf and for all their understanding and support during these years. I would also like to thank my friends, especially Dario and Andrea, who supported me during these years, reminding me to have fun and enjoy this period of my life.

Last, but definitely not least, I would like to express my most sincere thanks to my beloved soon to be wife Megumi whose support and encouragement was crucial especially during the conclusion of my studies.

Contents

List of Figures

Chapter 1

Preliminaries

In our work we focus on computations executing *straight-line programs* (opposed to branching programs) in a data-independent fashion, where the succession of the operations to be executed is thus not influenced by the specific value of input values (opposed to data-dependent computations).

1.1 Straight line programs and Computational Directed Acyclic Graphs

Definition 1.1 (Straight-line program). *A straight-line program is a set of steps, each associated with a distinct integer number i from 1 to n:*

- input step*: denoted as (s:* READ *x), where x denotes an input variable;*
- computation step*: denoted as (s:* OP *$o_1, \dots, o_k$), where* OP *identifies the operation executed and $o_1, \dots, o_k$ denote its operand;*
- output step*: denoted (s* OUTPUT *y), where y denotes an output variable;*

The operation executed at the i-th step can have as input operator only values which have been either computed during previous computational steps or acquired trough a previous input step.

Algorithms for many important problems such as Fast Fourier Transform (FFT) and matrix multiplication are naturally computed by straight-line programs.

The requirement that each computation step operates on results produced in preceding steps ensures that each such program can be modeled as a *Directed Acyclic Graph* (*DAG*), also called *Computational Directed Acyclic Graph* (*CDAG*) or *circuit*, whose vertices (also called *gates*) represent operations (of both input and computational type) and whose arcs represent data dependencies.

Trough this thesis we study the optimization of the implementation of a computation which has been specified in terms of a CDAG. The main advantage of the CDAG model for the present investigation is that it specifies neither the order in which the operations have to be executed nor the memory locations where data have to be stored. We leave to the implementor essentially two degrees of freedom: the definition of the *schedule of execution* of the operations, possibly including repetitions (i.e., *recomputations*), and the memory management, that is, the assignment of a memory location to each value produced in the computation during the time between the generation and last use of that value.

We now introduce the notation for CDAGs used by Bilardi and Peserico in [9], which we will use trough the thesis.

Definition 1.2 (Computational Directed Acyclic Graph). *A computation directed acyclic graph (CDAG) is a 4-tuple* (I, V, E, O) *of finite sets such that:*

- I *is the set of* input vertices*;*
- V *is the set of* operation vertices*;*
- *all vertices in* V *have at least one incoming edge;*
- $I \cap V = \emptyset$*;*
- $O \subseteq I \cup V$ *is the set of* output vertices*;*
- $E \subseteq (I + V) \times V$ *is the set of directed edges;*
- $G\left(I \cup V, E\right)$ *is a directed acyclic graph (DAG).*

In the following, we will use the simplified notation $G(I \cup V, E)$, or the shorter G, for the CDAG. All the graphs discussed in this thesis are CDAGs unless otherwise stated. Informally, with each vertex in $I \cup V$ we associate a value: for a vertex in I, the value is externally supplied and hence considered an input to the computation; for a vertex in V, the value is the result of an operation whose operands are provided by the predecessors of that vertex. The set O denotes which values, among all the ones being input or computed, form the desired output set. We say that two vertices u and v in $I \cup V$ are *adjacent* in G if there is an edge connecting them. For every directed edge (u, v) in E we say that u is a *predecessor* (or *immediate predecessor*, *parent*) of v ($u \prec v$), and v is a *successor* (or *immediate successor*, *child*) of u ($v \succ u$). We denote the set of all the predecessor of a vertex v by $pa(v)$ and the set of all its successors by $ch(v)$. The set $pa(v)$ represents all the operands of the operation that produces v and the set $ch(v)$ represents all the operation to whom v participates as an operand.

For a given vertex $v \in I \cup V$ its *in-degree* (resp., *out-degree*) $d^-(v)$ (resp., $d^+(v)$) it the number of its predecessors (resp., successors) $d^-(v) = |pa(v)|$(resp., $d^+(v) = |ch(v)|$. In this thesis we assume that vertices of in-degree (resp., out-degree) equal to zero constitute the set I of input vertices (resp., $O \subseteq I \cup V$ of the output vertices) of the CDAG. The *total-degree* of a vertex v, denoted as $d(v)$ corresponds to the total number of its adjacent vertices: $d(v) = d^-(v) + d^+(v)$. The maximum in-degree (resp., out-dregree) of a CDAG G is defined as $d^- = \max_{v \in V} (d^-(v))$ (resp., $d^+ = \max_{v \in I \cup V} (d + -(v)))$. Finally the maximum degree of G is defined as $d = \max_{v \in I \cup V} (d^-(v) + d^+(v))$.

A *path* p between two distinct vertices u and v in $I \cup V$ is a sequence of distinct vertices in which the first vertex is u, the last one is v and two consecutive vertices are connected by an edge, that is $p = (v_0 = u, v_1 \ldots, v_{m-1}, v_m = v)$ where (v_{i-1}, v_i) are edges in E for $i = 1, \ldots, m$ and $v_i \neq v_j$ for all $i \neq j$. We say that a path between two distinct vertices u and v in V is *directed* if all the directed edges in the path point at the direction toward v. We say that u is an *ancestor* of v ($u \prec^\star v$) and v is a *descendant* ($v \succ^\star u$) of u if there is a directed path from u to v in G. The set of all ancestors of v will be denoted as $an(v)$. The *depth* of a given CDAG corresponds to maximum length of any directed path connecting an input vertex to an output vertex.

Topological partitions and topological orderings of CDAG

For a given CDAG $G(I \cup V, E)$, a family of subsets of $I \cup V$, $\{V^{(1)}, V^{(2)} \ldots, V^{(i)}\}$, is called a *partition* of G if $\cup_{j=1}^{i} V^{(j)} = I \cup V$ and for every pair $a, b \in \{1, 2, \ldots, i\}$ we have $V^{(1)} \cap V^{(b)} = \emptyset$.

Definition 1.3 (Topological partition of a CDAG). *For a given CDAG $G(I \cup V, E)$, a family of subsets of $I \cup V$, $\{V^{(1)}, V^{(2)} \ldots, V^{(i)}\}$, is called a partition of G if it is a partition of G and for any $k \in \{1, 2, \ldots, i\}$ no vertex in $\cup_{j=k}^{i} V^{(j)}$ is a predecessor of a vertex in $cup_{j=1}^{k} V^{(j)}$.*

Let us consider all possible permutations (or orderings) of the vertices of a given CDAG G.

Definition 1.4 (Topological ordering). *A permutation π of the vertices of a CDAG $G(I \cup V, E)$ is a* topological ordering *(or permutation) of G if and only if any prefix-suffix partition of π is a topological partition of G.*

1.2 CDAG computations

A *computation* (or *schedule*) of $G(I \cup V, E)$ specifies a particular scheduling of the operations associated with its vertices, which satisfies data dependencies, and a particular memory management.

A *standard computation* of a CDAG $G(I \cup V, E)$ starts with the values of all input vertices stored in memory and must calculate the values of all output vertices by performing a sequence of *vertices evaluations* which correspond each to the execution of the operation associated to a vertex v, provided that all the vertices in $pa(v)$ are in memory. Input values can be removed from the memory but once they have been removed they cannot be recalled in memory. At the end of the computation all values of the output vertices must have been evaluated and stored in memory.

We will consider also another class of computations called *free-input computations*. A free-input computation starts with an initially empty memory, and every time an input value is needed it can be produced invoking a special load instruction. Furthermore, it is not necessary to maintain the value of the computed outputs stored in memory once they have been computed.

Read-once computations capture aspects of both standard and free-input computations. One such computation starts with an initially empty memory, and every time and each input value can be obtained once invoking a special load instruction.

In general it is possible that during a computations the same intermediate result is computed more than once. Doing so could allow to reduce the number of elements which have to be maintained in memory by the program during its execution at the cost of a possible increase in the number of computational steps required. Any such computation is particularly useful in all those situations in which the main priority is given to achieving the minimum execution time. We refer to computations in which no intermediate result is computed more than once as *computations with no recomputation* for short, *nr-computations*. The nr-computations of a CDAG are in one-to-one correspondence with the possible *topological orderings* of its vertices. It should be noted that in free-input nr-computations, the input values can be obtained just once by using the special load instruction (as in the read-once class). The key observation concerning this class of computations is that once an input value is loaded in memory or an intermediate result is calculated, then said value must remain available until the result of each operation which uses it as an input argument has been evaluated. In our work we will study how recomputation can affect both the performance and the analysis of straight line programs.

Chapter 2

Studying the space complexity of CDAGs using the visit method

In most computations the memory space available, be it the number of CPU registers or the cache memory size, is not sufficient to hold all the data on which a program operates. The same memory locations must be reused or the available space must be increased leading respectively to an increase or to a reduction of the number of the necessary computational steps (time). In this and the next chapter, we study CDAG computations in the *RAM model* [58] with a memory of unbounded size whose cells are addressed by natural numbers starting from 0. We refer as *size of the memory* to the number of *words* which can be stored in the memory, where we assume that one word can be stored in one memory cell. In this chapter, we focus on the study of the minimum space requirements for straight line algorithms expressed by means of a CDAG.

2.1 Space Complexity of a CDAG

Definition 2.1 (Space Complexity of a CDAG). *The* space complexity *of a given computational directed acyclic graph $G(I \cup V, E)$, denoted by $S(G)$ is defined as the minimum memory space strictly required by any* standard computation *of G.*

Since in standard computations all input values need to be stored in memory in memory at the start of the computation and all the output values need to be stored in memory at the end of it, for every CDAG G the following lower bound holds:

$$S(G) \geq \max\{|I|, |O|\}.$$

We can formalize an analogous concept for free-input computations:

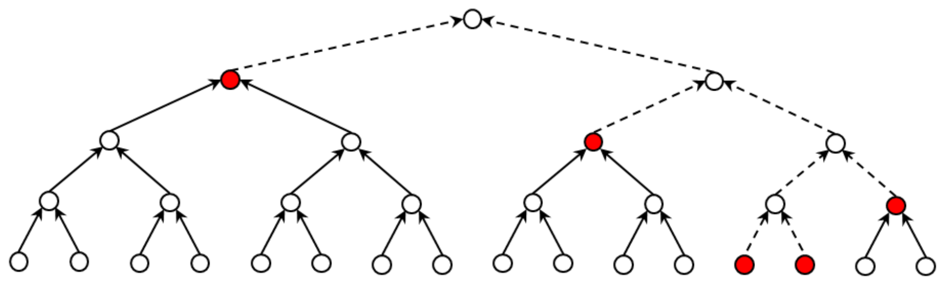


Figure 2.1: Pebble game played on a binary tree CDAG

Definition 2.2 (Free Input Space Complexity of a CDAG). *The* space complexity *of a given computational directed acyclic graph $G(I \cup V, E)$, denoted by $S_{free}(G)$ is defined as the minimum memory space strictly required by any* free-input computation *of G.*

Clearly, for any given CDAG G we have that $S(G) \geq S_{free}(G)$. In our work we will mostly focus on the analysis of the free-input space complexity of CDAGs. It is possible to define a concept that captures the minimum space requirement of nr-computations as follows:

Definition 2.3 (Space complexity of a CDAG for computations with no recomputation). *The space complexity (resp., free-input space complexity) of a given CDAG $G(I \cup V, E)$ for computations without recomputation, denoted by $S_{nr}(G)$ (resp., $S_{free-nr}(G)$) is defined as the minimum memory space strictly required by any standard (resp., free-input) nr-computation of G.*

2.1.1 The pebble game

The *pebble game* (also called *black pebble game*), introduced by Paterson and Hewitt in [47] is a simple yet power tool which allows us to study various types of computations and enables us to investigate the time and space requirements for the evaluation of a CDAG. The pebble game is a game played on directed acyclic graphs, which captures the dependencies of straight-line programs: pebbles are placed on vertices of a CDAG in a data-independent order to indicate that the value associated with a certain node is currently stored in memory. For a given CDAG $G(I \cup V, E)$, the rules for the pebble game are the following:

(R0) *Initialization*: at the beginning of the game no vertex is carrying a pebble

(R1) *Input*: a pebble can be placed on an input vertex in I at any time

(R2) *Computation step*: a pebble can placed on (or slided to) any non-input vertex in V only if all its immediate predecessors carry pebbles

(R3) *Pebble deletion*: a pebble can be removed at any time

(R4) *Goal*: each output vertex must be pebbled at least once

The placement of a pebble on an input vertex (rule (R1)) models the loading in memory of the input data, while the placement of a pebble on a non-input vertex corresponds to the computation of the value associated with the vertex (rule (R2)). The removal of a pebble (rule (R3)) models the deletion or the overwriting of the value previously stored in memory.

Allowing pebbles to be placed on input vertices at any time (rule R1) reflects the assumption that inputs are readily available. This, together with the rule according to which at the beginning of the game no pebble is placed on the CDAG (rule (R0)), is the key condition that associates the executions of the pebble game to the free-input computations rather than to the standard computations. This condition creates a certain distance between the pebble game and most of practical situations in which all input values must actually reside in memory. The model, however, maintains however a high degree of interest since it provides a lower bound to the space complexity when operating with a high degree of freedom.

The condition that all immediate predecessor vertices should carry pebbles in order to place a pebble on a vertex (rule (R2)) models the natural requirement that an operation can be performed only if all its arguments are available in main memory. Moving (or *sliding*) a pebble to a vertex from an immediate predecessor reflects the design of CPUs that allow the result of a computation to be placed in a memory location holding an operand.

Finally, rule (R4) represents the fact that all the output values of the corresponding straight line program have to be computed.

The execution of the rules of the pebble game on the vertices of a CDAG G is called a *pebble strategy* or simply *pebbling*. A pebbling is said to be *complete* if it satisfies rule (R4) (i.e. it is complete). Each complete pebble strategy corresponds to a free-input computation for G. Each step of the computation is associated to each placement of a pebble, ignoring steps on which pebbles are removed. The steps of a strategy can then be numbered consecutively from 1 to T, where T corresponds to the *time* required by the strategy. For any given CDAG $G(I \cup V, E)$, any pebbling strategy will require at least $|I \cup V|$ steps. The space requirement of a pebbling strategy is the minimum number S of pebbles which are necessary for the execution of the strategy itself. A strategy has *minimum spate requirement* (or is *minimal*) if no other strategy has lower space requirement. The free-input space complexity of a CDAG G corresponds to the space requirement of a minimum pebbling strategy for the pebble game played on G.

Pebbling strategies with no repebblings

nr-computations correspond to pebbling strategies in which each vertex receives a pebble just once, and remains pebbled until all its successors have been pebbled as well. This fact provides a very strong insight on the "*lifespan*" of a pebble placed on a specific vertex of a CDAG and, correspondingly, of data stored in memory. We refer to this class of pebbling strategies as *nr-pebblings* (which is short for "*pebbling strategies with no repebblings*").

The nr-pebblings of a CDAG are in one-to-one correspondence with the possible *topological orderings* of its vertices, and therefore with the corresponding nr-computations of the CDAG.

2.1.2 Pebbling technicalities

Unconditional and Conditional pebblings

In some setting, it may prove useful to relax rule (R0) and allowing some pebbles to be already placed on the CDAG at the beginning of the game. These pebblings are referred in literature as *conditional pebblings*, while pebblings for which no pebble is assumed to be placed in the CDAG are referred as *unconditional* [45]. In our study we use the assumption that no pebble is placed in the graph at the beginning of the computation (i.e. we focus on unconditional pebblings) just as in the original pebble game.

Visiting and Permanent pebblings

A variation of the pebbling game according to which in order to achieve a complete pebbling all the output vertices must hold a pebble is known in literature as *permanent pebbling game* [45]. In this thesis we refer to the original version of the pebbling game as previously presented, sometimes referred as *visiting pebbling* game, for which a in a complete pebbling strategy all vertices are to be pebbled at least once, but it is not required to maintain a pebble on all output vertices.

Variations of the pebble game

Variations of the basic pebble game have been introduced to study different aspects of CDAG computations. Hong and Kung [37] introduced the *red-blue pebble game*, which we describe in Chapter 4, which remains to this date the main point of departure of most lower bound analysis for hierarchical memory performance. The *black-white pebble game* was introduced by Cook and Sethi [18] to study the

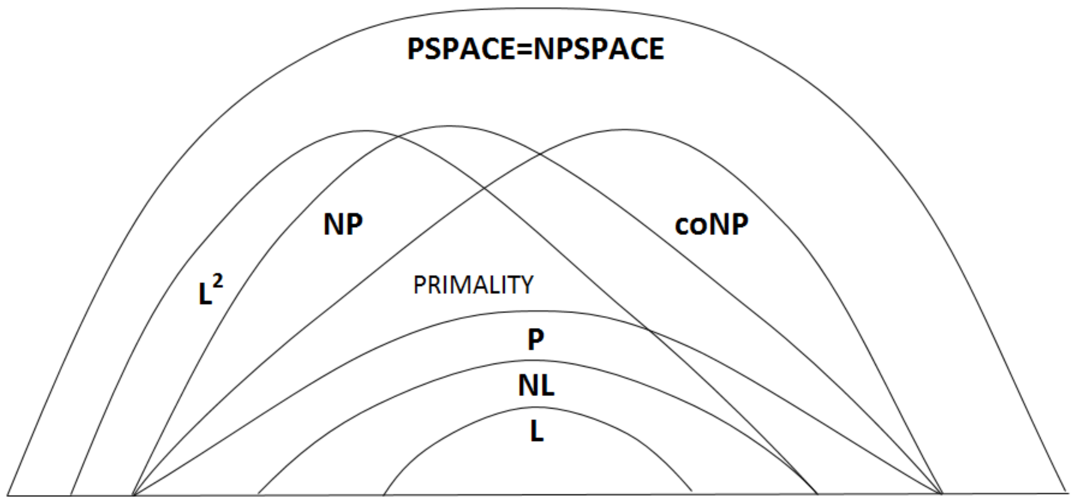


Figure 2.2: Relationship among complexity classes

space requirement of CDAG computations which can utilize non-deterministic steps. A two-person game introduced by Venkateswaran and Tompa [69] models parallel complexity classes.

2.1.3 Hardness of the space complexity problem

It is generally very hard to determine the space complexity of a CDAG. Rather than a general approach, specific pebbling strategies have to be tailored on the particular CDAG structure [58]. In terms of the traditional hierarchy of complexity classes, the problem of finding the minimum number of pebbles needed to pebble a CDAG can be modeled as a language consisting of strings each of which contains the description of a CDAG G, a vertex $v \in V$ and an integer S with the property that v can be pebbled with S or fewer pebbles. Gilbert, Lengauer, and Tarjan [32] and Loui [43] have shown that the languages associated with minimal pebblings of CDAGs are PSPACE-complete. PSPACE is the class of decision problems that are decidable by a Turing machine in space polynomial in the size of the input and are potentially much more complex of problems in P. The hardest problems in PSPACE are PSPACE-complete problems, in the sense that any PSPACE problem can be reduced to a PSPACE-complete problem in polynomial time by a Turing machine. Although it is not know whether this is the case, these problems are widely suspected to be outside of the more famous complexity classes P and NP, but that is not known. PSPACE-complete problems, however, are currently as infeasible as NP-complete problems, since both are solvable in exponential time and polynomial space [58].

2.2 The marking rule approach

When studying the space complexity of CDAGS, the possibility of evaluating the same vertex multiple times, greatly complicates the analysis with respect to what constitutes a valid schedule and to what must be in memory at any given step of the schedule. In [10], Bilardi et al. introduced the *Marking Rule* technique, a general framework for obtaining lower bounds to the free-input space complexity of a CDAG. In this method, the authors aim to show a correspondence between each possible computation of the CDAG to a permutation of its vertices, which in general is not a topological ordering of its vertices. Such correspondence is obtained using a *marking rule* which is a criterion to associate each vertex to a family of subsets of it successors.

2.2.1 Description of the method

A marking rule for a given CDAG G is a function $f : I \cup V \to 2^{2^V}$ for which:

- $q \in f(v) \implies q \subseteq ch(v)$;
- $v \in O \implies f(v) = \{\emptyset\}$;
- $v \in V \backslash O \implies \emptyset \notin f(v)$.

The marking rule associates every vertex $v \in I \cup V$ to a *family* of subsets of its successors. We refer to $f(v)$ as the *enabling family* of v, and to each $q \in f(v)$ as an *enabling set* for v. Let $G(I \cup V, E)$ be a CDAG such that $|I \cup V| = n$, and let $\phi = \phi_1\phi_2 \ldots \phi_n$ be a permutation of all the vertices in $I \cup V$ so that $\{\phi_i : 1 \leq i \leq n\} = I \cup V$. ϕ is a *legal* f-marking of G for a marking rule f if and only if for every $1 \leq i \leq n$ there exist $q \in f(\phi_i)$ such that $q \subseteq \{\phi_i : i \leq j \leq n\}$.

The i-boundary of ϕ is then defined as the set $B^f_\phi(i)$ of all the vertices $v \in V \backslash O$ that satisfy the following properties:

- $v \in \{\phi_1, \ldots, \phi_i\}$
- there exists $q \in f(v)$ such that $q \subseteq \{\phi_{i+1}, \ldots, \phi_n\}$.

Where $B^f_\phi(i)$ represents the set of vertices $v \in V \backslash O$ such that $v\phi_{i+1} \ldots \phi_n$ is the suffix of a legal f-marking of G.

In [10], a relation is the shown between the space complexity of the free-input computations of a CDAG G and the size of the boundaries of its f-marking. Let F_G denote the set of marking rules for G and $\Phi_f(G)$ the set of legal f-markings of G.

Theorem 2.4 (Lower bound for space complexity - Theorem 1 from [10]). *For any given CDAG $G(I \cup V, E)$ we have:*

$$S_{free}(G) \geq \max_{f \in F_G} \min_{\phi \in \Phi_f(G)} \max_{1 \leq i \leq |I \cup V|} \left| B^f_\phi(i) \right|. \tag{2.1}$$

Proof Consider an arbitrary marking function $f \in F_G$ and a T-step free-input parsimonious computation $\mathcal{C}$ for $G(I \cup V, E)$. Let v_t be the vertex evaluated at step t of $\mathcal{C}$, for $1 \leq t \leq T$. It is possible to obtain the corresponding f-marking of $\mathcal{C}$ $\phi = \phi_1\phi_2 \dots \phi_n$ by sweeping backward the steps of the computation using the following loop:

```
$j = n$;
for $t = T$ down-to 1 do
    if ($v_t \notin \{\phi_{j+1}, \dots \phi_n\}$) and ($\exists q \in f(v_t) : q \subseteq \{\phi_{j+1}, \dots \phi_n\}$)
    then $\phi_j = v_t$; $j = j - 1$;
```

It can be easily verified that the sequence ϕ obtained at the end of the loop is indeed a f-marking for $G(I \cup V)$. In order to prove the accuracy of the bound, it must be shown that, fixed an index i, $1 \leq i \leq n$ with $\phi_i = v_t$ for some t, the value of the vertex in $B^f_\phi(i)$ must actually be in memory at the end of step t of the computation $\mathcal{C}$. Let $v \in B^f_\phi(i)$. The definition of $B^f_\phi(i)$ and the fact that the computation $\mathcal{C}$ being used is parsimonious, implies that there exist two indices t_1 and t_2, with $1 \leq t_1 \leq t \leq t_2 \leq n$, such that $v_{t_1} = v$, $v_{t_2} \in ch(v)$, and $v_j \neq v$ for every $t_1 \leq j \leq t_2$. As a consequence, the value of v computed at step t_1 of $\mathcal{C}$ is used to compute v_{t^2} and therefore it must reside in memory at the end of step t. Since i was chosen arbitrarily, it is possible to conclude that the space required by $\mathcal{C}$ is not less than $\max_{1 \leq i \leq n} \left| B^f_\phi(i) \right|$. The theorem follows by minimizing over all possible $\phi \in \Phi_f(G)$ and by maximizing over all possible $f \in F_G$. □

Note that the lower bound obtained is generally not tight: while considering the vertex ϕ_i of a given f-marking it can be said that all the nodes that belong to the boundary $B^f_\phi(i)$ must be located in memory immediately after the evaluation of ϕ_i, it is not however possible to conclude that all the nodes that are in memory at that step of the computation will actually appear in the boundary $B^f_\phi(i)$. While in topological ordering each vertex must appear before all of his successors, in generic f-marking this constraint is relaxed, with every vertex appearing before at least one of its enabling sets.

2.2.2 The *singleton* and *topological* marking rule

One disadvantage of this approach is given by the high number of possible marking rules to be analyzed $|F_G|$. Among these, however, there are two rules which are of *natural* interest and particular importance:

- the *singleton marking rule* $f^{(sing)}$ which associates each vertex to an enabling family composed by the singleton sets containing each one of its successors:

$$f^{(sing)}(v) = \begin{cases} \{\{u\} | u \in ch(v)\} & \forall v \in V \backslash O \\ \{\emptyset\} & \forall v \in O \end{cases}$$

  For any $f \in F_G$ we have $\Phi_f(G) \subseteq \Phi_{f^{(sing)}}(G)$. For any $f \in F_G$ and any $\phi \in \Phi_f(G)$ we have:

$$\max_{1 \leq i \leq n} \left| B_{\phi}^{f^{(sing)}}(i) \right| \geq \max_{1 \leq i \leq n} |B_\phi(i)|.$$

- the *topological marking rule* $f^{(top)}$ which associates each vertex to to an enabling family composed by just the set of all its predecessors:

$$f^{(top)}(v) = \begin{cases} \{ch(v)\} & \forall v \in V \backslash O \\ \{\emptyset\} & \forall v \in O \end{cases}$$

  $\Phi_{f^{(top)}}(G)$ corresponds to the set of all topological orderings of the vertices of the CDAG, and therefore to the set of all possible nr-pebblings. For any $f \in F_G$ we have $\Phi_{f^{(top)}}(G) \subseteq \Phi_f(G)$. For any $f \in F_G$ and any $\phi \in \Phi_f(G)$ we have:

$$\max_{1 \leq i \leq n} \left| B_{\phi}^{f^{(top)}}(i) \right| \leq \max_{1 \leq i \leq n} |B_\phi(i)|.$$

$f^{(sing)}$ and $f^{(top)}$ are somehow at the opposite ends of the spectrum of all possible marking rules. However, they both exhibit a regular and general criterion in their definition, while using intermediate rules may require an analysis tailored on specific characteristics of a given CDAG.

2.2.3 Application of the marking rule approach

In this section, we show how the marking rule approach can be used to obtain a novel lower bound to the space complexity of a specific family of CDAGs. We start by introducing the *building blocks* of our family of CDAGs.

Definition 2.5 (Superconcentrator CDAG). *A CDAG $S(n)$ with n input vertices and m output vertices is said to be an n-superconcentrator if $n = m$ and for any couple of subset A and B of the input and output vertices respectively with $|A| = |B| = c$, there exist c vertex disjoint paths in G connecting each vertex in A to a vertex in B.*

Superconcentrators were originally introduced by Leslie Valiant [68] which proved that n-superconcentrators of linear size (intended as the number of vertices and edges) with respect to n can actually be built. Explicit constructions which ensures depth $O(\log n)$[1] and constant degree for the vertices for linear-size n-superconcentrators were later provided by Pippenger [52] and Gabber and Galil [30]. We shall now study the space requirement of the family of *superconcentrators stack* CDAGs which is defined as follows:

Definition 2.6 (n-superconcentrators r-stack). *An n-superconcentrators r-stack CDAG, denoted as $ST(n,r)$, is obtained by composing r linear-size n- superconcentrators $S_1(n), S_2(n), ..., S_r(n)$ by merging each output vertex of S_i with a distinct input vertex of S_{i+1} for $1 \leq i < r$. The input vertices of $S_1(n)$ and the output vertices of $S_r(n)$ act respectively as the input and the output vertices of the complete CDAG.*

The family of superconcentrators stack CDAGs were studied by Lengrauer and Tarjan in [40]. In the paper, the authors discuss trade-offs between the memory space and the time necessary for the computation of such CDAGs.

We now discuss a property of $ST(n,r)$ which will be the key to obtain the lower bound on the space complexity using the marking rule approach.

Lemma 2.7. *Let I and O denote respectively the set of input and output vertices of $ST(n,r)$. Let p be the set of vertices which constitute a path from a vertex $v_i \in I$ to an output vertex $v_o \in O$, in $ST(n,r)$. It is possible to partition $I \setminus \{v_i\}$ in two sets A and B such that:*

- *there exists $|A| \geq \min\{n-1, r\}$ paths, denoted as P_A, which connect each vertex in A to a vertex in p which are vertex-disjoint amongst themselves, except for the vertices in p;*
- *there exists $|B| \geq n - |A| - 1$ vertex-disjoint paths, denoted as P_B, which connect each vertex in B with an output vertex of $ST(n,r)$. The paths in P_B are vertex-disjoint with respect to p and all the paths in P_A.*

Proof The proof is by induction on r. In the following, we will assume $n \geq 2$.

[1] Through this thesis, we use the notation "$\log x$" as short for "$\log_2 x$".

Base: For $ST(n, 1)$ let us consider the vertex v_i which is the first vertex of the path p which connects v_i to v_o. Let us consider the subset $C = I \setminus \{v_i\}$ and an arbitrary subset D of the output vertices of $ST(n, 1)$ such that $v_o \in D$ and $|D| = |C|$. For the superconcentrator property there will be $|C|$ vertex disjoint paths connecting each vertex in $|C|$ to a vertex in $|D|$. Among these paths there will be at least one sharing a vertex with p (the path form a vertex $v_j \in C$ to v_o). Therefore there will be one or more paths P_A starting form vertices in $A \subseteq C$ to vertices in p which do not share any vertex besides those in p itself with $|A| \geq 1$. Additionally there will be $|C| - |A| = n - 1 - |A|$, vertex-disjoint paths P_B from vertices in $B = C \setminus A$ which do not have any vertex in common with p. Since $r = 1$ for the base case the statement is verified.

Induction: Proceeding in the proof we will assume that the statement of the lemma is verified for $ST(n, r - 1)$ for $r > 1$ and we will verify that the lemma holds for $ST(n, r)$ as well.

Let us consider the sub-CDAG $ST(n, r)'$ constituted by the first $r - 1$ n- superconcentrators sub-CDAGs starting from the one whose input vertices correspond to those of the whole $ST(n, r)$. We call p' the sub path of p from a vertex v_i in I to an output vertex v_o' of $ST(n, r)'$. Since $ST(n, r)'$ is in fact a stack of $r - 1$ linear-size n-superconcentrators the inductive hypothesis applies for the path p' and then there will be the sets A' and B' of inputs of $ST(n, r)'$ (and therefore $ST(n, r)$ as well) such that there will be $|A'| \geq \min\{\max\{\frac{n}{2}, n - r - 1\}, r - 1\}$ paths P_A' from vertices in $A' \subseteq I$ to vertices in the sub-path p' which are vertex-disjoint amongst themselves except for the vertices in p'. Additionally there will be $|B'| = n - |A'|$ paths P_B' from vertices in $B' \subseteq I$ to outputs of $ST(n, r)'$, to whom we will refer as C, which are vertex disjoint amongst themselves and the paths in P_A' and p'.

If $|A'| = n - 1$, then we can assume $A = A'$, thus $P_A = P_A'$ and the statement is therefore easily verified for $ST(n, r)$ as well. Otherwise $|A'| \geq r - 1$ and $|C| \geq 1$. Let D be a subset of the output vertices of $ST(n, r)$ with $v_o \in D$ and $|D| = |C|$. Since vertices in C are the input vertices of an n-superconcentrator (the r-th superconcentrator of the stack $ST(n, r)$), from the superconcentrator property follows that there will be $|C| = |D|$ vertex-disjoint path in $S_r(n)$ connecting each vertex in C to a distinct vertex in D. Since $v_o \in D$, at least one of these paths will encounter a vertex in p (at least in v_o). p to $|\pi| - |\pi'|$ additional paths may in fact arrive to vertices in p while being vertex disjoint amongst themselves, we refer to these paths as P_{C^A} and to the input vertices of $S_r(n)$ from which each of these paths is starting as $C^A \subseteq C$. Besides from the paths already mentioned, for the superconcentrator property there will be $|C| - |C^A|$ paths from vertices in $C^B = C \setminus C^A$ to output vertices of $S_r(n)$ (an therefore of $ST(n,r)$) as well, to whom we will refer as P_{C^B}, which do not encounter any vertex in p and are vertex disjoint amongst themselves and all the paths in P_{C^A}.

Since for the inductive hypothesis each vertex in C is connected to a vertex in $B' \subset I$ by the paths in P'_B, by composing each path in P_{C^A} with the corresponding paths from $B'' \subseteq B'$ to C^A we obtain the paths $P_{A''}$starting from $B'' \subseteq I$ to vertices in p with $|P_{A''}| \geq 1$. Similarly, by composing each path in P_{C^B} with the corresponding paths from $B' \setminus B''$ to C^B we obtain the vertex-disjoint paths P_B starting from $B' \setminus B'' \subseteq I$ and connected to the output of $ST(n,r)$. By construction, paths in P'_A will not have any vertex in common with the paths in $P_{A''}$. By putting them together we obtain the set of paths $P_A = P'_A \cup P_A$, starting form vertices in $A = A' \cup B''$ and connected to vertices in p which do not have any vertex in common besides those in p. Since for the inductive hypothesis $|A'| \geq \min\{n, r-1\}$ and it was proven that $|B''| \geq 1$, we can conclude that $|A| \geq r - 1 + 1$ and therefore $|A| \geq r$. Finally, we have the set P_B of vertex-disjoint paths starting from $B = B' \setminus B'' \subseteq I$ to output vertices of $ST(n,r)$ which do not share any of the vertices on the paths in P_A, with $|B| = |B'| - |B''| = n - 1 - |A'| - |B''| = n - 1 - |A|$.

□

We will now prove the lower bound on the space complexity for a $ST(n,r)$ CDAG by using the marking rule approach and the previously presented lemma.

Theorem 2.8 (Lower bound free-input space complexity for $ST(n,r)$)**.** *For any given r-stack of n-superconcentrator CDAG $ST(n,r)$ we have:*

$$S_{free}\left(ST(n,r)\right) \geq \min\{n-1, r\} + 1 = \min\{n, r+1\}. \tag{2.2}$$

Proof Let us consider now any $f^{(sing)}$-marking $\phi = \phi_1\phi_2 \ldots \phi_N$ of $ST(n,r)$, where N is the total number of vertices. In particular, if the superconcetrators being used in the construction of the stack are linear we have $N = \mathcal{O}\left(rn\right)$. Let $\phi' = \phi_1\phi_2 \ldots \phi_{N'}$ be the longest prefix of ϕ such that ϕ' contains all input vertices of $ST(n,r)$. $\phi_{N'}$ will therefore be an input vertex. Given the structure of $ST(n,r)$ and the properties of the singleton marking rule, in the suffix $\phi_{N'+1} \ldots \phi_N$ there will be all the vertices of a path p from $v_i = \phi_{N'}$ to an output vertex v_o which goes through all the r linear-size n-superconcentrators of $ST(n,r)$.

From Lemma 2.7 follows that there are $\Omega\left(\min\{n-1, r\}\right)$ vertex disjoint paths from vertices in $I \setminus \{v_i\}$, to vertices of the path p. Since all vertices in $I \setminus \{v_i\}$ are in the prefix and vertices in p are in the suffix every such path includes a node in $B_{\phi}^{f^{(sing)}}(N')$. From Theorem 2.4 we get:

$$S_{free}\left(ST(n,r)\right) \geq \min_{\phi \in \Phi_{f^{(sing)}}(ST(n,r))} \max_{1 \leq i \leq n} \left|B_{\phi}^{f}(i)\right| \geq \min\{n-1,r\}+1 = \min\{n,r+1\}$$

□

2.3 Visits of a CDAG

While the marking rule approach is quite versatile and powerful, the question of whether it is possible to attain *significant* lower bounds to the space complexity (i.e., the obtained lower bound is asymptotically correspondent to the true space complexity), remains open. In order to investigate this aspect of the marking rule technique we developed the concept of *visit* of a CDAG. In this section we describe how visits of a CDAG relate to the marking method and to the space complexity of a CDAG. We then use this method to study the bounds attainable using the $f^{(sing)}$ and $f^{(top)}$ marking rules.

2.3.1 Definition of Visit of a CDAG

A *visit* of the CDAG $G(I \cup V, E)$ is an $|I \cup V| = n$-step traversal of G which reaches all its vertices proceeding according to a series of "*legal*" steps. The i-th step, with $1 \leq i \leq n$, is said to be *legal* if the vertex $v_i \in V$ visited during the i-th step is *enabled for the visit* after the previous $1, \ldots, i-1$ steps (if any). In order to establish the conditions which enable a vertex to be visited we will use the notion of *visit rule.* A visit rule for a given CDAG G is a function $h : I \cup V \rightarrow 2^{2^V}$ for which:

- $q \in h(v) \Longrightarrow q \subseteq pa(v)$;
- $v \in I \Longrightarrow h(v) = \{\emptyset\}$;
- $v \in V \Longrightarrow \emptyset \notin h(v)$.

A visit rule h for G associates every vertex $v \in I \cup V$ to a family of subset of $pa(v)$ such that if $v \in I$, $h(v)$ contains only the empty set, and if $v \in V$ then $h(v)$ can not contain the empty set and must contain at least one subset of $pa(v)$. We denote the set of all possible visit rules for a given CDAG G as H_G. A permutation

$\psi = \psi_1\psi_2 \dots \psi_n$ is an h-visit of G iff for every $1 \leq i \leq n$ there exists $q \in h(\psi_i)$ such that $q \subseteq \{\psi_1, \dots, \psi_{i-1}\}$. We denote as $\Psi_h(G)$ as the set of all h-visits on G.

The i-boundary of a h-visit ψ is defined as the set:

$$B^h_\psi(i) = \{v \in V \backslash \{\psi_1, \dots, \psi_{i-1}\} | \exists q \in pa(v) \ s.t. \ q \subseteq \{\psi_1, \dots, \psi_{i-1}\}\}.$$

$B^i_\psi(\psi_i)$ is the set of vertices which are enabled to be visited after the $i-1$-th visit step. We refer as the maximum boundary size of a h-visit ψ as B^h_ψ:

$$B^h_\psi = \max_{i \in \{1,\dots n\}} \left|B^h_\psi(i)\right|. \tag{2.3}$$

The input vertices must be handled with care: a straightforward application of the previous criteria would impose $B^h_\psi(i) \in \Omega(|I|)$. We will however consider a "*free-input*" model for our visits: input vertices, which are enabled to be visited since the beginning of the computation, do not participate in the boundary B^h_ψ.

In the following we will consider three main classes of visit rules:

- the *topological visit rule* $h^{(top)}$ which associates each vertex to an enabling family composed by just the set of all its predecessors:

$$h^{(top)}(v) = \begin{cases} \{pa(v)\} & \forall v \in V \\ \{\emptyset\} & \forall v \in I \end{cases}$$

  $\Psi_G(h^{(top)})$ corresponds to the set of all topological orderings of the vertices of the CDAG. For any $h \in H_G$ we have $\Psi_G(h^{(top)}) \subseteq \Psi_G(h)$. For any $h \in H_G$ and any $\psi \in \Psi_G(h)$ we have:

$$\max_{1 \leq i \leq n} \left|B^{h^{(top)}}_\psi(i)\right| \leq \max_{1 \leq i \leq n} |B_\psi(i)|.$$

- the *singleton visit rule* $h^{(sing)}$ which associates each vertex to an enabling family composed by the singleton sets containing each one of its predecessors:

$$h^{(sing)}(v) = \begin{cases} \{\{u\}|u \in pa(v)\} & \forall v \in V \\ \{\emptyset\} & \forall v \in I \end{cases}$$

  For any $h \in H_G$ we have $\Psi_G(h) \subseteq \Psi_G(h^{(sing)})$. For any $h \in H_G$ and any $\psi \in \Psi_G(h)$ we have:

$$\max_{1 \leq i \leq n} \left|B^{h^{(sing)}}_\psi(i)\right| \geq \max_{1 \leq i \leq n} |B_\psi(i)|.$$

- *intermediate* visit rules h which do not behave as previous rules.

2.3.2 The *reach* and *enabled reach* concepts

Given a vertex $v \in I \cup V$ we define its *reach* $R(v)$ as the set of all its descendants:

$$R(v) = \{u \in V | \exists \langle v, u \rangle \in E \vee \exists \langle v', u \rangle \in E \ with \ v' \in R(v)\}.$$

We define an additional concept of reach called *enabled reach* with reference to a specific h-rule and to a set $\psi_{pre} \subseteq I \cup V$ to which we refer to as *pre-visit* (i.e., a prefix of a complete visit ψ). The h-enabled reach of a vertex $v \in V$ given a pre-visit ψ_{pre} is the subset of the descendants of v not already visited in ψ_{pre} which can be visited starting from v and ψ_{pre}:

$$R^h_{\psi_{pre}}(v) = \left\{ u \in R(v) | \exists q \in f(v) \ \ s.t. \ q \subseteq R^h_{\psi_{pre}}(v) \cup \psi_{pre} \right\}.$$

To give an useful intuition of the concept of h-enabled reach, we can think of it as the set of all the vertices which can be reached by an h-visit starting form the vertex v and a pre-visit ψ_{pre}. The enabled reach exhibits the following crucial property:

Lemma 2.9. *Let G be any n-vertex CDAG $G(I \cup V, E)$. For any $v \in I \cup V$ and for any given visit rule h, let ψ_{pre} be a legal sub h-visit from any input vertex in I to v. Let $G'(V', E')$ be the sub-CDAG induced by the h-enabled reach of v given ψ_{pre}:*

- $V' = R^h_{\psi_{pre}}(v)$
- $I' = \{v\}$
- *E' the subset of the edges in E which have both endpoints in V'*

The following conditions hold:

1. *Any visit ψ' of G' which is legal with respect to h, does not enable the visit of any vertex not in $R^h_{\psi_{pre}}(v)$.*
2. *Consider the partial visit $\psi_{pre}\psi'$ of length $m \leq n$ and let it be the prefix of a complete h-visit ψ of the entire CDAG G. Then for $i = 1, 2, \ldots, m$, the vertices appearing in the boundary $B^h_\psi(i)$ are either vertices in $\psi_{pre} \cup \psi'$ which have not yet been visited by the i-th step, or vertices in $V \setminus (\psi_{pre} \cup \psi')$ which have an enabling subset entirely contained in ψ_{pre}.*

Proof

1. In order for a vertex $u \notin R^h_{\psi_{pre}}(v)$ to become enabled for being visited after any visit of G', one of the enabling subsets of u must be contained in $\psi_{pre} \cup R^h_{\psi_{pre}}(v)$. If that is the case, according to the the definition of enabled h-reach, either $u \in \psi_{pre}$ and has therefore already been visited, or u must be in $R^h_{\psi_{pre}}(v)$.

2. From point (1) we have that any visit ψ' of G' which is legal with respect to h, does not enable the visit of any vertex not in $R^h_{\psi_{pre}}(v)$. By the definition of boundary none of these vertices can therefore appear in the boundary $B^h_{\psi}(i)$.

$\square$

It is interesting to observe that for the singleton marking rule the enabled reach and the reach of a vertex coincide (save for the vertices in the pre-visit).

2.3.3 Relation between markings and visits

In this section, we discuss the relation between the visits just described and the visit method discussed in the previous section.

Definition 2.10 (Reverse CDAG). *Given a computational directed acyclic graph $G(I \cup V, E)$ its reverse CDAG $G_R(I_R \cup V_R, E_R)$ is another directed graph on the same set of vertices such that:*

- $I_R = O$;
- $V_R = I \cup (V \setminus O)$;
- $O_R = I$;
- $E_R \subseteq (I_R \cup V_R) \times V_R$ *is the set of edges whose orientation is reversed with respect to G:*
$$E_R = \{(u, v) \in (I_R \cup V_R) \times V_R | (v, u) \in E\}$$
- $G_R(I_R \cup V_R, E_R)$ *is a directed acyclic graph.*

Note that G and G_R have the same number of vertices and edges. Since the orientation of the edges is inverted from G to G_R, for each vertex $v \in I \cup V$ we have that its in-degree $d^-(v)$ (resp., out-degree d^+) in G will be equal to the out-degree (resp., in-degree) of the same vertex in G_R. Correspondingly, the maximum in-degree (resp., out-degree) of G will correspond to the maximum out-degree (resp., in-degree) of G_R. Our definition is obtained from the more generic definition of *reverse directed graph* [24]. G_R is sometimes referred as the *converse* of G, because the reversal of arrows corresponds to taking the converse of an implication in logic [34], or as *transpose* of G because the adjacency matrix of the transpose directed graph is the transpose of the adjacency matrix of the original directed graph [21].

Definition 2.11 (Correspondence between marking and visit rules). *Given a CDAG $G(I \cup V, E)$ and its reverse $G_R(I_R \cup V_R, E_R)$ we say that a visit rule $h \in H_G$* corresponds *to a marking rule $f \in F_{G_R}$ if $\forall v \in I \cup V$ we have $h(v) = f(v)$.*

In the previous definition the role of G and G_R can be switched. Since all the visit rules in H_G and all the marking rules in F_{G_R} are distinct, one $h \in H_G$ corresponds to exactly one marking rule $f \in F_{G_R}$ and vice versa. In particular, we have that $h^{(sing)}$ (resp., $h^{(top)}$) on G corresponds to $f^{(sing)}$ (resp., $f^{(top)}$) on G_R. The correspondence between visit rules and marking rules implies a correspondence between the h-visit of G and the f-markings of G_R. For any corresponding pair (h, f) with $h \in H_G$ and $f \in F_{G_R}$, every visit $\psi \in \Psi_h(G)$ coincides with a marking $rev(\psi) \in \Phi_f(G_R)$, where $rev(\psi)$ denotes a permutation in which the elements appear in the opposite order of ψ. Furthermore, according to the respective definitions of boundary, we have that for any corresponding pair (ψ, ϕ) with $\psi \in \Psi_h(G)$ and $rev(\phi) \in \Phi_f(G_R)$ and h corresponding to f we have:

$$B^h_\psi(i) = B^f_{rev(\phi)}(i), \forall i \in \{1, \ldots, |I \cup V|\}$$

This implies that for any corresponding pair (h, f) with $h \in H_G$ and $f \in F_{G_R}$ we have:

$$\min_{\psi \in \Psi_h(G)} \max_{1 \leq i \leq |I \cup V|} \left|B^h_\psi(i)\right| = \min_{\phi \in \Phi_f(G_R)} \max_{1 \leq i \leq |I \cup V|} \left|B^f_\phi(i)\right| \tag{2.4}$$

The result in Theorem 4.7 can be restated in terms of visits:

Lemma 2.12 (Lower bound for space complexity based on visits). *For any given CDAG $G(I \cup V, E)$ we have:*

$$S_{free}(G) \geq \max_{h \in H_{G_R}} \min_{\psi \in \Psi_h(G_R)} \max_{1 \leq i \leq |I \cup V|} \left|B^h_\psi(i)\right|. \tag{2.5}$$

The proof follows from Theorem 4.7 and from the correspondence between makings an visits we just discussed.

Besides their intrinsic interest, visits on a CDAG provide also an alternative way to study the space complexity of the reverse CDAG. In the following sections we will study an upper bound to the boundary size achievable for visits using respectively the singleton and topological visit rules. These results will then provide an upper bound to maximum lower bound obtainable using the marking rule approach.

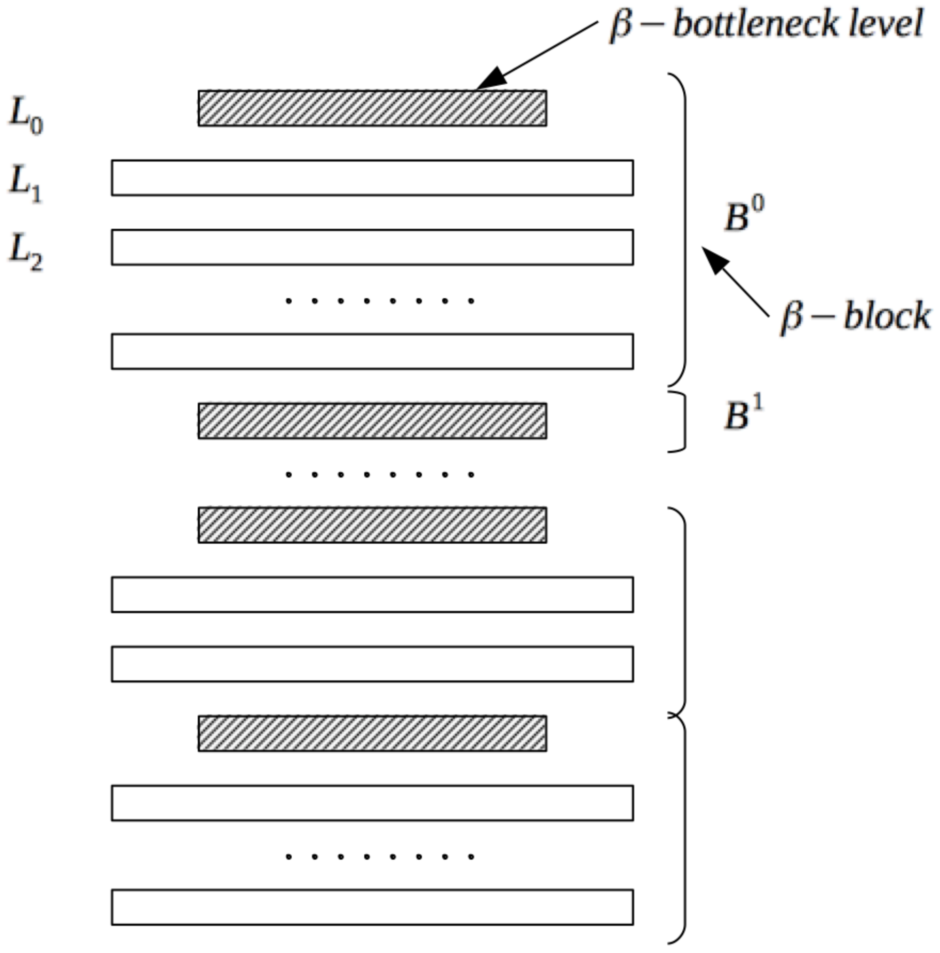


Figure 2.3: h-schedule of a CDAG and β-block partition

2.3.4 h -schedule of a directed acyclic graph

Given a CDAG $G(V, E)$ and a rule h a h-schedule $L^h(G)$ is a partition of the vertices of V in levels $L_1^h, \ldots, L_j^h$ built according the following inductive rule:

$$L_1^h = I$$

$$L_i^h = \left\{ v \in V | \exists q \in f(v) \ s.t. \ q \subseteq \bigcup_{k=1,\ldots,i-1} L_k^h \ \wedge \nexists q \in f(v) \ s.t. \ q \subseteq \bigcup_{k=1,\ldots,i-2} L_k^h \right\}.$$

The h-schedule for G can be easily built inductively starting with the initial level composed just by the input vertices of G. It is crucial to note that for a given DAG G and a given marking rule h, $L^h(G)$ is unique. The main idea behind the schedule is that each level L_i^h, with $1 \leq i \leq \ell$ "*shields*" the vertices in $\cup_{i=1}^{i-1} L_i^h$ from the vertices $\cup_{i=i+1}^{\ell} L_i^h$. This "*shielding*" condition corresponds to fact that for all vertices $v \in \cup_{i=i+1}^{\ell} L_i^h$ no enabling set of v is a subset of $\cup_{i=i+1}^{\ell} L_i^h$.

Let $\beta \in \mathbb{R}$ be an arbitrarily chosen integer value such that $1 \leq \beta \leq n$, we call $\beta-$ *bottlenecks* of a h-schedule of G all the levels L_i^h such that $\left|L_i^h\right| \leq \beta$. The number of non-$\beta$-bottleneck levels of a h-schedule ϖ is therefore upper bounded by $0 \leq \varpi < n/\beta < n$. The β-block partition of $L^h(G)$ is obtained partitioning the levels of $L^h(G)$ into β-blocks such that the i-th β-block contains all the levels whose index is smaller than the index of the i-th β-bottleneck level $L^h(G)$ and greater or equal than the index of the $i-1$-th β-bottleneck level $L^h(G)$(if any). An example of h-schedule and β-block partition is presented in Figure 2.3. According to this definition the first β-block is the only one for whom the first level may not be a β-bottleneck level. All the other β-blocks, if any, thus obtained are composed by a single β-bottleneck level and by at most $\lfloor \frac{n}{\beta} \rfloor \leq \frac{n}{\beta}$ non-β-bottleneck levels. The β-block partition of L^h induces a partition of $I \cup V$ in subsets composed by all the vertices associated to levels in the same β-block. Said partition is again unique since it descends form the unique L^h schedule of G. $L^{h^{(top)}}(G)$ is in fact a *greedy schedule* for $G\left(V, E\right)$ whose number of levels corresponds to the *depth* of $G(V, E)$. Additionally, the number of levels of the $h^{(sing)}$-schedule will be always less or equal than the number of levels in any other h-schedule. The number of levels of the $h^{(top)}$-schedule will vice-versa be higher or equal to the number of levels in any other h-schedule.

2.3.5 Proof method

The final goal of the study in this section is to show that is possible to construct h-visits which admit bounded maximum boundary size. We describe now a blueprint of the overall proof structure which we will use to achieve this goal, while later we get into the details of which results can be achieved for specific h-rules. The notions of h-schedule and β-block partition which we just introduced are crucial for our method.

Theorem 2.13 (Theorem blueprint). *Given an n-vertex CDAG $G(I \cup V, E)$, a real value $1 \leq \beta \leq n$, and a visit rule $h \in H_G$, $\exists \psi \in \Psi_h(G)$ s.t. $B_\psi^h \leq f(n, \beta) + 2\beta$, where f is a function of n and β which takes values in $\mathbb{R}$.*

Proof sketch We begin by building a the h-schedule $L^h(G)$, and we obtain the corresponding β-block partition. In order to verify that the statement of the theorem holds it will be sufficient to show that the following two lemmas are verified:

Lemma 2.14 (Block lemma for h). *Let us consider the β-block partition of the h-schedule of a given n-vertex CDAG $G(I \cup V, E)$, with $1 \leq \beta \leq n$. For any block in the β-block partition, let B denote the subset of vertices of $I \cup V$ which are entirely* *contained in the block. Let $G'(I' \cup V', E')$ be a sub-CDAG of G such that $E' = \{(u,v) \in E | u, v \in B\}$, $I' = \{v \in B | \forall u \in B, \nexists (u,v) \in E'\}$ and $V' = B \setminus I'$. Then $\exists \psi' \in \Psi_h(G')$ s.t. $B^h_{\psi'} \leq f(n,\beta)$, where f is a function of n and β which takes values in $\mathbb{R}$.*

Lemma 2.15 (Connection lemma)**.** *Given the n-vertex CDAG G, consider the β-block partition of $L^h(G)$. Suppose without loss of generality that the schedule has k blocks. Let $\psi^{(i)}$ be* sub-visits*, legal with respect to h, for the sub-CDAGs $G^{(i)}\left(V^{(i)}, E^{(i)}\right)$ induced each by one of k blocks of the β-block partition of L^h. Then the visit $\psi = \psi^{(1)}\psi^{(2)}\dots\psi^{(k)}$ is an h-visit of G.*

Furthermore, for any prefix-suffix partition of $\psi = \psi_1 \dots \psi_j | \psi_{j+1} \dots \psi_n$ such that $\psi_j \in \psi^{(i)}$, the following conditions hold:

- *at most β of vertices which do not belong to $\psi^{(i)}$ participate in the boundary $B^h_\psi(j)$;*
- *at most β vertices corresponding to the first level of the i-th block will appear in the boundary $B^h_\psi(j)$;*

Let $\psi^{(1)}, \psi^{(2)}, \dots, \psi^{(k)}$ be *sub-visits* for the sub-DAGs $G^{(i)}\left(V^{(i)}, E^{(i)}\right)$ of G induced by the β-block partition of $G(V,E)$ for which the *Block lemma* 2.14 holds. Since the *Connection lemma* 2.15 holds as well, the visit $\psi = \psi^{(1)}\psi^{(2)}\dots\psi^{(k)}$ is a visit on G and the for any prefix-suffix partition of $\psi = \psi_1 \dots \psi_i \psi_{i+1} \dots \psi_n$ we have boundary size $B^h_\psi(i) \leq f(n,\beta) + 2\beta$. The theorem follows. □

This blueprint allows us to divide the effort in proving the final results in two key lemmas. While the proof of the *Block lemma* will depend on the specific visit rule h being used, it is possible to provide a general proof for the *Connection lemma* which holds for any h-rule and any value $\beta \in \mathbb{R}^+$.

Proof of the Connection Lemma 2.15 In order to verify that ψ is an actually a h-visit on G we have to ensure that all vertices in $I \cup V$ occur exactly one time in ψ and that for every $1 \leq j \leq n$, with $\psi = \psi_1 \dots \psi_j \psi_{j+1} \dots \psi_n$, $\exists q \in h(\psi_j)$ s.t. $q \subseteq \{\psi_1, \dots, \psi_j\}$. Since the β-block partition induces a partition of the vertices of $I \cup V$ and ψ is obtained by cascading visits on the sub-CDAGs induced by the blocks partition all vertices in $I \cup V$ occur exactly one time in ψ.

Let $G^{(i)}\left(I^{(i)} \cup V^{(i)}, E^{(i)}\right)$ be the sub-CDAG induced by the i-th block of the β-block partition of G and let $\psi^{(i)}$ be a visit for $G^{(i)}$. Each vertex $v \in I^{(i)} \cup V^{(i)}$ is visited in $\psi^{(i)}$ iff in the previous steps all the vertices composing at least one of the enabling sets in $h(v)$ have already been visited either in the previous steps of $\psi^{(i)}$

or during the sub-visits of the previous blocks $\psi^{(1)} \dots \psi^{(i-1)}$ (if any). This property remains verified for the complete visit ψ, this allows us to conclude that ψ is actually a h-visit for G.

Let us now consider a vertex ψ_j in the global h-visit ψ with $\psi = \psi_1 \dots \psi_j \psi_{j+i} \dots \psi_n$ and let $\psi^{(i)}$ be the sub-visit for which $\psi_j \in \psi^{(i)}$. Consider the boundary set $B^h_\psi(j)$: since all the vertices in the blocks with index lower than i have already been visited, none of them will participate in the boundary; on the other hand, none of the vertices relative to blocks of index higher than $i+1$ will be enabled for a visit after $\psi^{(1)} \dots \psi^{(i)}$ since all the vertices relative to the block of index $i+1$ are yet to be visited. Therefore only vertices of the blocks of index i and $i+1$ can appear in $B^h_\psi(j)$. By construction of ψ no vertex of $\psi^{(j+1)}$ is visited until after $\psi_1 \dots \psi_i$. For the properties of the h-schedule we can conclude that only input vertices of the sub-DAG induced by the $i+1$-th block can appear in the $B^h_\psi(j)$. All the vertices corresponding to the initial level of the i-th block can appear in the boundary. Recall however that, according to our definition, the input vertices of G, which correspond to the first level of the first block, never appear in the boundary. According to the construction of the β-block partition, the first level of any other block is a β-bottleneck, and the number of vertices of said level which can appear in the boundary is upper bounded by β. By the construction of the β-block partition we can conclude that all these vertices are contained in the initial β-bottleneck level and there will be at most β such vertices. The lemma follows. □

In the following sections we will verify that the Block Lemma 2.14 is indeed verified for some relevant visit rules. We will then use these results to obtain an upper bound on the boundary space required by the relative visits.

2.3.6 Upper bound on the maximum boundary size of a $h^{(top)}$-visit

In this section we will prove that for any given n-vertex CDAG $G(I \cup V, E)$, $\exists \psi \in \Psi_{h^{(top)}}(G)$ such that $B^{h^{(top)}}_\psi \leq \sqrt{8d^+n}$, where d^+ denotes the maximum out-degree of G. In order to do so, we will first prove amore general result according to which for any given n-vertex CDAG G and for any real positive value $\beta \leq n$ there exists a schedule $\psi \in \Psi_{h^{(top)}}(G)$ such that $B^{h^{(top)}}_\psi \leq \alpha d^+ \frac{n}{\beta} + \beta$. To prove this result we will follow the steps described in the Blueprint Theorem 2.13.

We first introduce the following auxiliary lemma:

Lemma 2.16. *Given $i \geq 1$ consecutive levels of the $L^{h^{(top)}}$ schedule for an n-vertex CDAG G with maximum in-degree d^+, there exists a $h^{(top)}$-visit ψ' s.t. $B^{h^{(top)}}_{\psi'} \leq$*

$(d^{+}-1)(i-1)$.

Proof The proof is by induction on the number of levels i.

Base: Let us consider the base case $i = 1$, any permutation of the vertices of the initial (and only) level constitutes a visit of $G(I \cup V, E)$. As the only level is also the first level of the $h^{(top)}$-schedule it will contain just input values of G. As, by construction, the input vertices do not appear in the boundary, any such visit has maximum boundary size zero. This concludes the proof for the base case.

Inductive step: Let us assume that the inductive hypothesis is verified for $i \geq 1$, we will show that it still holds for $i + 1$. The visits begins by visiting any vertex $a_1 \in I$. If none of the direct successors of a are enabled for being visited. The visit proceeds by selecting another vertex of I. Suppose instead at least one of the successors of a_1 is enabled for being visited. a_1 has up to d^{+} distinct successors $b_1, b_2, \ldots, b_{d^{+}(a_1)}$ which may be enabled for being visited after visiting a_1. All such vertices will therefore appear in the boundary after a_1 has been visited. Without loss of generality, let us assume our visit proceeds to b_1. Let $R^{h}_{\psi_{pre}}(b_1)$ be the enabled reach of b_1 in G given the pre-visit $\psi_{pre} = a_1$.

Let G' be the sub-CDAG of G induced by the vertices in $R^{h}_{\psi_{pre}}(b_1)$. For the construction of the $h^{(top)}$-schedule of G, all the vertices of G' belong to the at most i levels of index higher than one of $L^{h^{(top)}}(G)$. Let us consider the $h^{(top)}$-schedule for G': $L^{h^{(top)}}(G')$ will have at most i levels, with the first one containing just b_1. By applying the inductive hypothesis we can find a $h^{(top)}$-sub-visit ψ_{b_1} for G' which admits maximum boundary size $B^{f^{((top)}}_{\psi_{b_1}} \leq (d^{+}-1)(i-1)$. Note that if $\left|R^{h}_{\psi_{pre}}(a_1)\right| = 0$ all the previous considerations are still verified with ψ_{b_1} being an empty visit.

We then re-evaluate which successors of a_1 are enabled for being visited given the pre-visit $a\psi_{b_1}$. We then select another vertex from this set and we repeat the operations previously described for b_1. At the i-th step, when evaluating the enabled reach of b_i we do so with respect to the pre-visit $a_1\psi_{b_1} \ldots \psi_{b_{i-1}}$ composed during the previous steps. These operations are repeated until there are no more successors of a enabled for being visited. This process leads to the visit $\psi_{a_1} = a_1\psi_{b_1}\psi_{b_2}\ldots$. As stated in Lemma 2.9, a visit of the enabled reach of a vertex will not enable vertices outside the enabled reach itself, as at most $d^{+}-1$ successors of a_1 can be in the boundary through ψ_{a_1} we can conclude that the maximum boundary size for ψ_{a_1} will be $B^{f^{(top)}}_{\psi_{a_1}} \leq (d^{+}-1) + (d^{+}-1)(i-1)$. We then select another vertex from I and we repeat the operations previously described for a_1 until all the vertices in I have been visited. At the j-th step, for $j > 1$, when evaluating the enabled reach of a_j we do so with respect to the pre-visit composed during the previous steps $\psi_{pre} = psi^{a_1}\psi^{a_2}\ldots\psi^{a_{j-1}}a_j$.

Let ψ be the complete visit obtained following this scheme. During each of the psi^{a_i} sub-visits all and only the vertices in the enabled reach of a_i are visited (with respect to appropriate pre-visit). As stated in Lemma 2.9, a visit of the enabled reach of a vertex will not enable vertices outside the enabled reach itself. We can therefore conclude $B_{\psi}^{h^{(top)}} \leq (d^+ - 1)i$. The lemma follows. □

This auxiliary lemma leads to an easy proof for the Block lemma for $h^{(top)}$:

Lemma 2.17 (Block lemma for $h^{(top)}$). *Let us consider the β-block partition of the $h^{(top)}$-schedule of a given n-vertex CDAG $G(I \cup V, E)$, with $1 \leq \beta \leq n$. For any block B in the β-block partition, let $G'(I' \cup V', E')$ be a sub-CDAG of G such that $E' = \{(u,v) \in E | u, v \in B\}$, $I' = \{v \in B | \forall u \in B, \nexists (u,v) \in E'\}$ and $V' = B \setminus I'$. Then $\exists \psi' \in \Psi_{h^{(top)}}(G')$ s.t. $B_{\psi'}^{h^{(top)}} \leq (d^+ - 1)\frac{n}{\beta}$.*

Proof Let G' be the CDAG corresponding to any block of a β-block partition of $L^{f^{(top)}}(G)$. As each block has just one β-bottleneck level, and at most n/β non-β-bottleneck levels the $L^{f^{(top)}}(G')$ has at most $n/\beta+1$ total levels. From Lemma 2.16 follows that there exists $\psi' \in \Psi_{h^{(top)}}(G')$ s.t. $B_{\psi'}^{h^{(top)}} \leq (d^+ - 1)\left(\frac{n}{\beta} + 1 - 1\right) \leq (d^+ - 1)\frac{n}{\beta}$.
□

Following the proof technique detailed in blueprint Theorem 2.13, it is possible to use the results given by the Block Lemma for $h^{(top)}$ (Lemma 2.17) and the Connection Lemma 2.15 to obtain the following result:

Theorem 2.18 (General upper bound to boundary size of $h^{(top)}$). *Given an n-vertex CDAG $G(I \cup V, E)$, a real value $1 \leq \beta \leq n$, and a visit rule $h \in H_G$, $\exists \psi \in \Psi_h(G)$ s.t. $B_{\psi}^{h} \leq (d^+ - 1)\frac{n}{\beta} + 2\beta$.*

Proof Let $L^{h^{(top)}}$ be the $h^{(top)}$-schedule for G (the *greedy schedule* for G). For a fixed β we can obtain a β-block partition of $L^{h^{(top)}}(G)$. Let $\psi^{(1)}, \psi^{(2)}, \ldots, \psi^{(k)}$ be *sub-visits* for the sub-DAGs $G^{(i)}\left(V^{(i)}, E^{(i)}\right)$ of G induced by the β-block partition of $G(V,E)$ for which the *Block lemma* 2.17 for $H^{(top)}$ holds. Since the *Connection lemma* 2.15 holds as well, the visit $\psi = \psi^{(1)}\psi^{(2)}\ldots\psi^{(k)}$ is an $h^{(top)}$-visit on G and the for any prefix-suffix partition of $\psi = \psi_1 \ldots \psi_i \psi_{i+1} \ldots \psi_n$ we have boundary size $B_{\psi}^{h^{(top)}} \leq (d^+ - 1)\frac{n}{\beta} + 2\beta$. The theorem follows. □

For $\beta = \sqrt{(d^+ - 1)n/2}$, we have $(d^+ - 1)\frac{n}{\beta} = 2\beta$ and the value of the upper bound of Theorem 2.18 is minimized.

Corollary 2.19. *Given an n-vertex CDAG $G(I \cup V, E)$, there exists a visit $\psi \in \Psi_{h^{(top)}}(G)$ s.t. $B_\psi^{h^{(top)}} \leq \sqrt{8(d^+ - 1)n}$. And therefore:*

$$\min_{\psi \in \Psi_h^{(top)}(G)} \max_{1 \leq i \leq |I \cup V|} \left| B_\psi^{h^{(top)}}(i) \right| \leq \sqrt{8(d^+ - 1)n}$$

Extension to topological-like visit rules

We say that a visit rule $h^{(top-l)}$ is *topological-like* if it associates to each vertex $v \in V$ a family constituted just by one subset of its predecessors:

$$h^{(top-l)}(v) = \begin{cases} \{q | q \subseteq pa(v)\} & \forall v \in V \\ \{\emptyset\} & \forall v \in I \end{cases}$$

Clearly, the $h^{(top)}$ visit rule satisfies this requirement.

We can extend the results previously presented for the $h^{(top)}$ visit rule to all topological-like visit rules:

Lemma 2.20. *Given an n-vertex CDAG $G(I \cup V, E)$, and any topological-like visit rule $h^{(top-l)}$, there exists a visit $\psi \in \Psi_{h^{(top-l)}}(G)$ s.t. $B_\psi^{h^{(top-l)}} \leq \sqrt{8(d^+ - 1)n}$.*

Proof Let $G'(I' \cup V', E')$ be the sub-CDAG of G for which:

- $E' = \{(u, v) \in E | u \in q \wedge q \in h^{(top-l)}(v)\}$;
- $I' = \{u \in I \cup V | \nexists (w, u) \in E'\}$;
- $V' = \{u \in V \setminus I'\}$.

We therefore have $|I' \cup V'| = n$. By Corollary 2.19 it is possible to find a visit $\psi \in \Psi^{h^{(top)}}(G')$ with maximum boundary size $\sqrt{8(d^+ - 1)n}$. As none of the edges in $E \setminus E'$ appear in any enabling set for any vertex of G, we have that $\psi \in \Psi^{h^{(top-l)}}(G)$ and at each step the boundary set for the visit ψ in G' will correspond to the boundary of the visit in G. $\square$

Impact on lower bound methods for space complexity using the topological visit rule

Recall that the maximum out-degree of the reverse CDAG G' corresponds to the maximum in-degree d^- of G. The result in Lemma 2.20 implies that using the topological-like visit rule in order to study the free-input space complexity of a given

CDAG with n vertices according to Theorem 2.12 leads to a lower bound, which is at most $\Omega\left(\sqrt{(d^- - 1)n}\right)$.

Furthermore, given the relationship between visits and markings described in Section 2.3.3 we have a corresponding result for the $f^{(top)}$ marking rule:

Corollary 2.21. *For any given n-vertex CDAG $G(I \cup V, E)$ there exists a marking $\phi \in \Phi_{f^{(top)}}(G)$ s.t. $B_\phi^{f^{(top)}} \leq \sqrt{8(d^- - 1)n}$. And therefore:*

$$\min_{\phi \in \Phi_{f^{(top)}}(G)} \max_{1 \leq i \leq |I \cup V|} \left| B_\phi^{f^{(top)}}(i) \right| \leq \sqrt{8d^- n}$$

Proof Given an n-vertex CDAG G, let us consider its inverse CDAG G_R. Recall that the maximum out-degree of the reverse CDAG G' corresponds to the maximum in-degree d^- of G. From Theorem 2.18 and from the relation between markings and visits in equation (2.4):

$$\min_{\phi \in \Phi_{f^{(top)}}(G)} \max_{1 \leq i \leq |I \cup V|} \left| B_\phi^{f^{(top)}}(i) \right| = \min_{\psi \in \Psi_h^{(top)}(G_R)} \max_{1 \leq i \leq |I \cup V|} \left| B_\psi^{h^{(top)}}(i) \right| \leq \sqrt{8(d^- - 1)n}$$

□

This result implies that using the topological marking rule in order to study the free-input space complexity of a given CDAG with n vertices according to Theorem 2.4 leads to a lower bound which is at most $\Omega\left(\sqrt{d^- n}\right)$. The same result holds for all *topological-like* marking rules, each of which corresponds to the appropriate topological-like visit rule previously defined.

2.3.7 Upper bound on the maximum boundary size of a $h^{(sing)}$-visit

In this section we will prove that given an n-vertex CDAG $G(I \cup V, E)$, $\exists \psi \in \Psi_{h^{(sing)}}(G)$ s.t. $B_\psi^{h^{(sing)}} \leq \alpha\sqrt{d^+ n}$ for some constant $\alpha \in \mathbb{R}^+$.

An important characteristic of the $h^{(sing)}$ visit rule is that, according to its definition, once a vertex has been visited *all* its successors are enable for being visited. This implies that for any vertex $v \in I \cup V$, its $h^{(sing)}$-enabled reach given *any* pre-visit ψ_{pre} will correspond to the entire reach of v (minus the vertices already visited in ψ_{pre}). Hence, if there exists a path connecting two vertices in G (i.e., one is a *descendant* of the other), then it will always be possible to visit such path with a $h^{(sing)}$-visit following the sequence of the vertices in the path. Because of this property of $h^{(sing)}$, in this section we use the lighter notation $R(v)$ to denote the $h^{(sing)}$-enabled reach of a vertex v.

In the following proof, we use the notion of *pivot vertex* of a CDAG. Given an n-vertex CDAG $G(I \cup V, E)$, we say that a vertex $v^* \in I \cup V$ is an *pivot vertex* for G if its reach is strictly greater than $n/2$ ($|R(v^*)| > n/2$), and all its immediate successors $u \in ch(v^*)$ have reach smaller or equal than $n/2$ (i.e., $R(u) \leq n'/2$).

Lemma 2.22 (Existence of a pivot)**.** *Let $G(I \cup V, E)$ be an n-vertex CDAG with $n \geq 3$ with $|I| = 1$. Then there is at least one pivot vertex in G.*

Proof Suppose G has no pivot vertices. Consider the unique input vertex: since it is not a pivot, it must have at least one successor whose reach is higher than $n/2$. As this vertex itself it is not a pivot, it must have at least one successor with reach higher than $n/2 - 1$. For any vertex v and any of its direct successors $u \in ch(v)$, from the properties of the $h^{(sing)}$-enabled reach we have that $R(u) \leq R(v) - 1$.

Suppose we repeat this scheme for $n/2$ steps; then al the successors of the vertex v^* considered at the $n/2$-th step must have reach smaller or equal than $n/2$. Furthermore, since v^* was chosen by the process at the $(n/2 - 1)$-th step, we have $R(v^*) > n/2$. v^* is therefore a pivot vertex for G. This leads to a contradiction. $\square$

We shall now prove that for any n-vertex CDAG there exists $h^{(sing)}$ which requires $\mathcal{O}\left(d^+ n\right)$ space. In the proof, we use a variation of the steps outlined in the Blueprint Theorem 2.13.

Theorem 2.23 (Upper bound for $h^{(sing)}$ visit)**.** *Given an n-vertex CDAG $G(I \cup V, E)$ there exists a visit $\psi \in \Psi_{h^{(sing)}}(G)$ s.t. $B_\psi^{h^{(sing)}} \leq \alpha\sqrt{d^+ n}$ for $\alpha = 3\sqrt{2}(\sqrt{2}+1)$.*

Proof The proof is by induction on n.

Base: For $n = 1$, it will possible to visit the unique vertex v of G with a visit $\psi = v$ s.t. $B_{\psi'}^{h^{(sing)}} = 1 \leq \alpha\sqrt{d^+ n}$.

Inductive step: Let us assume that the statement is verified for $n - 1 \geq 1$, we will show that it still holds for n. We start by building the $h^{(sing)}$-schedule for G, which coincides with the breadth-first schedule for G. We fix $\beta = \sqrt{d^+ n}$ and we obtain a $\sqrt{d^+ n}$-block partition. Each of the $\sqrt{d^+ n}$-blocks in the partition will be composed by at most $\sqrt{n/d^+} + 1$ levels. We will now consider each of the blocks of the $\sqrt{d^+ n}$-block partition separately. We will show that it is possible to visit the i-th sub-CDAGs of G each corresponding to the i-th block of the schedule with a visit $\psi^{(i)}$ which admits boundary size $\sqrt{d^+ n} + \alpha\sqrt{\frac{n}{2}}$. Let us consider the i-th block $B^{(i)}$ of the $\sqrt{d^+ n}$-block partition and let $G^{(i)}\left(I^{(i)} \cup V^{(i)}, E^{(i)}\right)$ be the corresponding sub-CDAG of G such that $E^{(i)} = \{(u,v) \in E | u, v \in B^{(i)}\}$, $I^{(i)} = \{v \in B | \forall u \in B^{(i)}, \nexists (u,v) \in E^{(i)}\}$ and $V^{(i)} = B^{(i)} \setminus I^{(i)}$.

As discussed in Section 2.3.2, the block partition ensures a shielding between the levels of each block. In the following we can consider independently the sub-CDAG corresponding each to a different block. From the properties of $h^{(sing)}$, we have that for any given vertex v in the block $B^{(i)}$, there will be a directed path from one of the vertices in the initial level of $B^{(i)}$ to v whose length (in terms of number of vertices of the path) will correspond to the index of the level on which v is collocated with respect to the block$B^{(i)}$. As a first thing we shall evaluate the reach of every vertex $v \in I^{(i)}$ with respect to $G^{(i)}$. For the properties of $h^{(sing)}$ the reach of a vertex is a superset of the enabled reach of that vertex given any possible pre-visit, including the empty pre-visit. There are two possible scenarios:

- **(for all $v \in I^{(i)}$ we have $|R(v) \cap V^{(i)}| \leq \frac{n}{2}$)** In this case we can construct a visit for $G^{(i)}$ as follows. We pick any vertex $a_1 \in I^{(i)}$ and we consider the sub-CDAG G^{a_1} corresponding to the reach of a_1 in $G^{(i)}$. We can apply the inductive hypothesis to G^{a_1} to obtain an $h^{(sing)}$ visit ψ^{a_1} for it with maximum boundary size $\alpha\sqrt{d^+ n/2}$. We then select a second vertex $a_2 \in I^{(i)}$ and we repeat the same operations with respect to the sub-CDAG corresponding to the vertices in the reach of a_2 in $G^{(i)}$ which have not already been visited (i.e., the vertices in its $h^{(sing)}$-enabled reach given the pre-visit ψ^{a_1}). Let $\psi^{(i)} = \psi^{a_1}\psi^{a_1}\ldots$ be the complete visit for $G^{(i)}$ obtained following this scheme. As verified in Lemma 2.9, a visit of the enabled reach of a vertex will not enable vertices outside the enabled reach itself. We can therefore conclude that the maximum boundary size for $\psi^{(i)}$ will be $\alpha\sqrt{d^+ n/2}$.

- **(there is a least one $v \in I^{(i)}$ such that $|R(v) \cap V^{(i)}| > \frac{n}{2}$)** In this case we can construct a visit for $G^{(i)}$ as follows. We pick any vertex $a_c \in I^{(i)}$ such that $|R(a_c) \cap V^{(i)}| > \frac{n}{2}$ and we consider the sub-CDAG G^{a_c} induced by all vertices in $R(a_c) \cap V^{(i)}$. By Lemma 2.22 there will be at least one pivot vertex v^* of G^{a_c} in one of the levels of the $h^{(sing)}$ schedule corresponding to the block $B^{(i)}$. Let us assume without loss of generality that v^* is placed in the k-th level of the block $B^(i)$, for $1 \leq k \leq \sqrt{\frac{n}{d^+}}$. For the construction of the schedule, there will be a path $\psi_{pre} = v_1 v_2 \ldots v_{k-i} \ldots v^*$ from a vertex $v_1 = a_1$ in the first level of the block, to v^* such that the vertices v_j in the path belong each to a different level of index for $1 < j < k$ in $B^{(i)}$. Note that a_1 can in general be different from a_c. For the properties of the $h^{(sing)}$ rule, it is possible to visit the path ψ_{pre} with the corresponding visit.

  Since each block of an $\sqrt{d^+ n}$-block partition of $L^{h^{(sing)}}$ has at most $\sqrt{n/d^+} + 1$ levels, any such path will have length $\sqrt{n/d^+} + 1$. Every vertex $v_j \in \psi_{pre}$, being

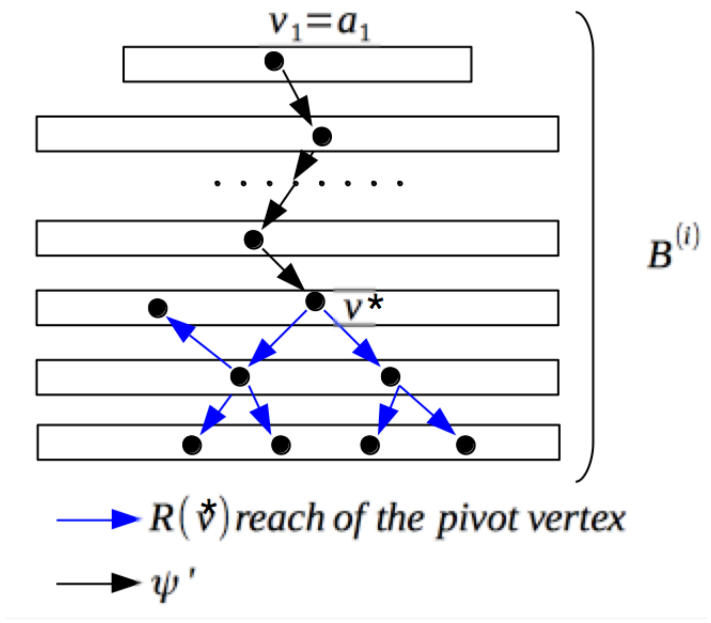


Figure 2.4: Construction of the visit using the pivot vertex method

visited in the path will enable all its at most d^+ successors (unless they have already been visited). The maximum boundary generated by ψ_{pre} is therefore $\sqrt{d^+ n}$ (the successors of v^* in the boundary are accounted separately). Let us consider a vertex $u_1 \in ch(v^*)$, let G_{u_1} be the sub-CDAG of $G^{(i)}$ induced by the vertices of $R^{h^{(sing)}}_{\psi'}(u_1)$ which are in $G^{(i)}$. From the definition of pivot vertex follows that $\left|R^{h^{(sing)}}_{\psi'}(u_1) \cap V^{(i)}\right| \leq n/2$. From the inductive hypothesis we have a visit ψ_{u_1} for G^{u_1} s.t. $B^{h^{(sing)}}_{\psi^{u_1}} \leq \alpha\sqrt{d^+ n/2}$. The same operations are repeated for all the u_i children of v^* in $G^{(i)}$, each time considering the $h^{(sing)}$-enabled reach of u_i limited to $G^{(i)}$ given the pre-visit $\psi_{pre_i} = \psi_{pre}\psi_{u_1}\psi_{u_2}\ldots\psi_{u_{i-1}}$. This process leads to the visit $\psi_{v^*} = \psi^{u_1}\psi^{u_2}\ldots$. Furthermore, since the sub-visit ψ^{u_j} will not enable any vertex in $G^{(i)}$ which is not in $R^{h^{(sing)}}_{\psi'}(u_j)$, we have $B^{h^{(sing)}}_{\psi'\psi^{v*}} \leq \sqrt{d^+ n} + \alpha\sqrt{d^+ n/2}$.

In order to complete the visit of $G^{(i)}$, we proceed by considering the vertices enabled by the pre-visit $\psi)pre$ and not visited in ψ_{v^*} which are themselves in $G^{(i)}$. We proceed starting from the successors of v_{k-1} up to the successors of v_1. By the definition of pivot all these vertices will now have a reach smaller than $n/2$. A visit ψ_b can therefore be constructed by simply reproducing the previously described step.

Finally, we consider the sub-CDAG induced by the vertices remaining in $G^{(i)}$ and not already visited in the previous steps. Since a there will be less than $n/2$ such vertices, we can apply again the inductive hypothesis and obtain a visit

ψ_r which admits maximum boundary $\alpha\sqrt{dn/2}$. The properties of the enabled reach discussed in Lemma 2.9 ensure that visit of the enabled reach of a vertex will not enable vertices outside the enabled reach itself. We can therefore conclude that the maximum boundary size for the visit $\psi^{(i)} = \psi_{pre}\psi_{v^*}\psi^b\psi^R$, obtained by composing the previously described sub-visits, will be $\sqrt{d^+n} + \alpha\sqrt{d^+n/2}$.

We have therefore obtained an $h^{(sing)}$-visit for the block $B^{(i)}$ whit boundary size $B^{h^{(sing)}}_{\psi^{(i)}} \leq \sqrt{d^+n} + \alpha\sqrt{d^+n/2}$.

We can apply the same steps for obtaining $h^{(sing)}$-visits for the sub-CDAGs of G each corresponding to one of the blocks of the $\sqrt{d^+n}$-block partition of the $h^{(sing)}$-schedule for G. All of these visits will have boundary size upper bounded by $\sqrt{d^+n}+\alpha\sqrt{d^+n/2}$.

For the Connection Lemma 2.15, the visit ϕ obtained by concatenating the sub-visits $\phi^{(1)}\phi^{(2)}\dots$ it is indeed an $h^{(sing)}$- visit for G and admits an upper bound for its maximum boundary size given by:

$$B^{h^{(sing)}}_{\psi^{(i)}} \leq \sqrt{d^+n} + \alpha\sqrt{d^+n/2} + 2\sqrt{d^+n} \leq \alpha\sqrt{d^+n}$$

for $\alpha \geq 3\sqrt{2}(\sqrt{2}+1)$. □

Extension to singleton-like visit rules

We say that a visit rule $h^{(sing-l)}$ is *singleton-like* if it associates to each vertex $v \in V$ a family of sets each containing some of its predecessors:

$$h^{(sing-l)}(v) = \begin{cases} \{\{u\}|u \in q \subseteq pa(v)\} & \forall v \in V \\ \{\emptyset\} & \forall v \in I \end{cases}$$

Clearly, the $h^{(sing)}$ visit rule satisfies this requirement. We can extend the results previously presented for the $h^{(sing)}$ visit rule to all singleton-like visit rules:

Lemma 2.24. *Given an n-vertex CDAG $G(I \cup V, E)$, and any singleton-like visit rule $h^{(sing-l)}$, there exists a visit $\psi \in \Psi_{h^{(sing-l)}}(G)$ s.t. $B^{h^{(sing-l)}}_{\psi} \leq 6(\sqrt{2}-1)\sqrt{d^+n}$.*

Proof Let $G'(I' \cup V', E')$ be the sub-CDAG of G for which:

- $E' = \{(u,v) \in E|\{u\} \in h^{(sing-l)}(v)\}$;
- $I' = \{u \in I \cup V|\nexists(w,u) \in E'\}$;

- $V' = \{u \in V \setminus I'\}$.

We therefore have $|I' \cup V'| = n$. By Theorem 2.23 it is possible to find a visit $\psi \in \Psi^{h^{(sing)}}(G')$ with maximum boundary size $6(\sqrt{2}-1)$. As none of the edges in $E \setminus E'$ appear in any enabling set for any vertex of G, we have that $\psi \in \Psi^{h^{(sing-l)}}(G)$ and at each step the boundary set for the visit ψ in G' will correspond to the boundary of the visit in G. The lemma follows. $\square$

Impact on lower bound methods for space complexity using the singleton visit rule

Recall that the maximum out-degree of the reverse CDAG G' corresponds to the maximum in-degree d^- of G. The result in Lemma 2.24 implies that using the topological-like visit rule in order to study the free-input space complexity of a given CDAG with n vertices according to Theorem 2.12 leads to a lower bound which is at most $\Omega\left(\sqrt{d^- n}\right)$.

Furthermore, given the relationship between visits and markings described in Section 2.3.3 we have a corresponding result for the $f^{(sing)}$ marking rule:

Corollary 2.25. *For any given n-vertex CDAG $G(I \cup V, E)$ there exists a marking $\phi \in \Phi_{f^{(sing)}}(G)$ s.t. $B_\phi^{f^{(sing)}} \leq 6(\sqrt{2}-1)\sqrt{d^- n}$. And therefore:*

$$\min_{\phi \in \Phi_{f^{(top)}}(G)} \max_{1 \leq i \leq |I \cup V|} \left| B_\phi^{f^{(top)}}(i) \right| \leq \sqrt{8 d^- n}$$

Proof Given an n-vertex CDAG G, let us consider its inverse CDAG G_R. Recall that the maximum out-degree of the reverse CDAG G' corresponds to the maximum in-degree d^- of G. From Theorem 2.23 and from the relationships between markings and visits in equation (2.4):

$$\min_{\phi \in \Phi_{f^{(top)}}(G)} \max_{1 \leq i \leq |I \cup V|} \left| B_\phi^{f^{(top)}}(i) \right| = \min_{\psi \in \Psi_h^{(top)}(G_R)} \max_{1 \leq i \leq |I \cup V|} \left| B_\psi^{h^{(top)}}(i) \right| \leq \sqrt{8 d^- n}$$

$\square$

This result implies that using the singleton marking rule in order to study the free-input space complexity of a given CDAG with n vertices according to Theorem 2.4 leads to a lower bound which is at most $\Omega\left(\sqrt{d^- n}\right)$. The same result holds for all *singleton-like* marking rules, each of which corresponds to the appropriate singleton-like visit rule previously defined.

2.4 Conclusion

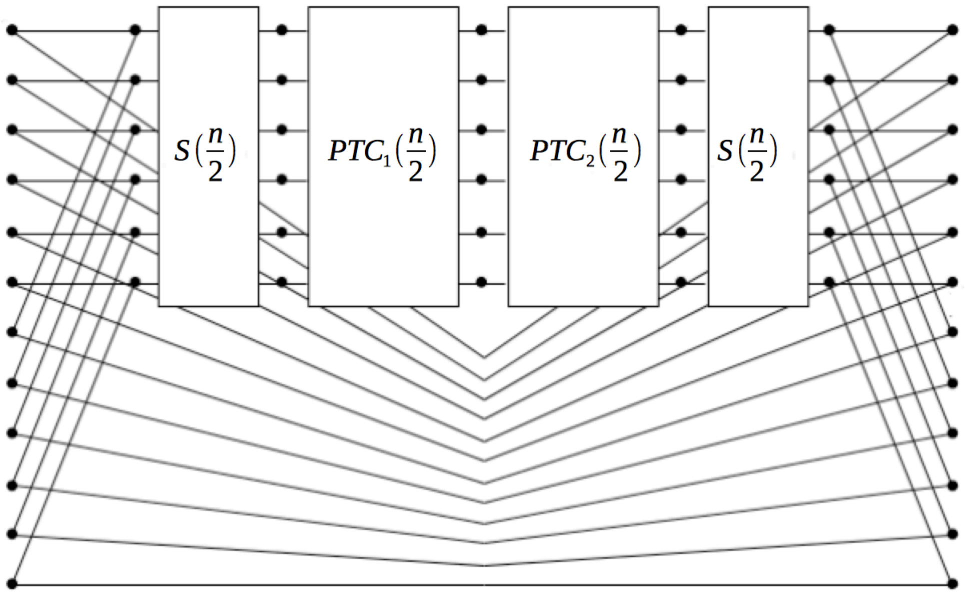


Figure 2.5: Paul-Tarjan-Celoni CDAG: in the figure we represent the recursive construction of the PTC(n) CDAG with n inputs. PTC(n) is constructed using two linear $n/2$-superconcentrators $S(n/2)$ CDAGs and two copies of PTC($n/2$) CDAGS. The base of the recursive construction is given for PTC(2^8) = $S(2^8)$ [48].

In this Chapter we studied the space complexity of computational directed acyclic graphs using the marking rule (Section 2.2) and the our novel visit rule approach (Section 2.3). We have furthermore shown that using the main visit (and therefore marking) rules can not lead to finding lower bounds for the space complexity of a CDAG higher than $\Omega\left(\sqrt{d^{-}n}\right)$. To the best of our knowledge the only CDAG proposed in literature that admits higher space complexity was introduced by Paul, Tarjan and Celoni in [48] and is here shown in Figure 2.5. In particular, said CDAG has free input space complexity $\Omega\left(n/\log n\right)$. The construction of the CDAG proposed in their paper based on a composition of superconcentrator CDAGS. This CDAG is particularly relevant as its space complexity matched the general upper bound on space complexity obtained by Hopcroft et al. in [35]. The study of this upper bound will be the main focus of the next chapter.

It is important to remark that our analysis of singleton and topological visit rules does not imply that the general visit approach as a whole does not suffice to provide asymptotically tight bound to the free-input space complexity of a generic CDAG. Rather, it suggests that for some families of CDAGS, such as the Paul-Tarjan-Celoni CDAG in [48], a more in depth analysis using some visit rule other than topological or singleton-like may be necessary.

Chapter 3

Upper bound to the space complexity of CDAG computations

While in the previous chapter we studied lower bounds on the space requirements for the computation of a straight line program represented by means of a CDAG, in this chapter we investigate an upper bound to the space complexity of any CDAG in the family $\mathcal{G}(m, d^-)$ of CDAGS with maximum in-degree d^- and m edges.

This question was originally studied by Hopcroft, Paul and Valiant in [35]. The authors showed that any CDAG with n vertices and maximum in-degree d^- can be pebbled using at most $d^- \frac{n}{\log n}$ pebbles[1]. An explicit construction of one such pebbling strategy is although not provided. In [48] provide give a recursive algorithm "*BEST-PEBBLE*" for pebbling any directed acyclic graph with n vertices using $c_1(d^- \log d^-)n/\log n$ pebbles ($c_1 > 0$ is a sufficiently large constant). An improved version of "*BEST-PEBBLE*", called "*FAST-PEBBLE*" was later introduced by Lengrauer and Tarjan in [40]. This algorithm allows to pebble any directed acyclic graph using $S = c_2(d^- \log d^-)n/\log n$ in time:

$$T \leq S(c_3 d^-)^{c_4^{(d^-+1)\frac{n}{S}}}$$

In [43] Micheal Loui, presented an alternative explicit construction of a pebbling strategy which allows to pebble any CDAG using at most $d^- \frac{n}{\log n}$ pebbles. His construction is based on the notion of *layered partition* of a CDAG. While in the paper the author presents a proof that a *layered partition* with the desired properties does indeed exists for any CDAG, no algorithm that actually constructs such layered partition is provided.

[1] Through this thesis, we use the notation "$\log x$" as short for "$\log_2 x$".

3.1 Our contribution

In this chapter we present an algorithm that, for any given CDAG $G(I \cup V, E) \in \mathcal{G}(m, d^-)$ constructs an explicit schedule for its computation which requires $\mathcal{O}(m \log m + d^-)$ space. Our algorithm requires as input just the CDAG G and *any* topological ordering of the vertices of G. It proceeds by obtaining a partition of the vertices of the CDAG which is then used in constructing a compete pebbling strategy which requires $\mathcal{O}\left(m \log m + d^-\right)$ pebbles.

3.2 Construction of the partition

For the given CDAG $G(I \cup V, E) \in \mathcal{G}(m, d^-)$, let $\phi = \phi_1, \ldots, \phi_n$, where $n = |I \cup V|$, be any topological ordering of the vertices of $I \cup V$. ϕ can be obtained with a simple exploration of G, such as a topological visit of G (described in Section 2.3.1) in $\mathcal{O}(n)$ time. Any topological ordering of the vertices is a permutation of the vertices in $I \cup V$ such that for any prefix-suffix partition of ϕ all the edges connecting vertices in the prefix to vertices in the suffix are exiting vertices in the prefix and entering vertices in the suffix (i.e., directed from the left to the right part of the permutation). Furthermore any such topological ordering corresponds to a complete computation of the CDAG for which every vertex is computed just once. In the following we denote as ϕ_i the i-th vertex in ϕ, while $\phi(v)$ denotes the position occupied by the vertex v in the permutation ϕ. Let us consider any possible sub-permutation $\phi_{i \to j} = \phi_i \phi_{i+1} \ldots \phi_{j-1} \phi_j$ of ϕ obtained by selecting all the elements with index greater or equal to i and smaller or equal to j. $\phi_{i \to j}$ is a topological ordering of all the vertices in $\phi_{i \to j}$.

Here we present the partitioning algorithm Part which divides the input CDAG G into a family of sub-CDGSs of G which is then used to construct a pebbling strategy for G which requires at most $(c+2)\frac{m}{\log m} + d^-$ pebbles, where $c \in \mathbb{R}^+$ is a constant value with $c > 13$. The algorithm requires as input the CDAG G which is to be partitioned, a topological ordering ϕ of its vertices, and a value $c \geq 13$. In the following, we use $B_\phi(i)$ to denote the number of vertices in the prefix $\phi_1 \ldots \phi_i$ which are connected to vertices in the suffix $\phi_{i+1} \ldots \phi_n$.

The algorithm proceeds evaluating whether for all prefix-suffix partitions of the topological ordering ϕ, the number of vertices of the prefix which have successors in the suffix is smaller or equal to $c\frac{m}{\log m}$. If that is the case, then the pebbling strategy whose steps follow the order of ϕ and for which, at each step only the vertices which have a successor in the suffix remain pebbled, is a complete pebbling for G which uses at most $c\frac{m}{\log m} + d^-$ pebbles. The algorithm returns the entire set of vertices of

Algorithm 1 Partitioning Algorithm

```
    Input: $G(I \cup V, E)$, $\phi$, $c$
 1: procedure PART($G(I \cup V, E), \phi, c$)
 2:     $m \leftarrow |E|$
 3:     $n \leftarrow |I \cup V|$
 4:     $i \leftarrow 1$
 5:     while $i \leq m$ and $B_\phi(i) \leq c\frac{m}{\log m}$ do
 6:         $i \leftarrow i + 1$
 7:     if $i \geq m$ then
 8:         return $(G, \phi)$
 9:     else
10:         $V^{(p)} \leftarrow \phi_{1\to i}$
11:         $E^{(p)} \leftarrow \{(u,v) \in E | u, v \in \phi_{1\to i}\}$
12:         $V^{(s)} \leftarrow \phi_{i+1\to n}$
13:         $E^{(s)} \leftarrow \{(u,v) \in E | u, v \in \phi_{i+1\to n}\}$
14:         PART($G^{(p)}(V^{(p)}, E^{(p)}), \phi_{1\to i}, c$); PART($G^{(s)}(V^{(s)}, E^{(s)}), \phi_{i+1\to n}, c$)
```

G and the topological ordering ϕ without splitting it further. If instead there is at least one prefix-suffix partition of ϕ for which $B_\phi(i) > c\frac{m}{\log m}$ then the computation corresponding to ϕ requires more than $c\frac{m}{\log m}$ memory space. The initial CDAG G is then divided in two parts by splitting the vertices of G in two sets: $V^{(p)}$ corresponding to the vertices in the prefix $\phi_1 \ldots \phi_i$, and $V^{(s)}$ corresponding to the vertices in the suffix $\phi_{i+1} \ldots \phi_n$. We then select the subset of edges $E_{pref} \subseteq E$ (resp., $E_{suff} \subseteq E$) whose endpoints are bot vertices in the prefix $\phi_1 \ldots \phi_i$ (resp., the suffix $\phi_{i+1} \ldots \phi_n$). The partitioning algorithm is then invoked recursively for the two generated sub-CDAGs $G^{(p)}(V^{(p)}, E^{(p)})$ and $G^{(s)}(V^{(s)}, E^{(s)})$ using $\phi_1 \ldots \phi_i$ (resp., $\phi_{i+1} \ldots \phi_n$) as a topological ordering of the vertices in $V^{(p)}$ (resp., $V^{(s)}$).

The vertices of the starting CDAG are partitioned into the subsets $V^{(p)}$ and $V^{(s)}$ such that $V = V^{(p)} \cup V^{(s)}$ and $V^{(p)} \cap V^{(s)} = \emptyset$. In the division of the set of edges E in the sub-sets $E^{(p)}$ and $E^{(s)}$, the at least $c\frac{m}{\log m} + 1$ edges which are exiting from vertices in $B_\phi(i)$ in the prefix and entering vertices in the suffix are *removed* such that $|E^{(p)}| + |E^{(s)}| \leq |E| - \lceil c\frac{m}{\log m} \rceil$. For a fixed constant c, PART splits a CDAG with m edges in two parts only if $m > 2^c$. If $m \leq 2^c$ we would in fact have $c\frac{m}{\log m} \geq m$, and the condition for the split would therefore not be verified.

Les $S = \left(G^{(1)}(V^{(1)}, E^{(1)}), \phi^{(1)}\right); \left(G^{(2)}(V^{(2)}, E^{(2)}), \phi^{(2)}\right); \ldots; \left(G^{(j)}(V^{(j)}, E^{(j)}), \phi^{(j)}\right)$ be sequence of sub-CDAGs of G obtained as output of PART. The following properties hold:

Lemma 3.1. *The family of subsets $\{V^{(1)}, \ldots, V^{(j)}\}$ is topological partition of G.*

Proof When PART splits the set of vertices of a CDAG into two sub-sets it does so by partitioning realizing a prefix-suffix partition of a topological ordering of the CDAG itself. By definition (1.4) each such partition is a topological partition. The lemma follows. □

Lemma 3.2. *Let $\ell(m)$ be the maximum number of sub-CDAGs of any given m-edge CDAG G returned as output by* PART(G, ϕ, c). *If $c \geq 7$, we have:*

$$\ell(m) = 1, \qquad for\ m \leq 2^c \tag{3.1}$$

$$\ell(m) \leq \frac{m}{(\log m)^2}, \qquad for\ m > 2^c \tag{3.2}$$

Proof The proof is by induction on the value of m.

Base: the base case is for $m \leq 2^c$. In this case we have $c \geq \log m$ and therefore, according to the functioning of PART, we have that no split can occur and the entire CDAG is returned as output.

Inductive step: in our inductive hypothesis we assume that the statement is verified for $m-1$, we shall now verify that it holds for $m > 2^c$ as well. Note that the function $x/(\log x)^2$ is monotonically increasing for values of $x \geq e^2$. Since we are considering the case for which $m > 2^c$, for any $c \geq 7$ we have $m > 128 > e^2$. The function $x/(\log x)^2$ is therefore monotonically increasing with $m > 2^c$

For $m > 2^c$, PART may split the initial CDAG in two sub-CDAGs $G^{(p)}(V^{(p)}, E^{(p)})$ and $G^{(s)}(V^{(s)}, E^{(s)})$ while removing at least $\lceil cm/\log m \rceil$ edges.

$$|E^{(p)}| + |E^{(s)}| \leq m - \lceil c\frac{m}{\log m} \rceil$$

As in this analysis we aim for an upper bound to the number of sub-CDAGs generated by PART, in the following we assume without loss of generality that the minimum possible number of edges $\lceil cm/\log m \rceil$ are removed by PART and therefore that $|E^{(p)}| + |E^{(s)}| = m - \lceil c\frac{m}{\log m} \rceil$. All following considerations are verified as well if more edges are removed. The split is in general not balanced:

$$|E^{(p)}| = m - \lceil c\frac{m}{\log m} \rceil - \alpha$$
$$|E^{(s)}| = \alpha$$

for $\alpha \in \{1, 2, \ldots, m - \lceil c\frac{m}{\log m} \rceil\}$. PART is then invoked for the two sub-CDAGs $G^{(p)}$ and $G^{(s)}$ separately. We thus have:

$$\ell(m) \leq \max_{\alpha \in \{0,1,\ldots,m-\lceil c\frac{m}{\log m} \rceil\}} \ell\left(\left(m - \lceil c\frac{m}{\log m} \rceil\right) - \alpha\right) + \ell(\alpha) \tag{3.3}$$

Let us first consider the case for $2^c < m \leq 2^{c+1}$. If the PART splits G into two sub-CDAGs $G^{(p)}$ and $G^{(s)}$ we have:

$$E^{(p)}| + |E^{(s)}| \leq m - \lceil c\frac{m}{\log m}\rceil \leq m\left(1 - \frac{c}{\log m}\right) \leq m\left(1 - \frac{c}{c+1}\right) \leq \frac{m}{c+1}$$

For $c \geq 7$ this implies:

$$E^{(p)}| + |E^{(s)}| \leq 2^c$$

This fact ensures that Part will not split any further either $G^{(p)}$ nor $G^{(s)}$. By inductive hypothesis we have:

$$\ell(m) = 2$$

For $c \geq 7$ we have:

$$\frac{m}{(\log m)^2} \geq \frac{2^c}{(c+1)^2} \geq \frac{128}{64} = \ell(m) = 2$$

This conclude the analysis for $2^c < m \leq 2^{c+1}$.

We now consider the case for $m > 2^{c+1}$. In our study of equation (3.3) we first consider the cases for which the split is unbalanced with $\alpha \in \{0, 1, \ldots, 2^c, m - \lceil c\frac{m}{\log m}\rceil - 2^c, \ldots, m - \lceil c\frac{m}{\log m}\rceil - 1, m - \lceil c\frac{m}{\log m}\rceil\}$. Because of the relation between the terms in equation (3.3), the case for $\alpha = i$ is equivalent to the case for $\alpha = m - \lceil c\frac{m}{\log m}\rceil - i$, for $i \in \{0, 1, \ldots, 2^c\}$. For any of these cases, we have that one of the sub-CDAGs generated by Part from G has size lower or equal than 2^c and is therefore guaranteed not to be split any further. From (3.3) we thus have:

$$\ell(m) \leq \max_{\alpha \in \{0,1,\ldots,2^c\}} \ell\left(\left(m - \lceil c\frac{m}{\log m}\rceil\right) - \alpha\right) + 1$$

If $m - \lceil c\frac{m}{\log m}\rceil - \alpha \leq 2^c$ then both sub-CDAGs are not divided by the recursive invocations of Part. We therefore have $\ell(m) = 2$ and the result seen for $2^c < m \leq 2^{c+1}$ holds for any $c \geq 7$.

If instead, $m - \lceil c\frac{m}{\log m}\rceil - \alpha > 2^c$, by inductive hypothesis we have:

$$\ell(m) \leq \frac{m - \lceil c\frac{m}{\log m}\rceil}{\left(\log\left(m - \lceil c\frac{m}{\log m}\rceil\right)\right)^2} + 1$$

Since the function $x/(\log x)^2$ is monotonically increasing for $x \geq 2^c$ we have:

$$\frac{m - \lceil c\frac{m}{\log m}\rceil - \alpha}{\left(\log\left(m - \lceil c\frac{m}{\log m}\rceil - \alpha\right)\right)^2} \leq \frac{m - c\frac{m}{\log m}}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2}$$

To verify that condition (3.2) is verified, it is thus sufficient to show that:

$$\frac{m}{(\log m)^2} \geq \frac{m - c\frac{m}{\log m}}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2} + 1$$

holds for any value of $m > 2^{c+1}$. Through some algebraic manipulations we have:

$$\frac{m}{(\log m)^2} \geq \frac{m - c\frac{m}{\log m}}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2} + 1$$

$$\frac{m}{(\log m)^2} \geq \frac{m}{(\log m)^2}\frac{1 - \frac{c}{\log m}}{\left(1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}\right)^2} + 1$$

$$\frac{m}{(\log m)^2}\left(1 - \frac{1 - \frac{c}{\log m}}{\left(1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}\right)^2}\right) \geq 1$$

$$\frac{m}{(\log m)^2}\left(1 - \frac{(\log m)^2}{(\log m)^2}\frac{1 - \frac{c}{\log m}}{\left(1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}\right)^2}\right) \geq 1$$

$$\frac{m}{(\log m)^2}\left(1 - \frac{(\log m)^2 - c\log m}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2}\right) \geq 1$$

$$\frac{m}{(\log m)^2}\left(\frac{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2 - (\log m)^2 + c\log m}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2}\right) \geq 1$$

$$\frac{m}{(\log m)^2}\left(\frac{\left(\log\left(1 - \frac{c}{\log m}\right)\right)^2 + 2\log m\log\left(1 - \frac{c}{\log m}\right) + c\log m}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2}\right) \geq 1$$

$$\frac{m}{(\log m)^2}\left(\frac{\left(\log\left(1 - \frac{c}{\log m}\right)\right)^2 + \log m\left(c + 2\log\left(1 - \frac{c}{\log m}\right)\right)}{\left(\log\left(m - c\frac{m}{\log m}\right)\right)^2}\right) \geq 1$$

Since $\left(\log\left(1-\frac{c}{\log m}\right)\right)^2 \geq 0$ and $m \geq 2^{c+1}$ we have:

$$\frac{m}{(\log m)^2}\left(\frac{\left(\log\left(1-\frac{c}{\log m}\right)\right)^2 + \log m\left(c+2\log\left(1-\frac{c}{\log m}\right)\right)}{\left(\log\left(m-c\frac{m}{\log m}\right)\right)^2}\right) \geq \frac{m}{(\log m)^2}\left(\frac{\log m\,(c-2\log(c+1))}{(\log m)^2}\right)$$

$$\frac{m}{(\log m)^2}\left(\frac{\log m\,(c-2\log(c+1))}{(\log m)^2}\right) = \frac{m}{(\log m)^3}\,(c-2\log(c+1))$$

For $m > 2^{c+1}$ with $c \geq 7$ we have $\frac{m}{(\log m)^3}\,(c-2\log(c+1)) > 1$ and the condition (3.2) is therefore verified.

Finally, we consider the case for $m > 2^{c+1}$, for which the split is executed according to $\alpha \in \{2^c, 2^c+1, \ldots, m-\lceil c\frac{m}{\log m}\rceil - 2^c - 1, m - \lceil c\frac{m}{\log m}\rceil - 2^c\}$.

In order to simplify the analysis of (3.2) we analyze the function:

$$f(\alpha) = \frac{m - c\frac{m}{\log m} - \alpha}{\left(\log\left(m-c\frac{m}{\log m}-\alpha\right)\right)^2} + \frac{\alpha}{(\log\alpha)^2}$$

for values of $\alpha \in [2^c, m-\lceil c\frac{m}{\log m}\rceil - 2^c]$. Note that since $x/(\log x)^2$ is monotonically increasing for $x \geq 2^c$, and that $\{2^c, 2^c+1, \ldots, m-\lceil c\frac{m}{\log m}\rceil - 2^c - 1, m-\lceil c\frac{m}{\log m}\rceil - 2^c\} \subseteq [2^c, m-\lceil c\frac{m}{\log m}\rceil] - 2^c$, we have:

$$\begin{aligned}\max_{\alpha\in[2^c, m-\lceil c\frac{m}{\log m}\rceil - 2^c]} f(\alpha) \geq \max_{\alpha\in\{2^c,\ldots,m-\lceil c\frac{m}{\log m}\rceil\}-2^c} & \frac{m-\lceil c\frac{m}{\log m}\rceil - \alpha}{\left(\log\left(m-\lceil c\frac{m}{\log m}\rceil-\alpha\right)\right)^2} + \frac{\alpha}{(\log\alpha)^2}\\ & \ell(m-\lceil c\frac{m}{\log m}\rceil - \alpha) + \ell(\alpha)\\ & \ell(m)\end{aligned} \tag{3.4}$$

In order to find the value of α which maximizes the value of $f(\alpha)$ we shall evaluate its derivative in α. I

$$\frac{d}{d\alpha}f(\alpha) = (\ln(2))^2\left(\frac{-\ln\left(m-c\frac{m}{\log m}-\alpha\right)+2}{\left(\ln\left(m-c\frac{m}{\log m}-\alpha\right)\right)^3} + \frac{\ln(\alpha)-2}{(\ln\alpha)^3}\right)$$

The derivative has value zero for $\alpha = \left(m - c\frac{m}{\log m}\right) - \alpha = \frac{1}{2}\left(m-c\frac{m}{\log m}\right)$, by studying the sign of the derivative we can verify that $f(a)$ is indeed maximized for $\alpha = \frac{1}{2}\left(m - c\frac{m}{\log m}\right)$. Note that the boundary values 2^c and $m - \lceil c\frac{m}{\log m}\rceil - 2^c$ could also maximize $f(\alpha)$ even though the derivative is not zero. We therefore have that $\forall \alpha \in [2^c, m - \lceil c\frac{m}{\log m}\rceil] - 2^c]$:

$$f(\alpha) \leq \max\{2\frac{\frac{m-c\frac{m}{\log m}}{2}}{\left(\log\left(\frac{m-c\frac{m}{\log m}}{2}\right)\right)^2}, \frac{\left(\left(m - c\frac{m}{\log m}\right) - 2^c\right)}{\left(\log\left(\left(m - c\frac{m}{\log m}\right) - 2^c\right)\right)^2} + \frac{2^c}{(\log 2^c)^2}\}$$

Since the limit values 2^c and $m - \lceil c\frac{m}{\log m}\rceil - 2^c$ have already been studied for the case $\alpha \in \{0, 1, \ldots, 2^c, m - \lceil c\frac{m}{\log m}\rceil - 2^c, \ldots, m - \lceil c\frac{m}{\log m}\rceil - 1, m - \lceil c\frac{m}{\log m}\rceil\}$, we shall focus on the case $\alpha\frac{1}{2}\left(m - c\frac{m}{\log m}\right)$.

Through some algebraic manipulations we have:

$$2\frac{\frac{m-c\frac{m}{\log m}}{2}}{\left(\log\left(\frac{m-c\frac{m}{\log m}}{2}\right)\right)^2} = \frac{m\left(1 - c\frac{1}{\log m}\right)}{\left(\log m + \log\left(1 - c\frac{1}{\log m}\right) - 1\right)^2} = \frac{m}{(\log m)^2}\frac{\left(1 - c\frac{1}{\log m}\right)}{\left(1 + \frac{\log\left(1-c\frac{1}{\log m}\right)-1}{\log m}\right)^2}$$

To verify that condition (3.2) is verified, it is thus sufficient to show that:

$$\frac{m}{(\log m)^2} \geq \frac{m}{(\log m)^2}\frac{\left(1 - c\frac{1}{\log m}\right)}{\left(1 + \frac{\log\left(1-c\frac{1}{\log m}\right)-1}{\log m}\right)^2}$$

Which in turn holds if:

$$1 - c\frac{1}{\log m} \leq \left(1 + \frac{\log\left(1 - c\frac{1}{\log m}\right) - 1}{\log m}\right)^2$$
$$(\log m)^2\left(1 - c\frac{1}{\log m}\right) \leq \left(\log m + \log\left(1 - c\frac{1}{\log m}\right) - 1\right)^2$$
$$\log m\left(2 - 2\log\left(1 - c\frac{1}{\log m}\right) - c\right) \leq \log\left(1 - c\frac{1}{\log m}\right)\left(\log\left(1 - c\frac{1}{\log m}\right) - 2\right) + 1 \tag{3.5}$$

As $\log\left(1 - c\frac{1}{\log m}\right)\left(\log\left(1 - c\frac{1}{\log m}\right) - 2\right) \geq 0$, we have that (3.5) is verified if:

$$2 - 2\log\left(1 - c\frac{1}{\log m}\right) - c \leq 1$$
$$1 - 2\log(c + 1) - c \leq 0$$

which holds for $c \geq 7$. The lemma follows. □

3.3 Construction of the computation

Let $(G^{(1)}, \phi^{(1)}); \ldots; (G^{(j)}, \phi^{(j)})$ be the family of sub-CDAGs of G, each paired with a topological ordering of their respective vertices, obtained as output of the partitioning algorithm PART when a topological ordering of its vertices ϕ and a constant value $c \geq 7$ are provided as input. We shall now show how to use these subsets in order to construct a complete pebbling strategy $\mathcal{C}$ for G. We shall then show that $\mathcal{C}$ requires at most $\mathcal{O}\left(\frac{m}{\log m}\right) + d$ pebbles where $|E| = m$.

3.3.1 The pebbling subroutine

We describe $\mathcal{C}$ in terms of its corresponding pebbling strategy (see Section 2.1). The main building block of the computation is the pebbling procedure PEBB whose pseudocode is reported in Algorithm 2.

Algorithm 2 Pebbling subroutine

```
    Input: a target vertex $v \in V^{(k)}$,$G^{(k)}$, $\phi^{(k)}$ the topological ordering of the vertices in $V^{(j)}$.
 1: procedure PEBB($v, G^{(k)}, \phi^{(k)}$)
 2:     $i \leftarrow 1$
 3:     $n \leftarrow \phi(v)$
 4:     while $i \leq n$ do
 5:         if $pa(v) \nsubseteq V^{(k)}$ then
 6:             for all $u \in pa(v) \setminus V^{(k)}$ do
 7:                 Let $V^{(u)}$ bet the subset s.t. $u \in V^{(u)}$
 8:                 PEBB($u, G^{(u)}, \phi^{(u)}$)
 9:         Pebble $\phi_i$
10:         Remove any pebble placed on vertices in $pa(v) \setminus V^{(k)}$
11:         Remove any pebble placed on vertices of $\phi_{1\rightarrow i}$ which do not have a successor in the suffix $\phi_{i+1\rightarrow n}$
12:     Remove all pebbles placed by the execution of PEBB on vertices in $V^{(k)}$, except for $v$
```

The PEBB procedure receives as input values a *target vertex* to be pebbled v, the sub-CDAG $G^{(k)}$ obtained using the partitioning algorithm PART for which $v \in V^{(k)}$ with $1 \leq k \leq j$, and the corresponding topological ordering $\phi^{(j)}$ for $G^{(k)}$. PEBB proceeds by a series of pebbling steps which are compliant to the rules of the pebbling game until the target vertex v is pebbled. The pebbling strategy followed by PEBB proceeds by pebbling vertices following the steps of $\phi^{(k)}$. In particular, once the steps corresponding to a prefix of $\phi^{(k)}$ have been executed, the only vertices carrying a pebble are those which have at least successor in the corresponding suffix. Proceeding according to this scheme ensures that whenever a vertex in $V^{(k)}$ is about to be pebbled, all its predecessors which are in $V^{(k)}$ are carrying pebbles as well. Let

$v^*V^{(k)}$ be a the vertex which is to be pebbled next according to $\phi^{(k)}$, if at least one of its predecessor is not in $V^{(k)}$ (and therefore, not pebbled) we proceed as follows:

1. we *interrupt*, the execution of $\phi^{(k)}$, none of the pebbles residing on vertices of $V^{(k)}$ are removed;
2. we consider one such predecessor $u \in pa(v^*s)$: we invoke the subroutine PEBB using u as the target vertex. Notice that since the partition of G obtained using PART is topological, u will belong to a subset $V^{(l)}$ with $l < j$. When this invocation of PEBB terminates, the vertex u is pebbled;
3. the steps in 2. are repeated until all the predecessors of v^* are carrying a pebble:
4. v^* is pebbled;
5. pebbles are removed from all predecessors of v^* not in $V^{(k)}$;
6. we reprise the execution of $\phi^{(k)}$.

We say that an execution of PEBB is *active* if it has not terminated yet. We say that two PEBB executions are *concurrently active* if both of them are active and if one of the two has been triggered by the other either directly, or as a result of a chain of PEBB invocations. This definition can be straightforwardly extended to multiple concurrent executions of PEBB.

Lemma 3.3. *Let $G^{(1)}, \ldots, G^{(j)}$ be the family of sub-CDAGs of G obtained as output of the partitioning algorithm* PART *when a topological ordering of its vertices ϕ and a constant value $c \geq 7$ are provided as input. At any time during the execution of the* PEBB *there may be at most $j \leq \ell(m)$ concurrently active executions of* PEBB*, where m is the number of edges in G.*

Proof As pointed out in 2. an execution of PEBB which has as a target a vertex in the k-th sub-CDAG of G, for $1 \leq k \leq j$, may trigger another execution of PEBB whose target vertex will belong to the l-th sub-CDAG of G with $1 \leq l < k$. From Lemma 3.2 we have that for any CDAG with at most m edges, the number of sub-CDAGs returned as output by PART is upper bounded by $l(m)$. The lemma follows. □

3.3.2 Challenging vertices

Recall that according to the (R2) rule of the pebble game described in Section 2.1, in order to put a pebble on a vertex of V it is necessary for all its predecessors to be carrying a pebble as well. This implies that a CDAG with maximum in-degree d^- requires at least d^- pebbles in order to be computed. We say that a vertex $v \in V$ is *challenging* if and only if its in-degree is higher than $\log m$. The set of challenging vertices $C(G) \subseteq V$ is defined as such:

$$C(G) = \{v \in V | d^-(v) \geq \log m\}$$

Clearly as the total number of edges in G is $|E| = m$, we have $|C(G)| \leq m/\log m$. Suppose the vertices in $C(G)$ to be ordered according to their position in ϕ, we denote as $C(G)_i$ the i-th vertex in $C(G)$ according to such ordering. Note that such ordering correspond to a topological ordering of the challenging vertices of G. Once the value corresponding to a challenging vertex is computed it is *never* removed from memory until the very end of the computation. We therefore *reserve* a certain number of pebbles (i.e., memory space) for challenging vertices, in order for each of them to be always available in memory for the following steps of the computation once they have been evaluated.

3.3.3 Composition of $\mathcal{C}$

In our strategy for the construction of the computation $\mathcal{C}$ of $G(I \cup V, E)$ we initially pebble each of the challenging vertices of G using the pebbling subroutine Pebb in conjunction with the family of sub-CDAGs $G^{(1)};\ldots;G^{(j)}$ of G obtained using the partitioning algorithm Part. In particular, we pebble the vertices in $C(G)$ one at a time proceeding according to their order. Once a challenging vertex is pebbled, said vertex remains pebbled until the end of the computation. After all the challenging vertices have been pebbled we proceed by pebbling the remaining vertices of the CDAG which may have not been pebbled yet. We start by pebbling the vertices in $V^{(j)}$ using the pebbling subroutine and we then proceed backwards to vertices in $V^{(j-1)}$ until all vertices in $V^{(1)}$ have been pebbled. A sketch for the construction of $\mathcal{C}$ is provided in Algorithm 3.

3.4 Analysis of the space requirements of $\mathcal{C}$

While the computation $\mathcal{C}$ obtained using the construction in Algorithm 3 is generally not optimal in terms of the number of pebbling steps, we will show that it allows to

Algorithm 3 Construction of $\mathcal{C}$

```
    Input: The CDAG $G(I \cup V)$ to be pebbled, $\phi$ a topological ordering of the vertices of $G$, a constant $c \geq 7$.
 1: $(G^{(1)}, \phi^i); \dots ; (G^{(j)}, \phi^i) \leftarrow$ PART$(G(I \cup V, E), \phi, c)$
 2: $k \leftarrow$ number of sub-CDAGs generated by PART
 3: $C(G) \leftarrow$ challenging vertices of $G$ ordered according to their position in $\phi$
 4: $i \leftarrow 1$
 5: while $i \leq |C(G)|$ do                        ▷ challenging vertices are pebbled one at a time
 6:     $v^* \leftarrow$ the $i$-th vertex of $C(G)$
 7:     $G^{(*)}(V^{(*)}, E^{(*)}) \leftarrow$ the sub-CDAG for which $v^* \in V^{(*)}$
 8:     PEBB$(v^*, G^{(*)}, \phi^{(*)})$
 9:     $i \leftarrow i + 1$
10: while $k > 0$ do                                ▷ pebbling of the remaining vertices of $G$
11:     $v \leftarrow$ the last vertex of $\phi^{(k)}$
12:     PEBB$(v, G^{(k)}, \phi^{(k)})$
13:     $k \leftarrow k - 1$
```

pebble the entire CDAG $G(I \cup V, E)$ using $\mathcal{O}\left(\frac{m}{\log m} + d^-\right)$ pebbles.

Let $(G^{(1)}, \phi^{(1)}); \dots ; (G^{(j)}, \phi^{(j)})$ be the family of sub-CDAGs of G (paired with the respective topological ordering of their vertices) obtained as output of the partitioning algorithm PART when a topological ordering of its vertices ϕ and a constant value $c \geq 7$ are provided as input.

Lemma 3.4. *For any $1 \leq k \leq j$, let $\mathcal{C}^{(k)}$ be the pebbling strategy which follows the steps in $\phi^{(k)}$ and for which, at each step, the only vertices carrying a pebble are those which have at least a successor which has yet to be pebbled. $\mathcal{C}^{(k)}$ is a valid pebbling strategy for the CDAG $G^{(k)}(V^{(k)}, E^{(k)})$ and requires at most $\min\{c\frac{|E^{(k)}|}{\log |E^{(k)}|}, |E^{(k)}|\}$ pebbles.*

Proof From the properties of PART, we have that for any prefix-suffix partition of $\phi^{(k)}$ the number of vertices in the prefix which have at least a successor in the suffix is at most $\min\{c\frac{|E^{(k)}|}{\log |E^{(k)}|}, |E^{(k)}|\}$. As stated in Section 2.1, each topological ordering of the vertices of a CDAG correspond to a complete nr-pebbling strategy for the CDAG itself. Once the pebbling steps corresponding to any prefix of $\phi^{(k)}$ have been executed, any pebble placed on a vertex of the prefix which has no successor in the prefix is useless (as no successor is yet to be pebbled) and can be therefore safely removed. Each vertex of $G^{(k)}$ is thus pebble once, and it remains pebbled until all its successors have been pebbled as well. □

Lemma 3.5. *For $c \geq 7$ we have:*

$$\min\{m, c\frac{m}{\log m}\} \geq \sum_{k=1}^{j} \min\{c\frac{|E^{(k)}|}{\log |E^{(k)}|}, |E^{(k)}|\} \tag{3.6}$$

for some value $c \geq 2$, where $m = |E|$.

Proof Let us analyze the partitioning algorithm PART. We can think of the criterion used in dividing the input CDAG as if PART was managing a *budget* of $\min\{\lfloor c\frac{m}{\log m}\rfloor, m\}$ pebbles. PART evaluates whether its budget is sufficient to pebble the CDAG $G(I \cup V, E)$ using the schedule which follows the steps in ϕ and for which, at each step, only the vertices for whom not all the successors have been pebbled remain pebbled. If this is the case PART stops returning the entire CDAG G and the lemma is trivially verified. If that is not the case then PART divides the CDAGs in two sub-CDAGs $G^{(p)}(V^{(p)}, E^{(p)})$ and $G^{(s)}(V^{(s)}, E^{(s)})$ according to the rules discussed in Section 3.2, with $|E^{(p)}| + |E^{(s)}| \leq m - \lceil c\frac{m}{\log m}\rceil$. To each of these sub-CDAGs it is then assigned a budget depending on the respective number of edges: respectively $\min\{\lfloor c|E^{(p)}|/\log|E^{(p)}|\rfloor, |E^{(p)}|\}$ and $\min\{\lfloor c|E^{(s)}|/\log|E^{(s)}|\rfloor, |E^{(s)}|\}$. In order to verify that the lemma holds, we need to verify that for any CDAG G with m edges the original budget allocated is greater or equal to the cumulative budget allocated for $G^{(p)}$ and $G^{(s)}$:

$$\min\{m, c\frac{m}{\log m}\} \geq \min\{|E^{(p)}|, \lfloor c\frac{|E^{(p)}|}{\log|E^{(p)}|}\rfloor\} + \min\{|E^{(s)}|, \lfloor c\frac{|E^{(s)}|}{\log|E^{(s)}|}\rfloor\} \tag{3.7}$$

The proof is by induction on the value of m:

Base: the base case is for $m \leq 2^c$. In this case the budget $\min\{c\frac{m}{\log m}, m\} = m$ is sufficient for pebbling the original CDAG. PART returns the entire CDAG G which can clearly be pebbled with m pebbles.

Inductive step: in our inductive hypothesis we assume that the statement is verified for $m-1$, we shall now verify that it holds for $m > 2^c$ as well. If the budget $\min\{m, \lfloor c\frac{m}{\log m}\rfloor\}$, is deemed sufficient for the pebbling of G the lemma is verified as PART returns only the entire G. If this is not the case, PART splits the initial CDAG in two sub-CDAGs $G^{(p)}(V^{(p)}, E^{(p)})$ and $G^{(s)}(V^{(s)}, E^{(s)})$ while removing at least $\lceil cm/\log m\rceil$ edges.

$$|E^{(p)}| + |E^{(s)}| \leq m - \lceil c\frac{m}{\log m}\rceil$$

Since the cumulative budget is a nondecreasing function of $|E^{(p)}| + |E^{(s)}|$, in the following, we assume without loss of generality that the minimum possible number of edges $\lceil cm/\log m\rceil$ are removed by PART and therefore that $|E^{(p)}| + |E^{(s)}| = m - \lceil c\frac{m}{\log m}\rceil$. All, considerations are verified as well if more edges are removed in the split. Said split in general is not balanced:

$$\begin{aligned}
|E^{(p)}| &= m - \lceil c\frac{m}{\log m}\rceil - \alpha \\
|E^{(s)}| &= \alpha
\end{aligned}$$

for $\alpha \in \{1, 2, \ldots, m - \lceil c\frac{m}{\log m}\rceil\}$. PART is then invoked for the two sub-CDAGs $G^{(p)}$ and $G^{(s)}$ separately. By hypothesis, the cumulative budget assigned to the sub-CDAGs generated by PART is therefore given by:

$$\min\{m - \lceil c\frac{m}{\log m}\rceil - \alpha, c\frac{m - \lceil c\frac{m}{\log m}\rceil - \alpha}{\log\left(m - \lceil c\frac{m}{\log m}\rceil - \alpha\right)}\} + \min\{\alpha, c\frac{\alpha}{\log \alpha}\}. \tag{3.8}$$

We shall consider different cases according to the value of m:

- For $2^c < m \leq 2^{2c}$, from the inductive hypothesis we have that the cumulative budget assigned to the sub-CDAGs generated by PART is upper bounded by:

$$\min\{|E^{(p)}|, c\frac{|E^{(p)}|}{\log |E^{(p)}|}\} + \min\{|E^{(s)}|, c\frac{|E^{(s)}|}{\log |E^{(s)}|}\} \leq |E^{(p)}| + |E^{(s)}| < m - c\frac{m}{\log m}$$

  We therefore have that the condition in equation (3.6) is surely verified if the cumulative budget assigned to the sub-CDAGs generated by PART is less or equal than the budget originally assigned for the entire CDAG $c\frac{m}{\log m}$:

$$\begin{aligned}
c\frac{m}{\log m} &\geq m - c\frac{m}{\log m} \\
2c\frac{m}{\log m} &\geq m
\end{aligned}$$

  Since for $2^c < m \leq 2^{2c}$ we have $\log m \leq 2c$, the condition is clearly verified.

- Let us now assume $m > 2^{2c}$, we first consider the cases for which the split is unbalanced with $\alpha \in \{0, 1, 2, m - \lceil c\frac{m}{\log m}\rceil - 2, m - \lceil c\frac{m}{\log m}\rceil - 1, m - \lceil c\frac{m}{\log m}\rceil\}$. Because of the relation between $|E^{(p)}|$ and $|E^{(s)}|$, the case for $\alpha = i$ is equivalent to the case for $\alpha = m - \lceil c\frac{m}{\log m}\rceil - i$, for $i \in \{0, 1, 2\}$. For any of these cases we have that the cumulative budget assigned to $G^{(p)}$ and $G^{(s)}$ is upper bounded by:

$$\min\{m - \lceil c\frac{m}{\log m}\rceil - \alpha, c\frac{m - \lceil c\frac{m}{\log m}\rceil - \alpha}{\log\left(m - \lceil c\frac{m}{\log m}\rceil - \alpha\right)}\} + \alpha \leq c\frac{m - c\frac{m}{\log m}}{\log\left(m - c\frac{m}{\log m}\right)} + 2$$

To verify that condition (3.7) is verified, it is thus sufficient to show that:

$$
\begin{aligned}
c\frac{m}{\log m} &\geq c\frac{m - c\frac{m}{\log m}}{\log\left(m - c\frac{m}{\log m}\right)} + 2 \\
c\frac{m}{\log m} &\geq c\frac{m}{\log m}\frac{1 - \frac{c}{\log m}}{1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}} + 2 \qquad (3.9)
\end{aligned}
$$

Through some algebraic manipulations, we have:

$$
\begin{aligned}
c\frac{m}{\log m}\left(1 - \frac{1 - \frac{c}{\log m}}{1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}}\right) &\geq 2 \\
c\frac{m}{\log m}\left(1 - \frac{\log m}{\log m}\frac{1 - \frac{c}{\log m}}{1 + \frac{\log\left(1 - \frac{c}{\log m}\right)}{\log m}}\right) &\geq 2 \\
c\frac{m}{\log m}\left(1 - \frac{\log m - c}{\log m + \log\left(1 - \frac{c}{\log m}\right)}\right) &\geq 2 \\
c\frac{m}{\log m}\left(\frac{c + \log\left(1 - \frac{c}{\log m}\right)}{\log m + \log\left(1 - \frac{c}{\log m}\right)}\right) &\geq 2
\end{aligned}
$$

Since $m > 2^{2c}$ we have:

$$
c\frac{m}{\log m}\left(\frac{c-1}{\log m}\right) \geq c\frac{m}{\log m}\left(\frac{c + \log\left(1 - \frac{c}{\log m}\right)}{\log m + \log\left(1 - \frac{c}{\log m}\right)}\right)
$$

For any $c \geq 7$ we have $c\frac{m}{\log m}\left(\frac{c-1}{\log m}\right) \geq 2$ and the constraint in (3.9) is therefore verified.

Finally, we consider the case for $m > 2^{2c}$, for which the split is executed according to $\alpha \in \{2, 3, \ldots, m - \lceil c\frac{m}{\log m}\rceil - 2 - 1, m - \lceil c\frac{m}{\log m}\rceil - 2\}$. In order to simplify the analysis we study the value of (3.8) we analyze the function:

$$
f(\alpha) = \frac{m - c\frac{m}{\log m} - \alpha}{\log\left(m - c\frac{m}{\log m} - \alpha\right)} + \frac{\alpha}{\log \alpha}
$$

for values of $\alpha \in [2, m - \lceil c\frac{m}{\log m}\rceil - 2]$. Note that since $x/\log x$ is monotonically increasing for $x \geq 2$, and that $\{2, \ldots, m - \lceil c\frac{m}{\log m}\rceil - 2\} \subseteq [2, m - \lceil c\frac{m}{\log m}\rceil] - 2,$

we have:

$$\max_{\alpha\in[2,m-\lceil c\frac{m}{\log m}\rceil-2]} f(\alpha) \geq \max_{\alpha\in\{2,\dots,m-\lceil c\frac{m}{\log m}\rceil\}-2} \frac{m-\lceil c\frac{m}{\log m}\rceil-\alpha}{\left(\log\left(m-\lceil c\frac{m}{\log m}\rceil-\alpha\right)\right)^2}+\frac{\alpha}{(\log\alpha)^2} \tag{3.10}$$

In order to find the value of α which maximizes the value of $f(\alpha)$ we shall evaluate its derivative in α. I

$$\frac{d}{d\alpha}f(\alpha)=\ln(2)\left(\frac{-\ln\left(m-c\frac{m}{\log m}-\alpha\right)+1}{\left(\ln\left(m-c\frac{m}{\log m}-\alpha\right)\right)^2}+\frac{\ln(\alpha)-1}{(\ln\alpha)^2}\right)$$

The derivative has value zero for $\alpha=\left(m-c\frac{m}{\log m}\right)-\alpha=\frac{1}{2}\left(m-c\frac{m}{\log m}\right)$, by studying the sign of the derivative we can verify that $f(a)$ is indeed maximized for $\alpha=\frac{1}{2}\left(m-c\frac{m}{\log m}\right)$. Note that the boundary values 2 and $m-\lceil c\frac{m}{\log m}\rceil-2$ could also maximize $f(\alpha)$ even though the derivative is not zero. We therefore have that $\forall\alpha\in[2,m-\lceil c\frac{m}{\log m}\rceil]-2]$:

$$f(\alpha)\leq\max\{2\frac{\frac{m-c\frac{m}{\log m}}{2}}{\left(\log\left(\frac{m-c\frac{m}{\log m}}{2}\right)\right)^2},\frac{\left(\left(m-c\frac{m}{\log m}\right)-2^c\right)}{\left(\log\left(\left(m-c\frac{m}{\log m}\right)-2^c\right)\right)^2}+\frac{2^c}{(\log 2^c)^2}\}$$

Since the boundary values $\alpha=2$ and $\alpha=m-\lceil c\frac{m}{\log m}\rceil-2$ have already been studied for the case $\alpha\in\{0,1,2,m-\lceil c\frac{m}{\log m}\rceil-2,,m-\lceil c\frac{m}{\log m}\rceil-1,m-\lceil c\frac{m}{\log m}\rceil\}$, we shall focus on the case $\alpha\frac{1}{2}\left(m-c\frac{m}{\log m}\right)$. Through some algebraic manipulations, we have:

$$2\frac{\frac{m-c\frac{m}{\log m}}{2}}{\log\left(\frac{m-c\frac{m}{\log m}}{2}\right)}=\frac{m\left(1-c\frac{1}{\log m}\right)}{\log m+\log\left(1-c\frac{1}{\log m}\right)-1}=\frac{m}{\log m}\frac{\left(1-c\frac{1}{\log m}\right)}{\left(1+\frac{\log\left(1-c\frac{1}{\log m}\right)-1}{\log m}\right)}$$

To verify that condition (3.8) is verified, it is thus sufficient to show that:

$$\frac{m}{\log m}\geq\frac{m}{\log m}\frac{\left(1-c\frac{1}{\log m}\right)}{\left(1+\frac{\log\left(1-c\frac{1}{\log m}\right)-1}{\log m}\right)}$$

Through some algebraic manipulations, we have:

$$\frac{m}{\log m} \geq \frac{m - c\frac{m}{\log m}}{\log\left(m - c\frac{m}{\log m}\right) - 1}$$

$$\frac{m}{\log m} \geq \frac{m\left(1 - c\frac{1}{\log m}\right)}{\log\left(m\left(1 - c\frac{1}{\log m}\right)\right) - 1}$$

$$\frac{m}{\log m} \geq \frac{m}{\log m}\frac{1 - c\frac{1}{\log m}}{1 + \frac{\log\left(1-c\frac{1}{\log m}\right)}{\log m} - \frac{1}{\log m}}$$

$$1 \geq \frac{1 - c\frac{1}{\log m}}{1 + \frac{\log\left(1-c\frac{1}{\log m}\right)}{\log m} - \frac{1}{\log m}}$$

Which is verified if:

$$1 + \frac{\log\left(1 - c\frac{1}{\log m}\right)}{\log m} - \frac{1}{\log m} \geq 1 - c\frac{1}{\log m}$$

$$\log\left(1 - c\frac{1}{\log m}\right) \geq 1 - c$$

Since $\log m > 2c$ we have:

$$\log\left(1 - c\frac{1}{\log m}\right) \geq \log\left(\frac{1}{2}\right) \geq 1 - c$$

which is verified for any $c \geq 7$.

$\square$

Using these two lemmas we can provide the proof for the main result of this chapter.

Theorem 3.6 (General upper bound to pebbling cost for CDAGs). *For any given CDAG $G(I \cup V, E) \in \mathcal{G}(m, d^-)$, there exists a complete pebble strategy for G which requires at most $\min\{n, (c+1)\frac{m}{\log m} + d^-\}$ for $c \geq 7$, where $|I \cup V| = n$.*

Proof If $n \leq c\frac{m}{\log m} + d^-$, then any pebbling strategy that follows the steps corresponding to a topological ordering of the vertices is a complete pebbling for G and requires at most n pebbles.

For $n \geq c\frac{m}{\log m} + d^-$, let us consider the pebbling strategy $\mathcal{C}$ whose construction has been described in Section 3.3.3. $\mathcal{C}$ is a complete pebbling of G. As d^- is the maximum in-degree of G, this implies that there is at least a vertex in G with d^- predecessor. It is therefore necessary to use at least d^- pebbles in $\mathcal{C}$. According to the construction of $\mathcal{C}$ in Section 3.3.3, the challenging vertex of G are pebbled one at a time according to their topological ordering with respect to G. Furthermore, once a challenging vertex is pebbled, it remains pebbled until the end of $\mathcal{C}$. This fact ensures that at each time during the execution of $\mathcal{C}$ there it will never therefore be necessary to concurrently accumulate the predecessors of two challenging vertices with in-degrees higher than $\log m$. Clearly, d^- pebbles will be necessary to accumulate the input of any challenging vertex.

Consider now the family of sub-CDAGs of G $(G^{(1)}, \phi^{(1)}); \ldots; (G^{(j)}, \phi^{(j)})$ obtained as output of the execution of $\textsc{Part}(G, \phi, c)$. From Lemma 3.4 we have that each of the $G^{(*)}$ sub-CDAGs can be pebbled following the steps in $\mathcal{C}^{(*)}$ using at most $\min\{c|E^{(*)}|/\log|E^{(*)}|, |E^{(*)}|\}$. $\mathcal{C}$ is obtained by combining the $C^{(*)}$ as described in Section 3.3.3.

As discussed in Section3.3.1, at any time during the execution of one of the $\textsc{Pebb}$ subroutine there can be at most j concurrently active executions of $\textsc{Pebb}$. Each of these concurrent executions will operate on a different sub-CDAG $G^{(*)}$ according to the steps in $\mathcal{C}^{(*)}$ (as described in detail in the description of the pebbling subroutine $\textsc{Pebb}$).The maximum number of pebble being concurrently used by the sub-computations $\mathcal{C}^{(*)}$ will therefore be at most $\sum_{j=1}^{i} \min\{c|E^{(j)}|/\log|E^{(j)}|, |E^{(j)}|\}$. From Lemma 3.5, for $c \geq 7$ we have that

$$c\frac{m}{\log m} \geq \sum_{k=1}^{j} \min\{|E^{(k)}|, c\frac{|E^{(k)}|}{\log|E^{(k)}|}\}$$

According to the construction of $\mathcal{C}$, for each of the at most j concurrently active invocations of $\textsc{Pebb}$ it is necessary to accumulate pebbles on all the predecessors of each of the *triggering* vertices that have triggered one of the concurrently active executions of $\textsc{Pebb}$. As previously discussed, our strategy for the challenging vertices ensures that at most one of these triggering vertices is challenging while all the others will have at most $\log m$ predecessors. From Lemma 3.2, for $c \geq 7$ we have $j \leq \frac{m}{(\log m)^2}$. The maximum number of pebbles necessary for pebbling the predecessors of the triggering vertices will therefore be upper bounded by:

$$\log m \frac{m}{(\log m)^2} + d^- = \frac{m}{\log m} + d^-$$

By combining these observations, we can therefore conclude that the maximum number of pebbles required by $\mathcal{C}$ is upper bounded by:

$$c\frac{m}{\log m} + \frac{m}{\log m} + d^- = (c+1)\frac{m}{\log m} + d^-.$$

The Theorem follows. $\square$

3.5 Conclusion

In this chapter we described an explicit construction of a pebbling strategy which allows to pebble any CDAG $G(I \cup V, E) \in \mathcal{G}(m, d^-)$ using $\min\{n, \mathcal{O}(m/\log m + d^-)\}$ pebbles, where $n = |I \cup V|$ and $m = |E|$. In order to construct said strategy, only a topological ordering of the vertices of G is required. This result implies an upper bound on the pebbling cost of any CDAG in the family $\mathcal{G}(m, d^-)$. In our result, the maximum in-degree d^- of the CDAG appears as an additive term to the main $m/\log m$ component of the upper bound rather than a multiplicative term as in results previously presented in literature [35, 43]. As $n \leq m \leq d^-|I \cup V| = d^-n$, if for a given CDAG we have $m = \Theta(dn)$ our bound corresponds to the $d^-\frac{n}{\log n}$ upper bound in [43, 35]. Note however that said bounds quickly go to $\mathcal{O}(n)$ even if a single node has in-degree greater or equal to $\log n$, therefore losing significance. Our bound, may still retain significance for some CDAGs for which $m = o(d^-n)$, even if a limited number of edges has high (i.e., greater than $\log n$) in-degree. For instance, for any CDAGs for which $m = \Theta n$ and which constant number of vertices with high degree our approach still allows to obtain a significant bound.

Chapter 4

On the I/O complexity of Strassen's matrix multiplication algorithm

When considering the complexity of algorithms two kinds of costs are therefore to be considered: the *arithmetic* cost which depends on the number of required computational steps, and the *communication* cost which depends on the required movement of data within the execution of an algorithm, either between levels of a memory hierarchy (in the sequential case), or over a network connecting processors (in the parallel case).

We can generally intend communication as the movement of data within the execution of an algorithm, either between levels of a memory hierarchy (in the sequential case), or over a network connecting processors (in the parallel case). In both of this applicative scenarios, the communication component of an algorithm often costs significantly more time than its arithmetic. Furthermore, while Moore's Law predicts an exponential speedup of hardware in general, the annual improvement rate of time per-arithmetic-operation has, over the years, consistently exceeded that of time-per-word read/write. The fraction of running time spent on communication is thus expected to increase further becoming more and more of a bottleneck for the performance of both multi-level memory and parallel computing architectures. It is therefore of interest to investigate the tradeoff between the memory space being used and the data communication needed for the algorithm execution (the *input-output (I/O) complexity*), and to design and implement algorithms which optimize the use of memory in order to minimize communication and attaining these lower bounds on the other hand. In particular, given the observation that the communication cost rather than the arithmetic cost constitutes the true bottleneck in the performance of algorithms, the question on whether computing more than one time intermediate values can allow to achieve a reduction of the minimum number of required I/O operations is of great interest.

Given its critical importance, this field of study has been largely investigated in literature. In the following sections, we provide a rigorous definition for our communication model which is based on the original work by Hong and Kung [37]. We then focus on the squared matrix multiplication problem and in particular on Strassen's matrix multiplication algorithm, the analysis of which constitutes the main part of this chapter. We begin by studying the I/O complexity of Strassen's algorithm when executed sequentially on a machine equipped with a two level memory hierarchy. We provide an alternative technique to those in [4] and [62] to obtain a tight lower bound to the I/O complexity of Strassen's matrix multiplication algorithm for computations in which no intermediate result is ever recomputed. We then obtain the first asymptotically tight lower bound to the I/O complexity of Strassen's algorithm for general computations i.e., computations without any restriction on the recomputation of intermediate values. Our technique is based on an application of Gigoriev's *information flow* concept [33].

We also study the I/O complexity of Strassen's algorithm when executed in parallel by P processors each equipped with a finite memory. We obtain a novel tight lower bound which holds for any computation (no restriction on recomputation), without any assumption regarding the distribution of the input data among the P processors at the beginning of the computation.

4.1 The square matrix multiplication function

The problem of interest for our analysis in the following sections is the square matrix multiplication which is introduced in this section. In the first part we present a rigorous definition for the problem and we discuss properties of relevance for our analysis. We then provide a brief survey of the algorithms which have been proposed in literature for its computation.

4.1.1 Problem definition

A *matrix* of size $m \times n$ over a set R is a rectangular array of elements drawn from R consisting of m of rows and n of columns. Rows are indexed by integers from the set $\{1, 2, 3, \ldots, m\}$ and columns are indexed by integers from the set $\{1, 2, 3, \ldots, n\}$. Given a matrix A, we denote as $a_{i,j}$ the entry in the i-th row and j-th column of A. A *square matrix* is an $n \times n$ matrix for some integer n.

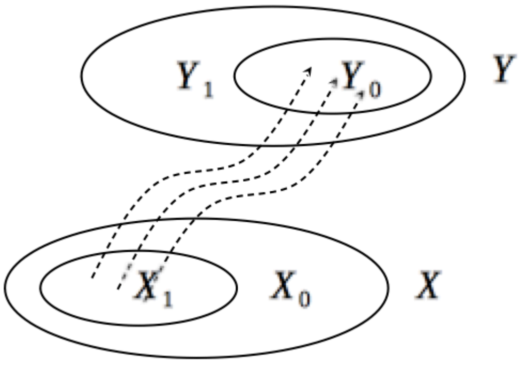


Figure 4.1: Representation of the information flow for a function $f(\cdot)$

In the following, we focus on the set of matrices whose entries are drawn from a ring $\mathcal{R}$.

Definition 4.1 (Matrix Multiplication Function over the ring $\mathcal{R}$)**.** *The matrix multiplication function $f^{(n)} : A \times B : \mathcal{R}^{(m+p)n} \to \mathcal{R}^{mp}$ multiplies an $m \times n$ matrix $A = [a_{i,j}]$ by an $n \times p$ matrix $B = [b_{i,j}]$ to produce the $m \times p$* product *matrix $C = f^{(n)}_{AB}(A, B) = A \times_n B = [c_{i,j}]$, where:*

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}. \tag{4.1}$$

The *square* matrix multiplication is therefore a special case of the general function for which the input *factor* matrices have size $n \times n$ for a given integer n and therefore $A, B, C \in \mathcal{R}^{n^2}$. In order to simplify the notation, in the following presentation we use $f_{n\times n}$ instead of $f^{(n)}_{A\times B}$ to denote the product of two square matrices $n \times n$.

4.1.2 Information flow property of matrix multiplication

We introduce here a characterization of the *information flow* of functions. We say that a function $f : \mathcal{A}^n \to \mathcal{A}^m$ has a large information flow from input variables in X_1 to output variables in Y_1 if there are values for input variables in $X_0 = X \setminus X_1$ such that many different values can be assumed by Y_1 as the values of the inputs in X_1 range over all the possible $\mathcal{A}^{|X_1|}$ values.

The concept of information flow of functions was originally introduced by Grigoriev [33] and was used in deriving timespace tradeoffs for the execution of straight-line programs represented by CDAGs. A revised version of the same concept was later presented by Savage [56] and used to derive lower bounds on area-time tradeoffs for VLSI algorithms. We use here the definition due to Savage as presented in [58].

Definition 4.2 (Information flow of a function). *A function $f : \mathcal{A}^n \to \mathcal{A}^m$ has a $w(u,v)$-flow if for all subsets X_1 and Y_1 of its n input and m output variables, with $|X_1| \geq u$ and $|Y_1| \geq v$, there is a sub-function h of f obtained by making some assignment to variables of f not in X_1 and discarding output variables not in Y_1 such that h has at least $|\mathcal{A}|^{w(u,v)}$ points in the image of its domain.*

A lower bound on the information flow for the square matrix multiplication was presented by Savage in [58].

Lemma 4.3 (Information flow of matrix multiplication over the ring $\mathcal{R}$ [58]). *The matrix multiplication function $f_{n\times n} : \mathcal{R}^{2n^2} \to \mathcal{R}^{n^2}$ over the ring $\mathcal{R}$ has a $w(u,v)$-flow, where:*

$$w_{n\times n}(u,v) \geq \frac{1}{2}\left(v - \frac{(2n^2-u)^2}{4n^2}\right) \tag{4.2}$$

Proof A complete proof can be found in [58](Theorem 10.5.1). □

It is important to remark that the information flow result just discussed is a property of the squared matrix multiplication function itself, *regardless* of the specific algorithm which is to be used in order to compute it.

As previously mentioned, Grigoriev's method (and its extension due to Tompa [65, 66]) based on the information flow properties of functions has been principally used to study area-time tradeoffs for VLSI algorithms [56] and time-space tradeoffs for various functions such as: matrix-vector product[65], polynomial multiplication[65], cyclic shift [59], integer multiplication [59], transitive closure [66].

In our work we will show an interesting new way of using the information flow property of functions in order to obtain lower bounds on their performance when run on a hierarchical memory machine.

4.1.3 Matrix multiplication algorithms

Various algorithms have been proposed in literature to compute the product of two matrices. In this section, we provide a brief survey of the main contributions in literature extracted from [70], before focusing on Strassen's fast matrix multiplication algorithm [63].

The *Naive* algorithm for computing the squared matrix multiplication product, computes each entry of the product matrix C through three nested loops requires $\mathcal{O}(n^3)$ arithmetic operations for input matrices of size $n \times n$.

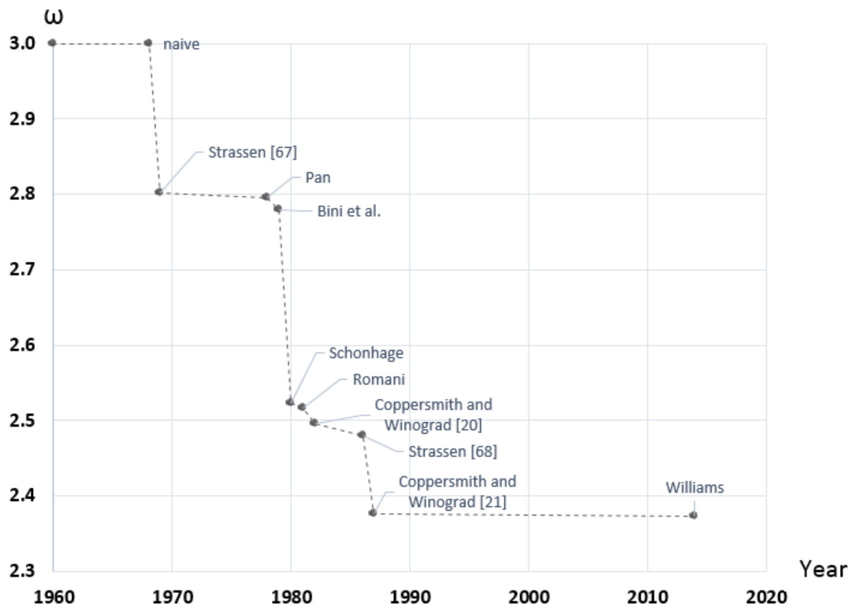


Figure 4.2: Evolution of the bound on ω

More efficient algorithms for "*fast matrix multiplication*" have however been proposed in literature starting from 1969, when Strassen [63] gave the first sub-cubic time algorithm for matrix multiplication, running in $\mathcal{O}\left(n^{2.808}\right)$ time. This discovery opened a long line of research which gradually reduced the matrix multiplication exponent ω over time. In 1978, Pan [46] proposed an algorithm with achieves $\omega < 2.796$ while, in the following year, Bini et al. [12] introduced an algorithm for approximate matrix multiplication based on the notion of *border rank* achieving $\omega < 2.78$. In [60] Schonhage combined his works with ideas from [46] and he showed that $\omega < 2.522$. This result was improved shortly afte by Romani [55], by achieving $\omega < 2.517$.

Coppersmith and Winograd [19] were the first to break the 2.5 threshold with $\omega < 2.496$. Strassen decreased again the bound to $\omega < 2.479$ using his new *laser* technique [64]. In 1987 Coppersmith and Winograd combined Strassens laset technique with a novel form of analysis based on large sets avoiding arithmetic progressions, and obtained the famous Coppersmith-Winograd algorithm [20] which achieves the bound of $\omega < 2.376$. In 2003, Cohn and Umans [17] introduced a new, group-theoretic framework for designing and analyzing matrix multiplication algorithms which led, through the collaboration with Kleinberg and Szegedy [16], to several novel matrix multiplication algorithms which however were not able to improve over the Coppersmith-Winograd algorithm. In 2014 Vassilevska-Williams [70] presented new tools for analyzing matrix multiplication constructions similar to the Coppersmith-Winograd construction, obtaining a new improved bound on $\omega < 2.372873$.

Despite the consistent asymptotical improvement provided by these fast matrix multiplication algorithms over the naive algorithm, the latter is still largely used in practice. This fact can be explained by analyzing two common drawbacks of fast matrix multiplication algorithms. First, the reduction in the number of arithmetic operations with respect to the Naive algorithm comes at the price of a reduced numerical stability [44]. Second, the constants terms of the upper bounds *hidden* by the asymptotic notation are so big that a true performance improvement would be achievable only for extremely big matrices, which are of little to none interest for todays applications.

4.2 Strassen's matrix multiplication algorithm

In this section, we present in detail the first fast matrix multiplication algorithm introduced by Volk Strassen in 1969 [63], which is the focus of our analysis.

4.2.1 Description of the algorithm

Let A, B be the two factor square matrices whose entries are drawn from the ring $\mathcal{R}$, and let C be the product matrix. It is possible to partition A, B and C into 4 equally sized bloc matrices as follows:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \tag{4.3}$$

Strassen's algorithm computes the following seven matrix products:

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})\, B - 1, 1 \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})\, B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned} \tag{4.4}$$

It is then possible to compute the sub-matrices $C_{i,j}$ as a combination of the products M_k:

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned} \tag{4.5}$$

This division process is iterated recursively $\log n$ times (recursively) until the sub-matrices degenerate into single numbers (elements of $\mathcal{R}$).

Given two factor matrices A and B of size $n \times n$, with $n > 1$, Strassen's algorithm generates seven sub-problems for which the input matrices have size $\frac{n}{2} \times \frac{n}{2}$. Following this recursive structure, the number of sub-problems for which the input matrices have size $\frac{n}{2^i} \times \frac{n}{2^i}$ will be:

$$7^{\log\left(\frac{n}{\frac{n}{2^i}}\right)} = 7^i \tag{4.6}$$

Given how entries of A and B are used to generate the input of the seven sub-problems, some of the input values of the latter will correspond to values of A and B. It should however be remarked that none of the seven sub-problem generated share any of their input. Applying this consideration to the 7^i sub-problems with input of size $\frac{n}{2^i} \times \frac{n}{2^i}$ generated by the recursive structure of Strassen's algorithm we can conclude that none of them have any common input value.

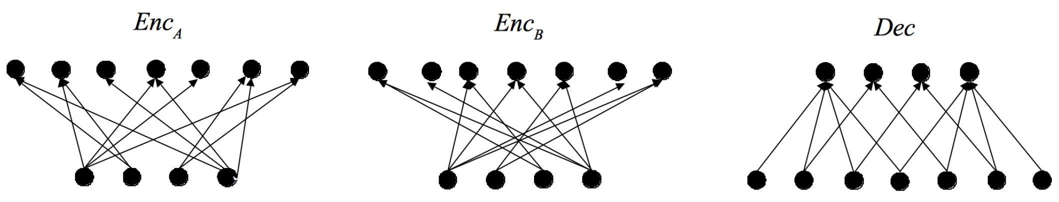
The number of additions and multiplications required in the Strassen's algorithm can be calculated through a simple recurrence equation. Let $f(n)$ be the number of operations required for computing the product of two $n \times n$ matrices. Then by recursive application of the Strassen's algorithm, we see that $f(n) = 7f(n/2) + ln^2$, for some constant l that depends on the number of additions performed at each application of the algorithm. We thus have $f(n) = (7 + o(1))^{\log n}$, and we can therefore conclude that the asymptotic complexity for multiplying matrices of size $n \times n$ using the Strassen algorithm is:

$$\mathcal{O}(7 + o(1))^{\log n} \approx \mathcal{O}(n)^{\log 7} \tag{4.7}$$

4.2.2 Construction of the CDAG of Strassen

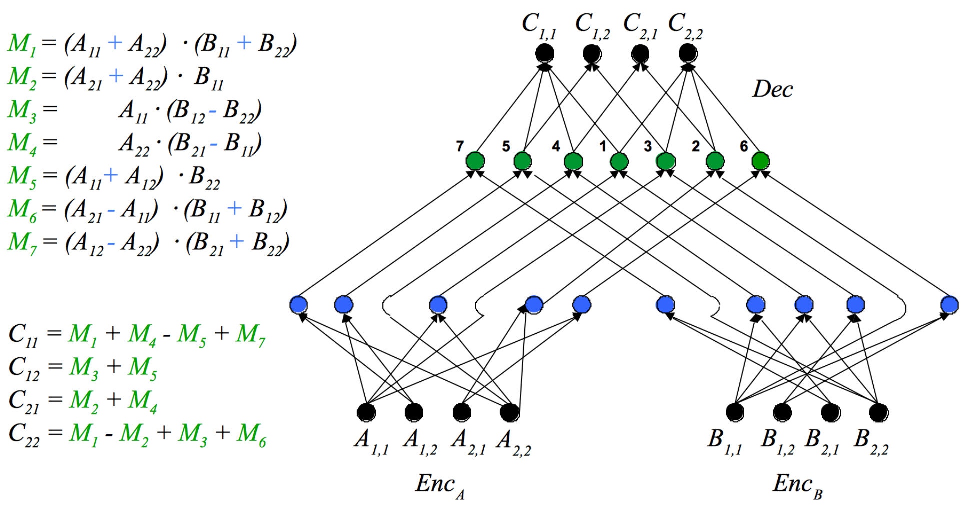
The execution of Strassen's algorithm on a given input can be modeled as a CDAG where each vertex represents either a value of one of the input matrices A and B, an intermediate result computed during the execution of the algorithm, or a value of the output matrix C; each edge represent a functional dependence between two values.

Let $H^{n \times n}$ denote the CDAG of Strassen's algorithm for input matrices of size



(a) Enc_A (b) Enc_b (c) Dec

Figure 4.3: Basic constructing blocks of Strassen's algorithm CDAG. Note that Enc_A and Enc_B are isomorphic.



$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\ M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\ M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\ M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{aligned}$$

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Figure 4.4: Strassen's $H^{2\times 2}$ CDAG: blue vertices represent combinations of the input values from the factor matrices A and B which are then used as input values for the sub-problems; red vertices represent the output of the seven sub-problems which are then used to compute the output values of the product matrix C.

$n \times n$. For $n > 2$, the CDAG $H^{n\times n}$ can be obtained by appropriately combining seven copies of $H^{n/2\times n/2}$ using a recursive construction which mirrors the recursive structure of Strassen's algorithm. The base of the construction, reported in Figure 4.4, is the CDAG $H^{2\times 2}$ which corresponds to the multiplication of two 2×2 matrices using Strassen's algorithm. Entries from A and B are combined (i.e., "*encoded*) into seven pairs of inputs of the seven sub-products M_i, using an *encoder* sub-CDAG Enc_A (Figure 4.3a) for the values from the factor matrix A, and respectively Enc_B (Figure 4.3b) for the values from the factor matrix B. The output values of the sub-products M_i are then combined using a *decoder* Dec sub-CDAG (Figure 4.3c) in order to compute the output values corresponding to the entries of the product matrix C. Note that the structure of Enc_A and Enc_B correspond to how entries of A and B are combined to generate the inputs to the $M_1, M_2, \ldots, M_7$ products; while the structure of Dec corresponds to how the products $M_1, M_2, \ldots, M_7$ are combined in order to compute the values of the output matrix C. Note furthermore that while

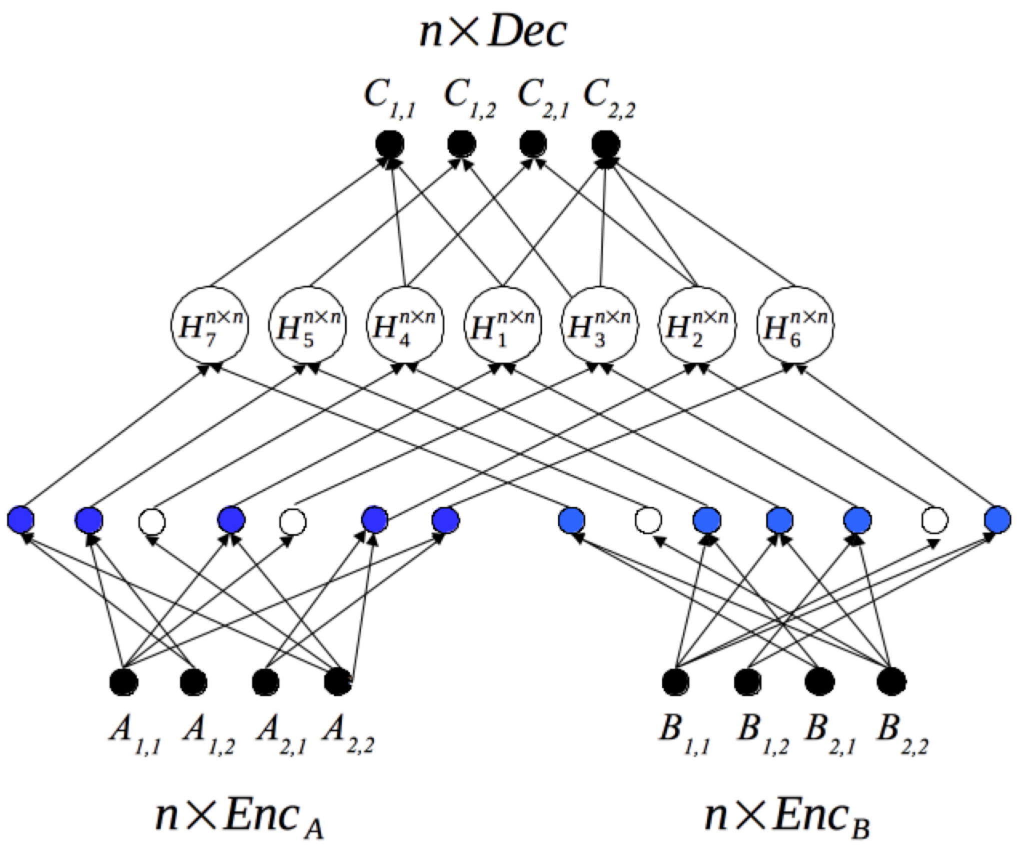


Figure 4.5: Recursive construction of Strassen's $H^{2n\times 2n}$ CDAG: blue vertices represent combinations of the input block sub-matrices of A and B which are then used as input values for the sub-problems $H^{n\times n}$; white vertices represents an input matrix for one of the sub-problems $H^{n\times n}$ which corresponds to one of the block sub-matrices of the input matrices A and B; the output vertices represent the four sub block-matrices of C.

Enc_A and Enc_B are isomorphic, they are not isomorphic to Dec.

The CDAG $H^{2n\times 2n}$, which represents the Strassen's algorithm for input matrices of size $2n \times 2n$, can be constructed by composing seven copies of $H^{n\times n}$, each of which corresponds to one of the seven sub-products generated by the algorithm. A schematic representation od the construction is presented in Figure 4.5. Entries from the factor matrices A and B are combined into seven pairs of input matrices of size $n \times n$ for the seven sub-products $H_i^{n\times n}$, with $i \in \{1, 2, \ldots, 7\}$. This combination is realized by using $2^{\log(2n)-1} = n$ vertex-disjoint copies of, respectively, the encoder CDAGs Enc_A and Enc_B. The encoders are used to connect the input vertices of $H^{2n\times 2n}$ (which correspond to the input values of the global product) to the opportune input vertices of the seven sub-CDAGs $H_i^{n\times n}$. The outputs of the seven sub-products are then combined to compute the output matrix C. Said combination is realized in the CDAG by connecting the vertices corresponding to the output of each of the seven sub-CDAGs $H_i^{n\times n}$ to the opportune output vertices of the entire $H^{2n\times 2n}$ CDAG using $2^{\log(2n)-1} = n$ copies of the decoder sub-CDAG *Dec*.

In Strassen's algorithm, we have that some of the input values of the seven sub-problems M_i correspond to the input values of the main problem. This situation is represented in the encoder CDAGs by an output which is connected by just one output. Although the provided "*layered*" representation is useful to gain some intuition on the structure of Strassen's CDAG, it is important to keep in mind that any pair of vertices such that one of the two has the other as unique predecessor, actually corresponds to one unique vertex.

As previously stated, the 7^i sub-problems with input of size $\frac{n}{2^i} \times \frac{n}{2^i}$ generated by the recursive structure of Strassen's do not share any input values. The corresponding $H^{n/2^i \times n/2^i}$ do not therefore share any input vertex. From this observation and the recursive construction of Strassen's algorithm CDAG we can state the following lemma.

Lemma 4.4. *Let $H^{n\times n}$ denote the CDAG of Strassen's algorithm for input matrices of size $n \times n$. For $0 \leq i \leq \log n - 1$, there are exactly 7^i sub-CDAGs $H^{n/2^i \times n/2^i}$ which do not share any vertex in $H^{n\times n}$ (i.e., they are* vertex disjoint sub-CDAGs of *$H^{n\times n}$).*

4.3 Communication model

In our analysis, we assume that sequential computations are executed on a system with a two-layer memory hierarchy consisting of a fast memory of limited size M (i.e., the *cache* memory) and a slow memory of unlimited size. In the following, we use the expression "*size* of the memory space" to indicate the number of words which can maintained in the memory. The execution of a straight-line program on this model can be analyzed via the *Red-Blue pebble game* proposed by Hong and Kung [37], which is played on the corresponding CDAG. Here we present it according to its formalization in [58].

The red (*hot*) pebbles identify values held in the cache while the blue (*cold*) pebbles identify values held in a secondary memory. Correspondingly, the number of available red pebbles is given by the maximum number of words which can be maintained in the cache (hence M), while unlimited blue pebbles are available. We assume the values identified with the pebbles as words. A pebble placed on a vertex identifies that the value associated to that vertex (input, output, intermediate result) in a location of the corresponding type of memory. At the instant before the game starts, blue pebbles reside on all input vertices, while there are no red pebbles on the CDAG. The goal of the game is to place a blue pebble on each output vertex, that is, to compute the values associated with these vertices and write them in the slow memory. These assumptions capture the idea that is initially stored in the slow memory (synthesized in rule R1), and that the results, once computed, must be deposited there as well (synthesized in rule R4). The rules of the Red-Blue pebble game are the following:

(R1) Initialization: A blue pebble can be placed on an input vertex at any time.

(R2) Computation Step: A red pebble can be placed on a vertex if all its immediate predecessors carry red pebbles.

(R3) Pebble Deletion: A pebble can be deleted from any vertex at any time.

(R4) Goal: A blue pebble must reside on each output vertex at the end of the game.

(R5) Input form secondary memory: A red pebble can be placed on any vertex carrying a blue pebble.

(R6) Output to secondary memory: A blue pebble can be placed on any vertex carrying a red pebble.

Rule (R2) formalizes the requirement that all the arguments on which a function depends must reside in primary memory before the function can be computed. The third rule (R3) allows the removal of a pebble from a vertex, which corresponds to the deletion of the corresponding value from the memory (either cache or slow depending on the color of the pebble). If a pebble is removed from a vertex that later needs a red pebble, then said vertex has to be *repebbled.* Rules (R5) and (R6) model the communications between cache and slow memory. The execution of a "*read*" operation from slow memory to the cache is captured by R5, while the execution of a "*write*" operation of a value contained in the cache to slow memory by R6. In the following we refer to both read and write as *I/O* operations.

Each pebbling strategy corresponds therefore to a computation of the algorithm, such that the execution of each step in the pebbling game correspond to the execution of either a computational or I/O operation for the computation.

A pebbling strategy $\mathcal{P}$ is given by the succession of the executions of the rules of the pebble game on the vertices of a given CDAG. We refer to each element a pebbling strategy as "*step*". For a given $\mathcal{P}$ its *computational time*, i.e., the number of computations executed, corresponds to the number of executions of the (R2) rule, while its *I/O time* corresponds to the number of executions of the (R4) and

(R5) rules. Different pebbling strategies the same CDAG may differ greatly in their computational and I/O time.

The *I/O complexity* $IO_G(M)$ of a straight line program represented by the CDAG G is defined as the minimum number of I/O operation necessary for its execution on a platform equipped with a cache of size M and unlimited slow memory. A lower bound for this quantity can be obtained as the minimum number of I/O operations executed by any of the possible pebbling strategies for the pebble game played with M red pebbles on the CDAG G corresponding to the algorithm. Expanding this definition we have that the *I/O complexity* of a problem is the minimum I/O complexity of any algorithm which computes the solution to said problem.

Access complexity A variation of the I/O complexity concept called *Access complexity* of an algorithm, was introduced by Bilardi and Preparata in [11]. The main difference with the definition of I/O complexity is given by the fact that, before the beginning of the computation up to M input values can be stored in the cache. This would imply a slight modification of the blue pebble game, as it would be possible for up to M red pebbles to be placed on input vertices of the CDAG before the beginning of the game. Let $Q_G(M)$ denote the access complexity of a CDAG G (and therefore of the corresponding algorithm) when executed on a platform equipped with a cache of size M and unlimited slow memory. Clearly we have $IO_G(M) \geq Q_G(M)$.

Pebbling strategies with or without restrictions on recomputation The rules of the red-blue pebble game allow to remove any red pebble placed on a vertex at a certain step s_1 without placing a blue pebble on it, and then place again a red pebble on the same vertex during a subsequent step s_2 even though the vertex is not carrying a blue pebble at $s_2 - 1$.

Such "*repebbling*" operation corresponds to multiple evaluations (*recomputations*) of the value of the operation associated to a vertex during the execution of the algorithm. Recomputing intermediate values during the computation may allow for some saving on the I/O time at the price of an increase of the computational time. An important class of computations are those for which no intermediate value is ever computed more than once. We refer to pebbling strategies in this class as *computations without recomputations* or *nr-computations* for short. Corresponding to the nr-computations, we define the nr-pebbling strategies for the red-blue pebble game as those strategies for which, once a pebble either red or blue have been has been placed on a vertex, said vertex retains a pebble, either red or blue, until a red pebble has been placed on all its successors. A variant of the original red-blue pebble game for which only nr-pebbling strategies are allowed could be obtained by modifying rule (R2):

(R2-NR) No-repebbling condition: A red pebble can be placed on a vertex if all its immediate predecessors carry red pebbles and if no red pebble was placed on the vertex during previous steps.

nr-pebbling strategies are of practical interest since they achieve minimum computational time.

Note that the assumption of no recomputation provides a very strong control over the lifetime of data in memory. Under this assumption, once the intermediate result of an operation has been computed for the first time, it is mandatory to keep it stored in memory until all the values of the operations which use it as an operand have been computed. Such values can either be kept in cache or moved back and forth from the slow memory through the use of I/O operations. Because of this strength, the assumption of no recomputation has frequently been used in literature to study the I/O complexity of algorithms. In particular, all previous result on the I/O complexity of Strassen's matrix multiplication algorithm have been obtained under the no-recomputation assumption [7, 62].

4.4 Previous work

Besides formalizing the Red-Blue pebble game in their seminal work [37], Hong and Kung proposed the $S-partitioning$ technique to obtain lower bounds to the I/O complexity of straight line programs. In the same work, they applied their technique to obtain lower bounds on the I/O complexity of various algorithms such as the FFT, and the naive algorithms for computing the vector-matrix and the product of rectangular matrices. In [57] Savage introduced an extension of the Red-Blue pebble game to multiple levels of memory hierarchy and a lower bound technique based on the concept of *S-Span* of a CDAG. This is a measure that intuitively represents the maximum amount of computation that can be done after loading data in a cache at some level without accessing higher level memories. Applications of the S-Span technique was used Savage et al. to obtain lower bounds to the I/O complexity of r-pyramids CDAGs in [53]. In [10], Bilardi et al. presented the *S-covering partition* technique which merges and extends aspects from both [37] and [57]. A further generalization of the model by Hong e Kung [37] called Hierarchical Memory Machine (HMM) was introduced by Aggarwal, Alpern, Chandra, and Snir in [1].

In [9] Bilardi and Preparata introduced a variation of the concept of I/O complexity called "*Access Complexity*" (discussed in Section 4.3). In [11] the same authors introduced the "*Dichotomy Width*" technique which allows to obtain lower bounds to the Access complexity of algorithms under the assumption that no intermediate value is computed more than once (no-recomputation assumption). In Ranjan et al. introduced the "*Boundary Flow*" technique and used it to derive lower bounds for the I/O complexity for the Binomial and FFT Computation Graphs for computations with no recomputations. Ballard et al. generalized the results on matrix multiplication of Hong and Kung [37] in [6, 5] by using the approach proposed in [36] based on the Loomis-Whitney geometric theorem [42, 67], by embedding segments of the computation process into a three dimensional cube. In the same work, the authors obtained algorithms and matching I/O complexity lower bounds for various classical linear algebra algorithms such as LU factorization, Cholesky factorization, LDLT factorization, QR factorization, as well as algorithms for eigenvalues and singular values. These results hold for dense matrix algorithms (most of them have $\mathcal{O}(n^3)$ complexity), as well as sparse matrix algorithms (whose running time depends on the number of non-zero elements, and their locations). In [23] Elango et al. proposed a technique allowing the combination of lower bounds on the I/O complexity of sub-CDAGS assuming that no intermediate value can be recomputed.

A lower bound to the I/O complexity of the naive $\mathcal{O}(n^3)$ algorithm was originally obtained in the seminal work by Hong and Kung [37]. While naive implementations of this algorithm are non communication-efficient, communication-minimizing sequential [14] and parallel [29] algorithms have been presented in memory. Since the asymptotic complexity of these algorithms matches the lower bounds respectively in [37] and in [36] these algorithms are optimal. The first results on Strassen's algorithm was achieved by Ballard et al. [7] using the "*edge expansion approach*". This technique relates the I/O-complexity of an algorithm to the edge expansion properties of the undirected underlying graph corresponding to the CDAG representing the algorithm. This technique can be extended to *Strassen-like* algorithms, but fails for algorithms with base graphs (correspondent to $H^{2\times 2}$) containing disconnected ecoder or decoder graphs. The edge expansion approach was later extended [4] to fast recursive matrix multiplication algorithms for rectangular matrices whose base graphs consist of multiple equal-size connected components. The "*path routing*" technique introduced by Scott et al. [62] allows to obtain the same lower bound for Strassen's algorithm obtained in [7]. This new technique can however be generalized to any recursive fast matrix multiplication algorithms involving arbitrary base graphs, as long as the same base graph is used at each recursive step. While the lower bounds for the naive algorithm in [37, 36] hold for any possible computation, all known lower bounds on the I/O complexity of Strassen's (and Strassen-like) algorithm, including those in [7, 62] were obtain under no-recomputation assumption. A communication avoiding implementation of Strassen's algorithm whose performance matches the lower bound for nr-computations [7, 62], was proposed by Ballard et al. [3].

In our contribution, we present an alternative, much simpler, technique to achieve the state of the art asymptotical lower bound for nr-computations. We then extend this result by removing the restrain on recomputation to obtain a novel tight lower bound for all possible pebbling strategies for Strassen's matrix multiplication algorithm.

4.5 Lower bound for computations with no recomputation

In this section, we present a lower bound on the I/O complexity of Strassen's matrix multiplication algorithm for computations for which no intermediate value is computed more than once (i.e., nr-computations) as defined in Chapter 1. This class of computations correspond to the class of recomputation restrained pebbling strategies in the red-blue pebble game as described in Section 2.1. Although our bound corresponds asymptotically to the one presented in [7, 62], our proof technique is much simpler than the ones previously presented in literature. Previous results on the I/O complexity for Strassen's algorithm are based on the analysis of specific combinatorial properties of the CDAG $H^{n\times n}$ representing the algorithm's execution such as the *edge expansion* [7] or the path routing property [62]). Our approach instead relates the I/O complexity of the algorithm to its recursive structure and in particular to the number of sub-problems generated.

Recall from Section 2.1 that the free input space complexity of nr-computations of the CDAG $S_{free-nr}(G)$ is defined as the minimum memory space necessary for execution of any free-input nr-computation of G in which no intermediate value is computed twice (i.e., in the pebble game no vertex is pebbled twice).

For the square matrix multiplication function $f_{n\times n} : \mathcal{R}^{2n^2} \rightarrow \mathcal{R}^{n^2}$ defined over the ring $\mathcal{R}$, there is significant relationship between the information flow property of $f_{n\times n}$ and the free input space complexity for nr-computations of *any* CDAG $G(I \cup V, E)$ which corresponds to a straight line program which computes $f_{n\times n}$. Recall that use the expression "*size of the memory space*" to indicate the number of words which can maintained in the memory. We assume furthermore that any memory word can be used to memorize a single value of the ring $\mathcal{R}$.

Lemma 4.5. *Let $G_{n\times n}(I \cup V, E)$ be the CDAG corresponding to the execution of* *any straight line program for the square matrix product function $f_{n\times n}$ defined over the ring $\mathcal{R}$. We have:*

$$S_{free-nr}(G_{n\times n}) \geq \frac{n^2}{4}. \tag{4.8}$$

Proof Consider any free input pebbling strategy $\mathcal{C}$ which computes the product matrix $C = AB$ without recomputing any intermediate result during its execution with $A, B, C \in \mathcal{R}^{n\times n}$. Let $\mathcal{C}_p$ be the shortest prefix of $\mathcal{C}$ during which a total of $\alpha 2n^2$, for $0 \leq \alpha \leq 1$, distinct input vertices of G are pebbled (i.e. distinct input values of A or B loaded into memory). Let $\mathcal{S}_s$ the corresponding suffix. The maximum number of complete rows of A or columns of B loaded into memory during the execution of $\mathcal{S}_p$ is $2\alpha n$. In order to compute one of the entries of $c_{i,j}$ of the product matrix C it is necessary to load into fast memory all the values in the i-th row of A and of the j-th column of B. The maximum number of values form C which can be computed in $\mathcal{S}_p$ is therefore $\alpha^2 n^2$ while the remaining $n^2 - \alpha^2 n^2$ will be computed during the execution of $\mathcal{S}_s$. From Lemma 4.3 we have that the information flow $w_{n\times n}(\alpha 2n^2, n^2(1-\alpha^2))$ between the $\alpha 2n^2$ input values loaded into memory during $\mathcal{S}_p$ and the $n^2(1-\alpha^2)$ output values computed during $\mathcal{S}_s$ is at least:

$$\begin{aligned} w_{n\times n}\left(\alpha 2n^2, n^2\left(1-\alpha^2\right)\right) &\geq \frac{1}{2}\left(n^2\left(1-\alpha^2\right) - \frac{\left(2n^2-\alpha 2n^2\right)^2}{4n^2}\right) \\ &= \frac{1}{2}\left(n^2\left(1-\alpha\right)\left(1+\alpha-\left(1-\alpha\right)\right)\right) \\ &= n^2\left(1-\alpha\right)\alpha. \end{aligned}$$

In order to determinate the value of α which maximizes $w_{n\times n}(\alpha 2n^2, n^2(1-\alpha^2))$ we shall consider it derivative in α:

$$\frac{d}{d\alpha} w_{n\times n}\left(\alpha 2n^2, n^2\left(1-\alpha^2\right)\right) = \frac{d}{d\alpha} n^2\left(1-\alpha\right)\alpha = n^2\left(1-2\alpha\right) \tag{4.9}$$

For $\alpha = 1/2$ the derivative in equation 4.9 equals zero, and the quantity $w_{n\times n}(\alpha 2n^2, n^2(1-\alpha^2))$ is maximized:

$$w_{n\times n}\left(n^2, \frac{3}{4}n^2\right) = \frac{n^2}{4}.$$

Let X_0 and X_1 denote the set input values loaded in memory respectively during $\mathcal{C}_p$ and $\mathcal{C}_s$. Also let Y_1 denote the set of output values computed during $\mathcal{C}_s$. By definition, there exists an assignment to the X_0 inputs such that such that the Y_1 outputs can assume at least $\mathcal{R}^{w_{n\times n}\left(n^2,\frac{3}{4}n^2\right)} = \mathcal{R}^{\frac{n^2}{4}}$ different values. Suppose a memory space of size less than $w_{n\times n}\left(n^2, \frac{3}{4}n^2\right)$ is being used during the computation, the values Y_1 assume more values than those which can be maintained in memory. Since no intermediate result computed during $\mathcal{C}_p$ can be computed a second time during $\mathcal{C}_s$ there are at least the memory space required by the computation is at least $w_{n\times n}\left(n^2, \frac{3}{4}n^2\right) = n^2/4$. □

$S_{free-nr}(G)$ can be used to obtain a straightforward lower bound to the I/O complexity for nr-pebbling strategies of G itself.

Lemma 4.6. *Let us consider the execution of an algorithm corresponding to a CDAG $G(I \cup V, E)$ in a system with a two level memory hierarchy, where the fast memory (cache) has size M. We have:*

$$IO_G(M) \geq 2\max\{0, S_{free-nr}(G) - M\} \tag{4.10}$$

Proof By definition, any schedule with no recomputation for G uses at least $S_{fn}(G)$ memory space, since the fast memory has size M it will be necessary to use at least $S_{free-nr}(G) - M$ slow memory cells. Furthermore, usage of slow memory will require both a write and a read operation hence the constant 2 in the bound. □

A generalization of this result was introduced by Bilardi et al. [10]:

Theorem 4.7 (Theorem 2 [10])**.** *Let G be a CDAG with h-vertex-disjoint sub-CDAGs $G_1, G_2, \ldots, G_h$. We have:*

$$IO_G(M) \geq \sum_{i=1}^{h} 2\max\{0, S_{free-nr}(G_i) - M\}. \tag{4.11}$$

Proof Let us consider any standard nr-computation $\mathcal{C}$ of G and let $\mathcal{C}_\rangle$ be the free input sub-computations relative to each of the sub-CDAGs G_i. For the same reasoning used in the proof of Lemma 4.6 we have that at least $S_{free-nr}(G_i)$ distinct memory cells are accessed to read/write values of G_i. Since the cache has size M, it will be necessary to use at least $S_{free-nr}(G) - M$ slow memory cells. Finally, since the sub-CDAGs are vertex-disjoint, they do not share any values. The lemma follows. □

We shall now use these results to obtain a lower bound on the access complexity for nr-computations of Strassen's matrix multiplication algorithm.

Theorem 4.8 (Lower bound I/O complexity Strassen's matrix multiplication for computations with no recomputation)**.** *Consider Strassen's matrix multiplication algorithm being used to multiply two square matrices of size $n \times n$ whose entries are* *drawn from the ring $\mathcal{R}$. Assuming no intermediate result is ever computed more than once, the I/O-complexity of Strassen's algorithm when run on a sequential machine with fast memory of size M is:*

$$IO_{H^{n\times n}}(M) \geq \frac{1}{7}\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M \tag{4.12}$$

Proof In the following proof we assume without loss of generality that $n = 2^a$ and $\sqrt{M} = 2^b$ for some $a, b \in \mathbb{N}$. At least $3M$ I/O operations are necessary in order to read all the $2n^2$ input values form slow memory to the cache and to write the n^2 output values to the slow memory once they have been computed. The statement of the theorem is therefore trivially if $n \leq 2\sqrt{M}$. In the following we will consider the case for $n \geq 4\sqrt{M}$ Let us consider one of the sub-problems generated by the recursive structure of Strassen's algorithm which has input matrices of size $\alpha\sqrt{M} \times \alpha\sqrt{M}$, where α is a constant such that $\alpha\sqrt{M}$ is a power of two. From Lemma 4.4 we have that the CDAG $H^{n\times n}$ which corresponds to the execution of Strassen's algorithm for input matrices of size $n \times n$ contains $\left(n/\alpha\sqrt{M}\right)^{\log 7}$ vertex-disjoint sub-CDAGs $H^{\alpha\sqrt{M}\times\alpha\sqrt{M}}$. Each of these sub-CDAGs corresponds to one of the distinct sub-problems with input size $\alpha\sqrt{M} \times \alpha\sqrt{M}$ generated by Strassen's algorithm at the $\log(n/\alpha\sqrt{M})$-th level of the recursion. Let us now consider one such sub-CDAGs $H^{\alpha\sqrt{M}\times\alpha\sqrt{M}}$. From Lemma 4.5, we have $S_{free-nr}(H^{\alpha\sqrt{M}\times\alpha\sqrt{M}}) \geq \alpha^2 M/4$. Since the $\left(n/\alpha\sqrt{M}\right)^{\log 7}$ of $H^{\alpha\sqrt{M}\times\alpha\sqrt{M}}$ are vertex-disjoint we can apply Theorem 4.7:

$$IO_{H^{n\times n}}(M) \geq \sum_{i=1}^{\left(\frac{n}{\alpha M}\right)^{\log_2 7}} 2\left(S_{free-nr}(H^{\alpha\sqrt{M}\times\alpha\sqrt{M}}) - M\right) = c\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M \tag{4.13}$$

where:

$$c = \frac{1}{2}\left(\frac{\alpha^2 - 4}{a^{\log 7}}\right)$$

In order to determinate the value of α which maximizes 4.13, we study the derivative of c in α:

$$\frac{d}{d\alpha}c = \frac{1}{2}\left(\frac{2\alpha^{1+\log_2 7} - (\alpha^2 - 4)\log_2 7\alpha^{\log_2 7 - 1}}{a^{2\log_2 7}}\right).$$

Through some algebraic manipulation, we can see that the previous derivative goes to zero for:

$$\alpha^* = \sqrt{\frac{4\log 7}{\log 7 - 2}} = 3.7294639034.$$

By further studying the sign of the derivative, we can conclude that c assumes maximum value for α^*. We must chose α in such a way that $\alpha\sqrt{M}$ is still a power of two. Since we are studying the case for $n \geq 4\sqrt{M}$, we select $\alpha = 4$ and we thus obtain $c = 1/7$. □

Our proof technique is based on the analysis of the recursive structure of Strassen's algorithm and on the identifications of the sub-CDAGs corresponding to the various sub-problems. The only property of these sub-CDAGs we consider is their being vertex-disjoint with respect to the overall CDAG.

On mixed matrix multiplication strategies

An interesting aspect of our proof technique is that no constraint is placed on the specific straight-line algorithm which is to be used for computing the sub-problems of size $\alpha\sqrt{M} \times \alpha\sqrt{M}$. The lower bound on the free input space complexity for nr-computations obtained in Lemma 4.5 holds in fact for any straight line program being used to compute the square matrix multiplication function.

This observation implies that the I/O complexity of *mixed* algorithmic strategies in which Strassen's algorithm is initially used to generate sub-problems of smaller size (e.g., sub-problems which can be entirely computed in cache) which are then computed using a different algorithm (e.g. the naive algorithm) can still be captured by our technique. The following Lemma formalizes this observation:

Lemma 4.9. *Consider a mixed matrix multiplication strategy for which Strassen's algorithm is initially used to reduce the input matrices of size $n \times n$, whose entries are drawn from the ring $\mathcal{R}$, by generating $\left(n/2^i\sqrt{M}\right)^{\log_2 7}$ sub-products of size $2^i\sqrt{M} \times 2^i\sqrt{M}$ for $0 \leq i \leq \log_2(n/2\sqrt{M})$. The single sub-product are then computed using any straight line algorithm for matrix multiplication. Let $H^{n\times n,i}$ denote the CDAG corresponding to this mixed strategy. Assuming no intermediate result is ever computed more than once, the I/O complexity any such strategy when run on a sequential machine with fast memory of size M is:*

$$IO_{H^{n\times n,i}}(M) \geq \left(\frac{n}{2^i\sqrt{M}}\right)^{\log_2 7} \frac{M}{4}\left(2^i - 1\right) \tag{4.14}$$

The proof of this lemma is given by a simple extension of the proof of Theorem 4.8. Although this bound is in general much more loose than the one provided by theorem 4.8, it provides an interesting indication of the I/O complexity for Strassen-mixed strategies.

Generalization to Strassen-like algorithms

Let us now consider a class of algorithms based on a recursive strategy according to which a matrix multiplication with input matrices of size $n \times n$ is solved by generating recursively ω sub-products of size $n/\beta \times n/\beta$ and then combining the results of the sub-problems to obtain the final result. Let $G^{n\times n}$ be the CDAG corresponding to this algorithm, if the sub-CDAGs corresponding each to one of the sub-problems generated at the i-th step of the recursion are all vertex-disjoint, we say that the straight line algorithm corresponding to $G^{n\times n}$ is (ω, β) *Strassen-like*. A straightforward extension of the result of Theorem 4.8 leads to the following, more general, lower bound for the I/O complexity of (ω, β) Strassen-like algorithms.

Theorem 4.10 (Lower bound I/O complexity Strassen-like algorithms for computations without recomputation)**.** *Let $H^{(\omega,\beta),n\times n}$ be the CDAG corresponding to a given (ω, β) Strassen-like matrix multiplication algorithm being used to multiply two square matrices of size $n \times n$ whose entries are drawn from the ring $\mathcal{R}$. Assuming no intermediate result is ever computed more than once, the I/O-complexity of the given algorithm when run on a sequential machine with fast memory of size M is:*

$$IO_{H^{(\omega,\beta),n\times n}}(M) = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_\beta \omega} M\right) \tag{4.15}$$

It is interesting to observe that the standard bloc matrix multiplication algorithm (also referred as Cannon algorithm), despite its recursive structure, it is not a Strassen-like algorithm since pairs of sub-problems may share some of their input values.

4.6 Lower bound for general computations

While the assumption of no recomputation appears reasonable when the goal is to minimize the computational time of a computation, it leaves open the question on whether a strategy in which an intermediate result can be computed multiple times (i.e., the correspondent vertex in the CDAG can be re-pebbled) can achieve a lower I/O time. Ideally, the possibility of computing certain results multiple times, could avoid the necessity of moving those between cache and slow memory. Besides the purely theoretical interest of the question, the possibility of achieving better I/O performances even at the cost of increase of the computational load would be very interesting in order to improve the performance of actual implementations.

In this section, we introduce a novel tight lower bound for the I/O complexity of Strassen's matrix multiplication algorithm which, differently from the result presented in Section 4.5 and previous similar results [7, 62], holds for any possible execution schedule without any restriction on the possibility of computing an intermediate value more than once.

Theorem 4.11 (Lower bound I/O complexity Strassen's matrix multiplication algorithm)**.** *Consider Strassen's matrix multiplication algorithm being used to multiply two square matrices of size $n \times n$ whose entries are drawn from the ring $\mathcal{R}$. The I/O-complexity of Strassen's algorithm when run on a sequential machine with fast memory of size M is:*

$$IO_{H^{n\times n}}(M) \geq \frac{1}{14}\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M. \tag{4.16}$$

The following sections will be devoted to the proof of Theorem 4.11. We provide here a high-level outline of our proof technique as a guide for the reader through the next sections.

Proof sketch for Theorem 4.11. Let $H^{n\times n}$ be the CDAG corresponding the computation of the product of two matrices A and B of size $n \times n$ using Strassen's algorithm. For $n \leq 2\sqrt{M}$ the statement is easily verified. If $n > 2\sqrt{M}$, in $H^{n\times n}$ there are $\left(\frac{n}{2\sqrt{M}}\right)^{\log_2 7}$ distinct sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$ each corresponding to one of the sub-problems for which the input matrices have size $2\sqrt{M} \times 2\sqrt{M}$.

Let $\mathcal{Z}$ denote the set of the $4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ output vertices of the $H^{2\sqrt{M}\times 2\sqrt{M}}$ sub-CDAGs. Vertices in $\mathcal{Z}$ represent the output values of the corresponding sub-problems. All the vertices in $\mathcal{Z}$ are pebbled at least once by any pebbling strategy for $H^{n\times n}$ in the red-blue pebble game.

Let $\mathcal{P}$ be any pebbling strategy (computation) for the Red-Blue pebble game played on $H^{n\times n}$ with M available red pebbles. We partition $\mathcal{P}$ in segments such that exactly $4M$ distinct vertex from $\mathcal{Z}$ are pebbled for the first time in each of them. In the main part of the proof, we will then show that at least $M/2$ IO operations must occur during each sub-computation corresponding to one of the segment. Since $\mathcal{P}$ is dividend into $\frac{1}{7}\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M$ segments, we will conclude that at least $\frac{M}{2}\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ IO operations will be executed in $\mathcal{P}$.

Before delving into the details of the proof of Theorem 4.11, it is important to stress why the proof technique discussed in Section 4.5 can not be directly used to achieve the more general result. Clearly, the result in Lemma 4.5 does not hold if the restriction on recomputation is removed. There may exists in fact schedules that require less memory space as some intermediate results are computed multiple times starting from the input vertices. There exists in fact pebbling strategies for $H^{n\times n}$

which require space $\mathcal{O}(\log n)$. It is therefore not possible to apply Theorem 4.7 to obtain the desired result.

4.6.1 Technical lemmas

We present here some technical lemmas, which we will then use for the proof of Theorem 4.11.

Relation between information flow of a function and dominator set

The *dominator set* concept was originally introduced in [37].

Definition 4.12 (Dominator set). *Given a CDAG $G(I \cup V, E)$, A* dominator *set for $V' \subseteq I \cup V$ is a set of vertices in V such that every path from a vertex in I to the vertices in V' contains at least a vertex of the set. A* minimum dominator set *for V' is a dominator set with minimum cardinality.*

In our proof we will use a specular concept, commonly referred as *post-dominator set* in literature.

Definition 4.13 (Post-dominator set). *Given a CDAG $G(I \cup V, E)$, let $O \subset V$ denote the set of output vertices. A* post-dominator set *for $V' \subseteq V \setminus O$ with respect to $O' \subseteq O$ is defined to be a set of vertices in V such that every path from a vertex in V' to the output vertices in O' contains at least a vertex of the set. A* minimum post-dominator *set for $V' \subseteq V \setminus O$ with respect to $O' \subseteq O$ is a post-dominator set with minimum cardinality.*

The next lemma highlights an interesting relation between the information flow of a function $f(\cdot)$ and the minimum size of the post-dominator of a subset of the input vertices of the CDAG corresponding to the straight line program used to compute $f(\cdot)$.

Theorem 4.14. *Let $G(I \cup V, E)$ be the CDAG corresponding to any straight line algorithm that computes a given function $f(\cdot) : \mathcal{A}^n \to \mathcal{A}^m$ defined on the ring $\mathcal{A}$ with information flow $w_f(u, v)$. Let $I \subset V$ and $O \subset V$ denote respectively the set of input and output vertices of G. Any minimum post-dominator set for any subset $I' \subseteq I$ with respect to any subset $O' \subseteq O$ has size at least $w_f(|V'|, |O'|)$.*

Proof The proof is by contradiction. Given $I' \subseteq I$ and $O' \subseteq O$, suppose the values of the input variables corresponding to vertices in $I \setminus I'$ to be fixed. Let Γ be a post-dominator set for I' with respect to O'. According to the hypothesis on the information flow of the function f, there exists an assignment of the input variables corresponding to vertices in I' such that the output variables in O' can assume $|\mathcal{A}|^{w_f(|I'|,|O'|)}$. As there is no path from I' to O' which has not a vertex in Γ, the values of the outputs in O' can be determined by the inputs in $I \setminus I'$, which are fixed, and the values corresponding to the vertices in Γ. If $\Gamma < w_f(|I'|, |O'|)$, the outputs in O' can assume more values than can be taken by the $|\Gamma|$ values corresponding to the post-dominator Γ, which leads to a contradiction. □

Connection properties of *Enc* sub-CDAGs

Here we discuss a property of the encoder sub-CDAGs Enc_A and Enc_B which will be then used in the latter stages of the proof.

Lemma 4.15. *Given an encoder CDAG, for any subset Y of its output vertices, there exists a sub-set X of its input vertices of size $g(|Y|) \leq |X| \leq |Y|$ such that each vertex in X can be connected to a distinct vertex in Y.*

$\lvert Y\rvert$	0	1	2	3	4	5	6	7
$g(\lvert Y\rvert)$	0	1	2	2	3	3	4	4

Proof We provide the proof for Enc_A, a the results holds for Enc_B as they are isomorphic. Note that in Enc_A there are some pairs of input-output vertices u, v for which the input vertex u is the only predecessor of the output vertex v. This implies that the two vertices are really one unique vertex. With a little abuse of notation, we will still say that u can be connected to v via a single edge.

We assign an index to each of the output vertices of Enc_A according to how is indicated in Figure 4.6. Note that the index assigned to each output corresponds to the index of the sub-problem generated by Strassen's algorithm for which the corresponding value is used as in input (see Figure 4.4 and Figure 4.5 in Section 4.2.2).

In order to verify that this lemma holds, we study all possible compositions of a subset Y of the output vertices of Enc_A. Each of these compositions is identified by a vector $\mathbf{y}$ with seven components, where $y_i = 1$ if the i-th output of Enc_A is in Y or zero otherwise, for $i \in \{1, 2, \ldots, 7\}$. In order to improve the presentation, we associate to each of the possible 128 compositions a *code* given by $\sum_{i=1}^{7} y_i 2^{7-i}$. In Table 1 (presented in the Appendix), we study each of the 128 possible compositions of Y, which are ordered by the value of $|Y|$ and by their code. The value in the last column $c(\mathbf{y})$ denotes the maximum size of a sub-set X of the input vertices of Enc_A such that each vertex in X can be connected to a distinct vertex in the subset Y corresponding to $\mathbf{y}$. Each of these values can be obtained through a straightforward

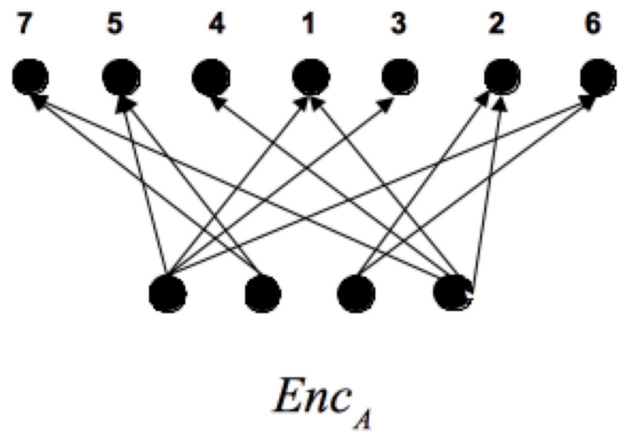


Figure 4.6: Detail of the *Enc* sub-CDAG.

analysis of Enc_A. The value of $g(|Y|)$ can then be obtained as the minimum value of $c(\mathbf{y})$ for all compositions of Y with the same cardinality:

$$g(a) = \min_{\mathbf{y}\in\{0,1\}^7 s.t.\ |\mathbf{y}|=a} c(\mathbf{y})$$

The lemma follows. □

Lemma 4.15 ensures therefore that, for any given subset Y of the output vertices there exists at least a subset X of the input vertices of the encoder, with $|Y| \geq |X| \geq g(|Y|)$, such that each vertex of X an be connected each to a distinct vertex in Y using vertex-disjoint paths, were each such path will be composed by just one edge.

Minimum size of a dominator set for subsets of $H^{n\times n}$

Lemma 4.16. *Let $H^{n\times n}$ be the CDAG which corresponds to the execution of Strassen's matrix multiplication algorithm for input matrices of size $n \times n$ whose entries are drawn from the ring $\mathcal{R}$, with $n \geq 2\sqrt{M}$. Let $\mathcal{Y}$ (resp., $\mathcal{Z}$) denote the set of input (resp., output) vertices of the $\left(n/2\sqrt{M}\right)^{\log_2 7}$ sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$. Suppose furthermore that γ pebbles are placed on* internal *vertices (i.e., not input nor output vertices) of any of any of the sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$. For any subset $Z \subseteq \mathcal{Z}$ such that $|Z| \geq 2\gamma$ there exists a set $Y \subset \mathcal{Y}$ with $|Y| \geq 4\sqrt{M\left(|Z|-2\gamma\right)}$ such that each vertex in Y is connected to at least a vertex in Z by a pebble-free path. Let $\mathcal{X}$ denotes the set on input vertices of $H^{n\times n}$. There exists a subset $X \subseteq \mathcal{X}$, with $|X| = |Y|$ such that vertices in Y can be connected to vertices in X through vertex disjoint paths.*

Proof The proof is by induction on the size of the input matrices being multiplied.

Base: In the base case we have $n = 2\sqrt{M}$. We therefore have $H^{n\times n} = H^{2\sqrt{M}\times 2\sqrt{M}}$ and the sets $\mathcal{Y}$ and $\mathcal{X}$ coincide. For any possible placement of the γ pebbles on internal vertices of $H^{n\times n}$, let $X_0 \subseteq \mathcal{X}$ denote the set of input of $H^{n\times n}$ which are not connected to the outputs vertices in Z by any pebble free path. The set of vertices carrying the γ pebbles is therefore a post-dominator for X_0 with respect to Z. From Theorem 4.14 we have that any post-dominator for X_0 with respect to Z must have size at least $w_{2\sqrt{M}\times 2\sqrt{M}}(|X_0|, |Z|)$. Since such post-dominator is constituted by the γ pebbled vertices, the following condition must hold:

$$w_{2\sqrt{M}\times 2\sqrt{M}} \leq \gamma$$

From Lemma 4.3 we have:

$$\frac{1}{2}\left(|Z| - \frac{(8M - |X_0|)^2}{16M}\right) \leq \gamma$$

Let $X_1 = \mathcal{X} \setminus X_0$ denote the set of input values which are connected by pebble free paths to vertices in Z. Since the input size is $\mathcal{X} = 2\times 4M$, we have $|X_1| = 8M - |X_0|$. We therefore have:

$$\frac{1}{2}\left(|Z| - \frac{|X_1|^2}{16M}\right) \leq \gamma$$
$$|X_1| \geq 4\sqrt{M\left(|Z| - 2\gamma\right)}$$

This concludes the proof for the base case.

Inductive step: Let us now assume that the statement is verified for $H^{n\times n}$, with $n \geq 2\sqrt{M}$. We shall show that the statement is verified for $H^{2n\times 2n}$ as well. Let $H_1^{n\times n}, H_2^{n\times n}, \ldots, H_7^{n\times n}$ denote the seven sub-CDAGs of $H^{2n\times 2n}$, each corresponding to the seven sub-products generated by Strassen's algorithm. Let $\mathcal{Z}_i$ (resp., $\mathcal{Y}_i$) denote the subset of $\mathcal{Z}$ (resp., $\mathcal{Y}$) which correspond to vertices in $H_i^{n\times n}$. Note that, according to the structure of Strassen's algorithm, the subsets $\mathcal{Z}_1, \mathcal{Z}_2, \ldots, \mathcal{Z}_7$ (resp., $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_7$) are a partition of $\mathcal{Z}$ (resp., Y). Recall that the seven sub-CDAGs $H_i^{n\times n}$ (and thus, all the sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$) are vertex disjoint among themselves. This implies $\sum_{i=1}^{7} \gamma_i = \gamma$. Let $\delta_i = \max\{0, |Z^i| - 2\gamma_i$, we have $\delta = \sum_{i=1}^{7} \delta_i \geq |Z| - 2\gamma$.

Applying the inductive hypothesis to each sub-CDAG $H_i^{n\times n}$, we have that there is a subset $Y_i \subseteq \mathcal{Y}_i$ with $|Y_i| \geq 4\sqrt{M\delta_i}$ such that vertices of Y_i are connected via to vertices in Z_i via pebble free paths. Furthermore each of these paths can be extended to a subset K_i of the input vertices of $H^{n\times n}$ with $|K_i| = |Y_i|$, such that all vertices in Y_i can be connected to vertices in K_i using vertex-disjoint paths. Since the sub-CDAGs $H_i^{n\times n}$ are vertex disjoint, so are the paths connecting vertices in Y_i to vertices in K_i.

In order to conclude our proof we need to show that is possible to extend at least $4\sqrt{M\left(|Z|-2\gamma\right)}$ of these paths to vertices in $\mathcal{X}$ while still being vertex disjoint. As described in Section 4.2.2, vertices in $K_1, K_2, \ldots, K_7$ are connected to vertices in $\mathcal{X}$ by means of $n^2/2$ encoding sub-CDAGs *Enc*. None of these encoding sub-CDAGs share any input or output vertex. For any given encoder sub-CDAGs each of its output vertices belongs to a different sub-CDAG $H_i^{n\times n}$. This fact ensures that for a single sub-CDAG $H_i^{n\times n}$ it is possible to connect all the vertices in K_i to a subset of the vertices in $\mathcal{X}$ via vertex disjoint paths.

For each of the $n^2/2$ encoder sub-CDAG, let us consider the vector $\mathbf{y}_j \in \{0,1\}^7$ associated with the j-th encoder sub-CDAG. We have that $\mathbf{y}_j[i] = 1$ if the corresponding i-th output vertex (according to the numbering indicated in Figure 4.6) is in K_i or $\mathbf{y}_j[i] = 0$ otherwise. That is, $\mathbf{y}_j[i] = 1$ iff there is a pebble free path connecting the i-th output of the j-th encoder sub-CDAG to a vertex in Z. From Lemma 4.15, we have that there exists a subset $X_j \in \mathcal{X}$ of the input vertices of the j-th encoder sub-CDAG, with $|X_j| \geq g(|\mathbf{y}_j|)$, for which is possible to connect each vertex in X_j to a distinct output verticex of the j-th encoder sub-CDAG using vertex disjoint paths, each constituted by a singular edge. From Lemma 4.15 we have:

$\lvert\mathbf{y}\rvert$	0	1	2	3	4	5	6	7
$g(\lvert\mathbf{y}\rvert)$	0	1	2	2	3	3	4	4

The number of vertex disjoint paths connecting vertices in $\cup_{i=1}^{7} K_i$, to vertices in $\mathcal{X}$ is therefore at least $\sum_{j=1}^{2n^2} g(|\mathbf{y}_j|)$, under the constraint that $\sum_{j=1}^{2n^2} \mathbf{y}_j[i] = 4\sqrt{M\delta^i}$.

Without loss of generality, let us assume that $\delta_1 \geq \delta_2 \geq \ldots \geq \delta_7$. As previously stated, it is possible to connect all vertices in K_1 to vertices in $\mathcal{X}$ through vertex disjoint paths. Consider now all possible dispositions of the vertices in $\cup_{i=2}^{7} K_i$ over the outputs of $n^2/2$ encoders. Recall that the output vertices of an encoder sub-CDAG belong each to a different $H^{n\times n}$ sub-CDAG. From Lemma 4.15, we have that for each encoder, there exists a subset $X_j \subset X$ of the input vertices of the j-th encoder sub-CDAG, with

$$|X_j| \geq g(|\mathbf{y}_j|) \geq \mathbf{y}_j[1] + \lceil \frac{\sum_{i=2}^{7} \mathbf{y}_j[i]}{2} \rceil \geq \mathbf{y}_j[1] + \frac{\sum_{i=2}^{7} \mathbf{y}_j[i]}{2},$$

for which is possible to connect all vertices in X_j to the $|\mathbf{y}_j|$ output vertices of the j-th encoder sub-CDAG which are in $\cup_{i=1}^{7} K_i$ using $|X_j|$ vertex disjoint paths. As all the *Enc* sub-CDAGs are vertex disjoint, we can sum their contributions and we can therefore conclude that the number of vertex disjoint paths connecting values in $\mathcal{X}$ to vertices in $\cup_{i=1}^{7} K_i$ is at least:

$$|K_1| + \frac{1}{2}\sum_{i=2}^{7} |K_i| = 4\sqrt{M}\left(\sqrt{\delta_1} + \frac{1}{2}\sum_{i=2}^{7}\sqrt{\delta_i}\right) \tag{4.17}$$

Squaring this quantity leads to:

$$\left(4\sqrt{M}\left(\sqrt{\delta_1} + \frac{1}{2}\sum_{i=2}^{7}\sqrt{\delta_i}\right)\right)^2 = 16M\left(\delta_1 + \sqrt{\delta_1}\sum_{i=2}^{7}\sqrt{\delta_i} + \left(\frac{1}{2}\sum_{i=2}^{7}\sqrt{\delta_i}\right)^2\right)$$

As by assumption, $\delta_1 \geq \delta_2 \geq \ldots \geq \delta_7$, we have that $\sqrt{\delta_1}\sqrt{\delta_i} \geq \delta_1$, for $i = 2, 3, \ldots, 7$. We thus have:

$$\begin{aligned}\left(4\sqrt{M}\left(\sqrt{\delta_1} + \frac{1}{2}\sum_{i=2}^{7}\sqrt{\delta_i}\right)\right)^2 &\geq 16M\sum_{i=1}^{7}\delta_i \\ &\geq \left(4\sqrt{M\left(|Z - 2\gamma|\right)}\right)^2\end{aligned}$$

There are therefore at least $4\sqrt{M\left(|Z - 2\gamma|\right)}$ vertex disjoint paths connecting vertices in $\mathcal{X}$ to vertices in $\cup_{i=2}^{7} K_i$ (and therefore to vertices in $\cup_{i=2}^{7} Z_i$). The lemma follows. □

Lemma 4.17. *Let $H^{n\times n}$ be the CDAG which corresponds to the execution of Strassen's matrix multiplication algorithm for input matrices of size $n \times n$ whose entries are drawn from the ring $\mathcal{R}$, with $n \geq 2\sqrt{M}$. Let $\mathcal{Y}$ (resp., $\mathcal{Z}$), denote the set of input (resp., output) vertices of the $\left(n/2\sqrt{M}\right)^{\log_2 7}$ sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$. Suppose furthermore that γ pebbles are placed on vertices of $H^{n\times n}$. For any subset $Z \subseteq \mathcal{Z}$ such that $|Z| \geq 2\gamma$ there exists a set $Y \subset \mathcal{Y}$ with $|Y| \geq 4\sqrt{M\left(|Z| - 2\gamma\right)}$ such that each vertex in Y is connected to at least a vertex in Z by a pebble-free path. Let $\mathcal{X}$ denotes the set on input vertices of $H^{n\times n}$. There exists a subset $X \subseteq \mathcal{X}$, with $|X| = |Y|$ such that vertices in Y can be connected to vertices in X through vertex disjoint paths.*

Proof Let $0 \leq \gamma' \leq \gamma$ be the number pebbles be placed on "internal" vertices of any of any of the sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$. The remaining $\gamma - \gamma$ pebbles may be placed anywhere in $H^{n\times n}$. From Lemma 4.16 we have that there exist at least $4\sqrt{M\left(|Z| - 2\gamma'\right)}$ pebble free paths that connecting a subset $X \subseteq \mathcal{X}$ of global input vertices to vertices in Z passing through a subset $Y \subseteq \mathcal{Y}$ and that the sub-paths of such paths that connect vertices in Y to vertices in X are vertex disjoint. One of these paths can be blocked by one of the "*external*" $\gamma-\gamma'$ if one of the vertices in the sub-path connecting X and Y is pebbled. Since said sub-paths are vertex disjoint, each of the $\gamma-\gamma'$ pebbled vertices can block at most one of the paths connecting X to Y (and therefore Z). The number of pebble free paths from X to Z will therefore be at least:

$$4\sqrt{M\left(|Z|-2\gamma'\right)}-\left(\gamma-\gamma'\right)$$

let us raise this quantity to the second power:

$$\begin{aligned}\left(4\sqrt{M\left(|Z|-2\gamma'\right)}-\left(\gamma-\gamma'\right)\right)^2 = \\ &= 16M\left(|Z|-2\gamma'\right)+\left(\gamma-\gamma'\right)^2-8\sqrt{M\left(|Z|-2\gamma'\right)}\left(\gamma-\gamma'\right)\\ &\geq 16M\left(|Z|-2\gamma'\right)-8\sqrt{M\left(|Z|-2\gamma'\right)}\left(\gamma-\gamma'\right)\\ &\geq 16M\left(|Z|-2\gamma\right)+32M\left(\gamma-\gamma'\right)-8\sqrt{M\left(|Z|-2\gamma'\right)}\left(\gamma-\gamma'\right)\\ &\geq 16M\left(|Z|-2\gamma\right)+\left(\gamma-\gamma'\right)\left(32M-8\sqrt{M\left(|Z|-2\gamma'\right)}\right)\end{aligned}$$

Since, by hypothesis $|Z|-2\gamma'\leq 4M$ we have:

$$\left(4\sqrt{M\left(|Z|-2\gamma'\right)}-\left(\gamma-\gamma'\right)\right)^2\geq 16M\left(|Z|-2\gamma\right)$$

and we can thus conclude:

$$4\sqrt{M\left(|Z|-2\gamma'\right)}\geq 4\sqrt{M\left(|Z|-2\gamma\right)}$$

The lemma follows. □

Corollary 4.18. *Any subset $Z\subseteq\mathcal{Z}$, with $|Z|=4M$ has a dominator set of size at least $|Z|/2=2M$.*

Lemma 4.17 and Corollary 4.18 provide us the tools required to complete the proof of the main Theorem 4.11.

4.6.2 Proof of the main theorem

Proof of Theorem 4.11 In the following proof we assume without loss of generality that $n=2^a$ and $\sqrt{M}=2^b$ for some $a,b\in\mathbb{N}$. At least $3M$ I/O operations are necessary in order to read all the $2n^2$ input values form slow memory to the cache and to write the n^2 output values to the slow memory once they have been computed. The statement of the theorem is therefore trivially verified if $n\leq 2\sqrt{M}$.

In the following, we will consider the case for $n \geq 4\sqrt{M}$ Let $\mathcal{Y}$ (resp., $\mathcal{Z}$), denote the set of input (resp., output) vertices of the $\left(n/2\sqrt{M}\right)^{\log_2 7}$ sub-CDAGs $H^{2\sqrt{M}\times 2\sqrt{M}}$ of $H^{n\times n}$. Let $\mathcal{P}$ any pebbling strategy for the Red-Blue pebble game played on $H^{n\times n}$ using M red pebbles. We partition $\mathcal{P}$ into segments according to the following criteria:

- the first segment σ_0 is the shortest prefix of $\mathcal{P}$ during which $4M$ distinct values of $\mathcal{Z}$ are pebbled for the first time.
- for $i > 1$, let $\mathcal{P}_i$ denote the suffix of $\mathcal{P}$ which starts after the end of the $i-1$-th segment. The i-th segment, is the shortest prefix of $\mathcal{P}_i$ during which $4M$ distinct values of $\mathcal{Z}$ are evaluated for the first time. We denote the set of vertices corresponding to these $4M$ values as Z_i

According to our previous considerations we have $|\mathcal{Z}| = 4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ and there will thus be a total of $\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ segments.

We shall now verify that at least M I/O operations are to be executed in every segment σ_i of the overall pebbling strategy $\mathcal{P}$. The proof is by contradiction. Let Γ_i be the set of vertices which are either carrying a red pebble at the beginning of σ_i, or receive a red pebble during σ_i by means of a *read* from the slow memory. At the beginning of any segment σ_i at most M vertices can carry a red pebble. Suppose during each interval at most $M-1$ vertices receive a red pebble though a *read* from secondary memory (R5). This implies $|Gamma| \leq 2M-1$. In order for the $4M$ values from Z_i to be computed during the segment without any additional I/O operation, there must be no path connecting any vertex in $\mathcal{Z}$ to any input vertex of $H^{n\times n}$ which does not have at least one vertex in Γ. According to the terminology in [37], this is equivalent to saying that Γ has to be a *dominator set* of Z_i. From Corollary 4.18, we have that any sub-set of $4M$ elements of $\mathcal{Z}$ has dominator size at least $2M$. This leads to a contradiction. At least M I/O operations are thus executed during each segment σ_i. Since, by construction, the $\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ segments are not overlapping, we can therefore conclude that at least $M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ are necessary for the execution of any pebbling strategy $\mathcal{P}$ for the Strassen's algorithm. □

Our lower bound to the I/O complexity of Strassen's matrix multiplication algorithm corresponds asymptotically to the ones proposed in [7, 62] for recomputation nr-computations. In [3] Ballard et al. presented a version of Strassen's algorithm whose I/O cost matches, up to a constant, the one indicate by our bound, which is therefore tight. Furthermore, our result implies that the version of Strassen's algorithm presented by Ballard et al. in [3], is indeed optimal even without any restriction on multiple evaluations of some intermediate result as its asymptotical I/O cost matches our lower bound. As in the optimal algorithm presented in [3] no intermediate result is ever recomputed, we can conclude the use of recomputation can lead at most to a constant factor reduction of the I/O complexity for the execution of Strassen's matrix multiplication algorithm.

Constant term in the lower bound

We will now discuss some extension of the previous results similar to those we presented for the result in Section 4.5.

Our proof technique for Theorem 4.11 could be modified in order to focus our analysis on sub-problems of size $\alpha\sqrt{M} \times \alpha\sqrt{M}$, for $\alpha \leq n/\sqrt{M}$, instead of sub-problems $2\sqrt{M} \times 2\sqrt{M}$. The form of the obtained lower bound would be:

$$IO_{H^{n\times n}}(M) \geq \frac{\alpha^2 - 2}{2\alpha^{\log_2 7}} \left(\frac{n}{\sqrt{M}}\right)^{\log_2 7} M. \tag{4.18}$$

Under the constraint that $\alpha\sqrt{M}$ is a power of two, the choice of $\alpha = 2$ maximizes the constant term of equation 4.18.

On mixed matrix multiplication strategies

In our technique, no constraint is imposed on the specific algorithm being used to compute the results of the $\alpha\sqrt{M} \times \alpha\sqrt{M}$ sub-products. This allows to generalize our result to mixed algorithmic strategies in which Strassen's algorithm is initially used to generate sub-problems of smaller size (e.g., sub-problems which can be entirely computed in cache) which are then computed using a different algorithm (e.g. the naive algorithm) can still be captured by our technique.

Lemma 4.19. *Consider a mixed matrix multiplication strategy for which Strassen's algorithm is initially used to reduce the input matrices of size $n \times n$, whose entries are drawn from the ring $\mathcal{R}$, by generating $\left(n/2^i\sqrt{M}\right)^{\log_2 7}$ sub-products of size $2^i\sqrt{M} \times 2^i\sqrt{M}$ for $0 \leq i \leq \log_2(n/2\sqrt{M})$. The single sub-product are then computed using any straight line algorithm for matrix multiplication. Let $H^{n\times n,i}$ denote the CDAG corresponding to this mixed strategy. The I/O complexity any such strategy when run on a sequential machine with fast memory of size M is:*

$$IO_{H^{n\times n,i}}(M) \geq \left(\frac{n}{2^i\sqrt{M}}\right)^{\log_2 7} \frac{M}{4}\left(2^i - 1\right) \tag{4.19}$$

The proof of this lemma is given by a simple extension of the proof of Theorem 4.11 by focusing the analysis on the sub-problems with input size $2^i\sqrt{M}\times 2^i\sqrt{M}$. Although this bound is in general much more loose than the one provided by theorem 4.8, it provides an interesting indication of the I/O complexity for Strassen-mixed strategies.

4.7 Lower bound to the I/O complexity of Strassen's algorithm in the parallel model

As suggested in the introduction of this chapter, the I/O complexity of an algorithm can also be studied with respect to its parallel execution on multiple processors. For parallel computations we consider a model with P processors each equipped with a local memory of size M, all P processors are connected by means of a network. We assume that the input is initially distributed among all processors, so MP has to be at least as large as the input, we do not however assume anything about *how* the input is distributed among the P processors. In this context, we therefore have that the *read* and *write* operations, which model the communication between fast and slow memory, are replaced by communications (respectively, incoming and outgoing) between processors.

The I/O cost of an algorithm in this model (also referred as *bandwidth cost* in literature) will therefore be given by the number of messages (i.e., words or values) communicated between processors along the *critical path* as defined in [71], that is two values that are communicated simultaneously are counted only once. This metric is closely related to the total running time of the algorithm.

Theorem 4.20. *Consider Strassen's matrix multiplication algorithm being used to multiply two square matrices of size $n\times n$ whose entries are drawn from the ring $\mathcal{R}$. The I/O-complexity of Strassen's algorithm when run on a parallel machine with Ps processors each equipped with a local memory of size M is:*

$$IO_{H^{n\times n}}(M) = \Omega\left(\left(\frac{n}{\sqrt{M}}\right)^{\log_2 7}\frac{M}{P}\right) \tag{4.20}$$

Proof Let $\mathcal{Z}$ denote the set of the $4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ output vertices of the $H^{2\sqrt{M}\times 2\sqrt{M}}$ sub-CDAGs. Vertices in $\mathcal{Z}$ represent the output values of the corresponding sub-problems. All the values corresponding to the vertices in $\mathcal{Z}$ are computed at least once during the execution of the algorithm. Among the P processors, at least one shall compute at least $\frac{1}{P}4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ of these values. Let us assume, without loss of generality, that P_1 is such processor. We denote as $\mathcal{Z}_1$ the subset of values in $\mathcal{Z}$ computed by P_1. We shall now focus on the computation $\mathcal{C}_1$ executed by P_1. We partition $\mathcal{C}_1$ into segments according to the following criteria such that during each segment $4M$ distinct values from $\mathcal{Z}_1$ are computed for the first time. As $|\mathcal{Z}_1| \geq \frac{1}{P}4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ there will be at least $\frac{1}{P}4M\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ We shall now verify that at least M I/O operations are to be executed during every segment σ_i of $\mathcal{C}_\infty$. The proof is by contradiction. At the beginning of any segment σ_i, at most M values each corresponding to a vertex of $H^{n\times n}$ can be maintained in the memory of P_1. Suppose during each interval P_1 acquires at most $M-1$ through communications with other processors. In order for the $4M$ values from $\mathcal{Z}$ to be computed during the segment without any additional I/O operation, there must be no path connecting any vertex in $\mathcal{Z}_1$ to any input vertex of $H^{n\times n}$. That is, the vertices corresponding set of the at most $2M-1$ values either present in memory at the beginning of the segment or acquired via communications with other processors must be a dominator set of the vertices corresponding to the $4M$ values from $\mathcal{Z}$ computed during the segment. From Corollary 4.18, we have that any sub-set of $4M$ elements of $\mathcal{Z}$ has dominator size at least $2M$. This leads to a contradiction. At least M I/O operations are thus executed during each segment σ_i. Since, by construction, the $\frac{1}{P}\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ segments are not overlapping, we can therefore conclude that at least $\frac{M}{P}\left(\frac{n}{2\sqrt{M}}\right)^{\log 7}$ communications are necessary for the execution of any computation of Strassen's algorithm. $\square$

Our lower bound to the I/O complexity of Strassen's matrix multiplication algorithm corresponds asymptotically to the ones proposed in [7, 62] which have been obtained under the assumption that no intermediate value is computed more than once (no recomputation assumption). Our results does not require any restriction on the recomputation of intermediate values, nor any assumption on the distribution of the input data among the P processors at the beginning of the algorithm's execution.

The same extension can be obtained using the same argument for the results on mixed multiplication algorithms (Lemma 4.19).

4.8 Conclusion

In this chapter, we studied the I/O complexity of Strassen's algorithm when executed sequentially on a machine equipped with a two level memory hierarchy. We use an alternative technique to those in [4] and [62] to obtain a tight lower bound to the

I/O complexity of Strassen's matrix multiplication algorithm for computations in which no intermediate result is ever recomputed. Our proof technique relates the I/O complexity to the recursive structure and the information flow of the function implemented by the algorithm.

In the main contribution of the chapter we obtained the first asymptotically tight lower bound to the I/O complexity of Strassen's algorithm for general computations i.e., computations without any restriction on the recomputation of intermediate values. Our technique is based on a novel application of Gigoriev's *information flow* concept [33], which used to determinate a lower bound for the size of a dominator set of subset of the Strassen's algorithm CDAG. As an implementation of Strassen's matrix multiplication algorithm whose I/O complexity matches (asymptotically) our lower bound while never computing an intermediate value more than once has been presented in literature [3], this allows us to conclude that our bound is tight, and that the use of recomputation in the execution of Strassen's matrix multiplication algorithm can lead at most to a constant factor reduction of the I/O complexity.

We also studied the I/O complexity of Strassen's algorithm when executed in parallel by P processors each equipped with a finite memory. We obtain an lower bound which holds for any computation (no restriction on recomputation), without any assumption regarding the distribution of the input data among the P processors at the beginning of the computation.

Chapter 5

Algorithms resilient to memory faults

In this chapter we briefly presents some results regarding the effect of opportune memory utilization in the context of *error resilient algorithms*, which provide (almost) correct solutions even when silent memory errors occur.

A complete presentation of the results mentioned in this chapter, including the details on the mentioned algorithms and the proofs of the theoretical results, can be found in [22]. This paper was co-authored with Francesco Silvestri.

Our analysis spawns from the observation that memories of modern computational platforms are not completely reliable. As documented in several practical applications [8, 61], various causes, such as cosmic radiations and alpha particles [8], may lead to a transient failure of a memory unit and to a consequent loss or corruption of its content. Such memory errors are usually not detectable by the system (i.e., they are "*silent*"). An application may therefore successfully terminate even if the final output is irreversibly corrupted due to the presence of errors.

Although hardware-level countermeasures, such as Error Correcting Codes (ECC), can be used to prevent problems originating from said memory corruptions, these are often costly and sensibly reduce space and time performance. Algorithmic (software-level) approaches for dealing with unreliable memory are thus very attractive and have therefore received considerable attention in literature under different settings, we refer to [25] for a survey. Algorithms and data structures which maintain an "*acceptable level of functionality*", tolerating the occurrence of silent memory errors and corruptions, are called *resilient*. Note that what is to be considered an acceptable level of functionality is in general dependent on the specific problem (sorting, searching, FFT,...) being considered. While for some problems such as sorting, a definition of acceptable level of functionality is quite straightforward (i.e., a limited number of un-ordered elements in the output is tolerable), the definition can be much more challenging for other problems such as matrix multiplication and FFT [54], or graph algorithms.

The *Faulty RAM* (*FRAM*) model, introduced by Finocchi and Italiano in [28], has received considerable attention in literature. In this model, an adaptive adversary can corrupt up to δ memory cells of a large unreliable memory at any time (even simultaneously) during the execution of an algorithm. The same memory location can be affected by multiple corruptions trough the execution of the algorithm.

Various algorithms and data structures have been designed in this designed in this model for many problems, including sorting [26], selection [39], dynamic programming [13], dictionaries [27], priority queues [38], matrix multiplication and FFT [54], K-d and suffix trees [31, 15]. The practical validity of this model has also been experimentally evaluated [51, 54, 50, 49].

5.1 Our contribution

Let δ and α denote respectively the maximum amount of faults which can happen during the execution of an algorithm and the actual number of occurred faults, with $\alpha \leq \delta$. Previous results in the FRAM model assume the existence of a *safe memory* of constant size which cannot be corrupted by the adversary and which is used for storing crucial data such as code and instruction counters. Following up the preliminary investigation in [13], we enrich the FRAM model with a safe memory of arbitrary size S and then give evidence that an increased safe memory can be exploited to improve the performance of resilient algorithms.

In particular, we present the S-Sort algorithm, which can be used to resiliently sort n entries in $\mathcal{O}(n \log n + \alpha(\delta/S + \log S))$ time when a safe memory of size $\Theta(S)$ is available in the FRAM.

Finally, we use the proposed resilient sorting algorithm for constructing a resilient priority queue data structure. Our implementation uses $\Theta(S)$ safe memory words and $\Theta(n)$ faulty memory words for storing n keys, and requires $\mathcal{O}(\log n + \delta/S)$ amortized time for each insert and minimum element removal operation *Deletemin.*

In addition to its theoretical interest, the adoption of such a model is supported by recent research on hybrid systems that combine algorithmic-level resiliency with the use of a limited amount memory protected at hardware-level using ECC [41]. In this setting, S would denote the memory that is protected by the hardware.

5.2 The extended FRAM model

Our extended FRAM model features two memories: the *faulty memory* whose size is potentially unbounded, and the *safe memory* of size S. For the sake of simplicity, we allow algorithms to exceed the amount of safe memory by a multiplicative constant factor. At any time the adversary can read the content of any memory location in the faulty memory and can corrupt the value stored in any such location up to δ times. Said corruptions (or faults) can occur simultaneously and the adversary is allowed to corrupt a value which was already previously altered. The adversary can read any memory location of the safe memory as well[1] but he cannot alter (corrupt) the values stored in such memory locations. We denote with $\alpha \leq \delta$ the actual number of faults (corruptions) occurred during the execution of the algorithm.

5.3 Resilient sorting algorithm

We say that a value is *faithful* if it has never been corrupted and that a sequence is *faithfully ordered* if all the faithful values in it are correctly ordered. In the resilient sorting problem we are given a set of n keys and the goal is to correctly order all the faithful input keys (corrupted keys can be arbitrarily positioned).

We propose S-*Sort*, a resilient sorting algorithm which uses$\Theta(S)$ safe memory words and runs in time $\mathcal{O}(n \log n + \alpha(\delta/S + \log S))$. While our approach is indeed inspired by the resilient sorting algorithm in [26], several major modifications are required in order to fully exploit the safe memory.

In particular, S-Sort forces the adversary to inject $\Theta(S)$ faults in order to invalidate part of the computation and to increase the running time by an additive $\mathcal{O}(\delta + S \log S)$ term. In comparison, just $\mathcal{O}(1)$ faults suffice to induce an additive overhead term $\mathcal{O}(\delta)$ to the execution time of previous algorithms [28, 26, 39], even if a safe memory of size $\omega(1)$ is available.

We therefore have that our algorithm runs in optimal $\Theta(n \log n)$ time for $\delta = \mathcal{O}\left(\sqrt{Sn \log n}\right)$ and $S \leq n/\log n$: this represents a $\Theta\left(\sqrt{S}\right)$ improvement with respect to the state of the art [26], where optimality is reached for $\delta = \mathcal{O}\left(\sqrt{n \log n}\right)$.

As S-Sort is based on *mergesort* [21], we also introduce a resilient merging algorithm S-Merge, which exploits the available safe memory. S-Merge runs in $\mathcal{O}(n + \alpha(\delta/S + \log S))$ time using $\Theta(S)$ safe memory. The algorithm merges two input faithfully ordered sequences of length n each with $\Theta(S)$ into one unique faithfully ordered output sequence.

[1]This constitutes the main difference with respect to a similar model adopted in [13], for which the adversary is not allowed to read the safe memory

We here provide a high-level description of the S-Merge algorithm. An incomplete merge of the two input sequences is initially computed with S-PurifyingMerge: this algorithm returns a faithfully ordered sequence Z of length at least $2(n-\alpha)$ that contains a partial merge of the values of the input sequences that S-PurifyingMerge has determined to be faithful, and a sequence F with the at most 2α remaining input entries that S-PurifyingMerge has deemed to potentially be corrupted values and has failed to insert *faithfully* into Z. The algorithm S-PurifyingMerge runs in $\mathcal{O}(n + \alpha\delta/S)$ time.

The values in F are then inserted into Z using the S-*BucketSort* algorithm, which runs in $\mathcal{O}(n + \alpha(\delta/S + \log S))$ time, thus obtaining the final faithfully ordered merge of all input values.

S-Sort is then obtained by using S-Merge in the classical mergesort algorithm[2] [21].

A complete description and analysis of these results, including the detailed presentation and analysis of S-PurifyingMerge and S-BucketSort, is available in our published paper [22].

It is important to point out that the $\Omega(n \log n + \alpha\delta)$ lower bound in [28] on the performance of resilient comparison-based sorting and merging algorithms does not apply to S-Sort and S-Merge as the lower bound does not account for the presence of any safe memory location.

For the complete description of the mentioned algorithms, and for the detailed analysis of their performance, please refer to our published paper [22].

5.4 Resilient priority queue

A *resilient priority queue* is a data structure which maintains a set of keys that can be managed and accessed through two main operations: Insert, which allows to add a key to the queue; and Deletemin, which returns the minimum faithful key among those in the priority queue or an even smaller corrupted key and then removes it from the priority queue.

In our work [22], we constructed an implementation of the resilient priority queue that exploits a safe memory of size $\Theta(S)$. Let n denote the number of keys in the queue. Our implementation requires $\mathcal{O}(\log n + \delta/S)$ amortized time per operation, a safe memory of size $\Theta(S)$ and $\Theta(n)$ words in the faulty memory. Our resilient priority queue is based on the fault tolerant priority queue proposed in [38], which uses various elements of the cache-oblivious priority queue in [2].

[2]The standard recursive mergesort algorithm requires a stack of length $\mathcal{O}(\log n)$ which cannot be corrupted. It is however easy to derive an iterative algorithm where a $\Theta(1)$ stack length suffices.

The performance of the state-of-the-art implementation of resilient priority queue proposed in [38] is here improved by exploiting the safe memory and by using the S-Merge and S-Sort algorithms previously discussed. The $\Omega(\log n + \delta)$ lower bound in [38] on the performance of the resilient priority queue does not apply to our data structure as the lower bound argument assumes that keys are not stored in safe memory between operations.

The amortized time for each operation in our implementation matches the performance of classical optimal priority queues in the RAM model if the maximum number of tolerated corruptions is $\delta = \mathcal{O}(S \log n)$: we therefore obtain a $\Theta(S)$ improvement with respect to the state of the art [38].

For the complete description of the structure of our priority queue, and for the detailed analysis of its performance, please refer to our published paper [22].

5.5 Conclusion

In this chapter we considered how limitations on the size of the available safe memory can affect the performance of resilient algorithms and stata structured when silent memory errors and corruptions may occur. We provide an overview of the main results presented in [22] among which a novel resilient sorting algorithm S-Sort, and a novel implementation od the resilient priority queue data. These results achieve an improvement over the respective state of the art by exploiting a safe memory of size $\Theta(S)$. As future research, it would be interesting to investigate which other problems could benefit from a non-constant safe memory, and then obtain tradeoffs between the achievable performance and the size of the available safe memory.

Note in fact that the use of an S-size safe memory is not guaranteed to improve the performance of resilient algorithms for any problem. For instance, the $\Omega(\log n + \delta)$ lower bound for searching derived in [28] applies even if a safe memory of size $S \leq \epsilon n$, for a suitable constant $\epsilon \in (0, 1)$, is available [22].

Chapter 6

Conclusion and future work

In this thesis we studied various aspects related to the computation of straight-line programs for which limitations in the available memory space play a crucial role.

In Chapter 2 and Chapter 3 we studied the memory space requirements for straight-line programs represented by means of a Computational Directed Acyclic Graph $G(I \cup V, E)$, making use of the pebble game abstraction. In Chapter 2, we studied techniques for obtaining lower bounds to the memory space required for the computation of a CDAG algorithm (i.e., its space complexity). We reviewed the *marking rule* approach [10] and applied it in order to obtain a lower bound to the space complexity of Superconcentrator-Stack CDAGs. In order to study the limits of the *marking rule* technique, we introduced the concept of *visit* of a CDAG and we showed how studying properties of the visits of a CDAG is related to its space complexity. In our results we showed that for both the *singleton* and *topological* it is possible to construct a visit that requires space $\mathcal{O}\left(\sqrt{d^{-}n}\right)$, where $|I \cup V| = n$ and d^{-} is the maximum in-degree of G. An important open question with regards to this topic, is whether a similar result holds for *intermediate* (i.e., not singleton not topologic) visit rules. If this was indeed not the case for a generic CDAG, it would be interesting to determinate whether this is the case for some particular families of CDAGs. As future work, we aim to study whether the marking rule technique and the visit approach could be applied to the main variants of the basic pebble game know in literature such as the Black-White pebble game introduced by Cook and Sethi [18] or the two-people pebble game introduced by Tompa [65].

In Chapter 3 we moved to the analysis of an upper bound to the pebbling cost of a generic CDAG with m edges and maximum in-degree d^{-}. We proposed an algorithm that allows to construct a complete pebbling strategy that requires $\mathcal{O}\left(m/\log m + d^{-}\right)$ pebbles. While a family of CDAGs whose complexity matches the upper bound when the maximum in-degree of a CDAG is a constant with respect to the size of the CDAG has been studied in literature [48] (a CDAG in this family is represented in Figure 2.5), it would be interesting to investigate whether the same condition holds for non-constant maximum in-degree. The open question of whether it is indeed possible to construct a family of n-vertex CDAGs attaining space complexity $\Theta(n)$ with maximum in-degree $o(n)$, and if so, which is the minimum value of d^- for which this is possible, is of high theoretical interest.

In Chapter 4 we studied the I/O complexity of CDAG computations in the hierarchical memory setting as modeled by Hong and Jung [37]. In particular, we focused on Strassen's matrix multiplication algorithm [63].

We provide an alternative technique to those in [4] and [62] to obtain a tight lower bound to the I/O complexity of Strassen's matrix multiplication algorithm for computations in which no intermediate result is ever recomputed. Our proof technique relates the I/O complexity to the recursive structure of the algorithm, rather than to specific combinatorial properties of the corresponding CDAG as done in previous contributions [7, 62].

We then obtain the first tight lower bound to the I/O complexity of Strassen's matrix multiplication algorithm which does not require any restriction on the recomputation of intermediate values. In our technique, we use elements from Grigoriev's information flow method [33]. Because of the fact that an algorithm whose performance match this bound without recomputing any intermediate value is known in literature [3], we conclude that the use of recomputation does not allow to reduce the I/O cost of Strassen's algorithm for more than a constant factor. In our future work we aim to verify whether these techniques may be applied to variants of Strassen's algorithm (the so called *Strassen-like* algorithms). The study of the characteristics of a straight-line program, or of the CDAG corresponding to it, which may indicate whether recomputation may or may not allow for a reduction of the I/O cost, is an open problem of great theoretical importance. As future research, we aim to study whether the information flow concept may shed some insight into this problem.

For what pertains algorithms and data structures resilient to memory faults, as future research, it would be interesting to investigate which other problems can benefit of a non-constant safe memory and propose tradeoffs between the achievable performance and the size of the available safe memory. Such tradeoffs may also provide useful insights for designing hybrid systems mounting both cheap faulty memory and expensive ECC memory. Furthermore, it would be interesting to study whether the use of the safe memory S could allow for designing resilient algorithms which achieve efficient performance without explicit knowledge of the value of the parameter δ (i.e., δ-oblivious).

Bibliography

[1] Aggarwal, A., Alpern, B., Chandra, A., and Snir, M. (1987). A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314. ACM.

[2] Arge, L., Bender, M. A., Demaine, E. D., Minkley, B. H., and Munro, J. I. (2007). An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal Computing*, 36(6):1672–1695.

[3] Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., and Schwartz, O. (2012a). Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 193–204. ACM.

[4] Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., and Schwartz, O. (2012b). Graph expansion analysis for communication costs of fast rectangular matrix multiplication. In *Design and Analysis of Algorithms*, pages 13–36. Springer.

[5] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. (2010). Communication-optimal parallel and sequential cholesky decomposition. *SIAM Journal on Scientific Computing*, 32(6):3495–3523.

[6] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. (2011). Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901.

[7] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. (2012c). Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)*, 59(6):32.

[8] Baumann, R. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. Devive and Materials Reliability*, 5(3):305–316.

[9] Bilardi, G. and Peserico, E. (2001). A characterization of temporal locality and its portability across memory hierarchies. In *Automata, Languages and Programming*, pages 128–139. Springer.

[10] Bilardi, G., Pietracaprina, A., and DAlberto, P. (2000). On the space and access complexity of computation dags. In *Graph-Theoretic Concepts in Computer Science*, pages 47–58. Springer.

[11] Bilardi, G. and Preparata, F. P. (1999). Processortime tradeoffs under bounded-speed message propagation: Part ii, lower bounds. *Theory of Computing Systems*, 32(5):531–559.

[12] Bini, D., Capovani, M., Romani, F., and Lotti, G. (1979). O (n 2.7799) complexity for n× n approximate matrix multiplication. *Information processing letters*, 8(5):234–235.

[13] Caminiti, S., Finocchi, I., Fusco, E. G., and Silvestri, F. (2011). Dynamic programming in faulty memory hierarchies (cache-obliviously). In *Proc. 31st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 13 of *LIPIcs*, pages 433–444.

[14] Cannon, L. E. (1969). A cellular computer to implement the kalman filter algorithm. Technical report, DTIC Document.

[15] Christiano, P., Demaine, E., and Kishore, S. (2011). Lossless fault-tolerant data structures with additive overhead. In *Proc. 12th Workshop on Algorithms and Data Structures (WADS)*, volume 6844 of *LNCS*, pages 243–254.

[16] Cohn, H., Kleinberg, R., Szegedy, B., and Umans, C. (2005). Group-theoretic algorithms for matrix multiplication. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 379–388. IEEE.

[17] Cohn, H. and Umans, C. (2003). A group-theoretic approach to fast matrix multiplication. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 438–449. IEEE.

[18] Cook, S. and Sethi, R. (1974). Storage requirements for deterministic/polynomial time recognizable languages. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 33–39. ACM.

[19] Coppersmith, D. and Winograd, S. (1982). On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492.

[20] Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6. ACM.

[21] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3rd edition.

[22] De Stefani, L. and Silvestri, F. (2015). Exploiting non-constant safe memory in resilient algorithms and data structures. *Theor. Comput. Sci.*, 583:86–97.

[23] Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. (2015). On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 567–580. ACM.

[24] Essam, J. W. and Fisher, M. E. (1970). Some basic definitions in graph theory. *Reviews of Modern Physics*, 42(2):271.

[25] Finocchi, I., Grandoni, F., and Italiano, G. F. (2007). Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2):77–87.

[26] Finocchi, I., Grandoni, F., and Italiano, G. F. (2009a). Optimal resilient sorting and searching in the presence of memory faults. *Theoretical Computer Science*, 410(44):4457–4470.

[27] Finocchi, I., Grandoni, F., and Italiano, G. F. (2009b). Resilient dictionaries. *ACM Transactions on Algorithms*, 6(1):1:1–1:19.

[28] Finocchi, I. and Italiano, G. F. (2008). Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332.

[29] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE.

[30] Gabber, O. and Galil, Z. (1981). Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420.

[31] Gieseke, F., Moruz, G., and Vahrenhold, J. (2012). Resilient k-d trees: k-means in space revisited. *Frontiers of Computer Science*, 6(2):166–178.

[32] Gilbert, J. R., Lengauer, T., and Tarjan, R. E. (1980). The pebbling problem is complete in polynomial space. *SIAM Journal on Computing*, 9(3):513–524.

[33] Grigor'ev, D. Y. (1976). Application of separability and independence notions for proving lower bounds of circuit complexity. *Zapiski Nauchnykh Seminarov POMI*, 60:38–48.

[34] Harary, F. (2005). Structural models: An introduction to the theory of directed graphs.

[35] Hopcroft, J., Paul, W., and Valiant, L. (1977). On time versus space. *Journal of the ACM (JACM)*, 24(2):332–337.

[36] Irony, D., Toledo, S., and Tiskin, A. (2004). Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026.

[37] Jia-Wei, H. and Kung, H.-T. (1981). I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM.

[38] Jørgensen, A. G., Moruz, G., and Mølhave, T. (2007). Priority queues resilient to memory faults. In *Proc. 10th Workshop on Algorithms and Data Structures (WADS)*, volume 4619 of *LNCS*, pages 127–138.

[39] Kopelowitz, T. and Talmon, N. (2012). Selection in the presence of memory faults, with applications to in-place resilient sorting. In *Proc. 23rd International Symposium on Algorithms and Computation (ISAAC)*, volume 7676 of *LNCS*, pages 558–567.

[40] Lengauer, T. and Tarjan, R. E. (1979). Upper and lower bounds on time-space tradeoffs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 262–277. ACM.

[41] Li, D., Chen, Z., Wu, P., and Vetter, J. S. (2013). Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[42] Loomis, L. H. and Whitney, H. (1949). An inequality related to the isoperimetric inequality. *Bull. Amer. Math. Soc.*, 55(10):961–962.

[43] Loui, M. C. (1979). *Minimum Register Allocation Is Complete in Polynomial Space*. Massachusetts Institute of Technology, Laboratory for Computer Science.

[44] Miller, W. (1975). Computational complexity and numerical stability. *SIAM Journal on Computing*, 4(2):97–107.

[45] Nordström, J. (2009). New wine into old wineskins: A survey of some pebbling classics with supplemental results. *Manuscript in preparation. Current draft version available at the webpage http://people. csail. mit. edu/jakobn/research.*

[46] Pan, V. Y. (1978). Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *19th Annual Symposium on Foundations of Computer Science*, pages 166–176. IEEE.

[47] Paterson, M. S. and Hewitt, C. E. (1970). Comparative schematology. In *Record of the Project MAC conference on concurrent systems and parallel computation*, pages 119–127. ACM.

[48] Paul, W. J., Tarjan, R. E., and Celoni, J. R. (1976). Space bounds for a game on graphs. *Mathematical Systems Theory*, 10(1):239–251.

[49] Petrillo, U. F., Finocchi, I., and Italiano, G. F. (2010). Experimental study of resilient algorithms and data structures. In *Proc. 9th International Symposium Experimental Algorithms*, pages 1–12.

[50] Petrillo, U. F., Grandoni, F., and Italiano, G. F. (2013). Data structures resilient to memory faults: An experimental study of dictionaries. *ACM Journal of Experimental Algorithmics*, 18.

[51] Pilla, L., Rech, P., Silvestri, F., Frost, C., Navaux, P., Sonza Reorda, M., and Carro, L. (2014). Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. *IEEE Transactions on Nuclear Science*, 61(4):1874–1880.

[52] Pippenger, N. (1977). Superconcentrators. *SIAM Journal on Computing*, 6(2):298–304.

[53] Ranjan, D., Savage, J., and Zubair, M. (2012). Upper and lower i/o bounds for pebbling r-pyramids. *Journal of Discrete Algorithms*, 14:2–12.

[54] Rech, P., Pilla, L., Silvestri, F., Navaux, P., and Carro, L. (2013). Neutron sensitivity and software hardening strategies for matrix multiplication and FFT on graphics processing units. In *Proc. 3rd Workshop on Fault-tolerance for HPC at Extreme Scale (FTXS)*, pages 13–20.

[55] Romani, F. (1982). Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM Journal on Computing*, 11(2):263–267.

[56] Savage, J. E. (1984). The performance of multilective vlsi algorithms. *Journal of Computer and System Sciences*, 29(2):243–273.

[57] Savage, J. E. (1995). Extending the hong-kung model to memory hierarchies. In *Computing and Combinatorics*, pages 270–281. Springer.

[58] Savage, J. E. (1998). Models of computation. *Exploring the Power of Computing.*

[59] Savage, J. E. and Swamy, S. (1978). Space-time trade-offs on the fft algorithm. *Information Theory, IEEE Transactions on*, 24(5):563–568.

[60] Schönhage, A. (1981). Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455.

[61] Schroeder, B., Pinheiro, E., and Weber, W. D. (2011). DRAM errors in the wild: a large-scale field study. *Communications of the ACM*, 54(2):100–107.

[62] Scott, J., Holtz, O., and Schwartz, O. (2015). Matrix multiplication i/o-complexity by path routing. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 35–45. ACM.

[63] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

[64] Strassen, V. (1986). The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 49–54. IEEE.

[65] Tompa, M. (1978). Time-space tradeoffs for computing functions, using connectivity properties of their circuits. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 196–204. ACM.

[66] Tompa, M. (1982). Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM Journal on Computing*, 11(1):130–137.

[67] V. A. Zalgaller, A. B. Sossinsky, Y. D. B. (1989). *The American Mathematical Monthly*, 96(6):544–546.

[68] Valiant, L. G. (1975). On non-linear lower bounds in computational complexity. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 45–53. ACM.

[69] Venkateswaran, H. and Tompa, M. (1989). A new pebble game that characterizes parallel complexity classes. *SIAM Journal on Computing*, 18(3):533–549.

[70] Williams, V. V. (2012). Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898. ACM.

[71] Yang, C.-Q. and Miller, B. P. (1988). Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 366–373. IEEE.

Appendix

Table 1: Study of the possible compositions of sub-sets of output vertices of *Enc* for Lemma 4.15 in Section 4.6.1

composition code	$\|\mathbf{y}\|$	y_1	y_2	y_3	y_4	y_5	y_6	y_7	$c(\mathbf{y})$
0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	1	1
2	1	0	0	0	0	0	1	0	1
4	1	0	0	0	0	1	0	0	1
8	1	0	0	0	1	0	0	0	1
16	1	0	0	1	0	0	0	0	1
32	1	0	1	0	0	0	0	0	1
64	1	1	0	0	0	0	0	0	1
3	2	0	0	0	0	0	1	1	2
5	2	0	0	0	0	1	0	1	2
6	2	0	0	0	0	1	1	0	2
9	2	0	0	0	1	0	0	1	2
10	2	0	0	0	1	0	1	0	2
12	2	0	0	0	1	1	0	0	2
17	2	0	0	1	0	0	0	1	2
18	2	0	0	1	0	0	1	0	2
20	2	0	0	1	0	1	0	0	2
24	2	0	0	1	1	0	0	0	2
33	2	0	1	0	0	0	0	1	2
34	2	0	1	0	0	0	1	0	2
36	2	0	1	0	0	1	0	0	2
40	2	0	1	0	1	0	0	0	2
48	2	0	1	1	0	0	0	0	2
65	2	1	0	0	0	0	0	1	2
66	2	1	0	0	0	0	1	0	2

68	2	1	0	0	0	1	0	0	2
72	2	1	0	0	1	0	0	0	2
80	2	1	0	1	0	0	0	0	2
96	2	1	1	0	0	0	0	0	2
7	3	0	0	0	0	1	1	1	3
11	3	0	0	0	1	0	1	1	3
13	3	0	0	0	1	1	0	1	3
14	3	0	0	0	1	1	1	0	3
19	3	0	0	1	0	0	1	1	3
21	3	0	0	1	0	1	0	1	3
22	3	0	0	1	0	1	1	0	3
25	3	0	0	1	1	0	0	1	3
26	3	0	0	1	1	0	1	0	3
28	3	0	0	1	1	1	0	0	3
35	3	0	1	0	0	0	1	1	3
37	3	0	1	0	0	1	0	1	3
38	3	0	1	0	0	1	1	0	3
41	3	0	1	0	1	0	0	1	3
42	3	0	1	0	1	0	1	0	3
44	3	0	1	0	1	1	0	0	3
49	3	0	1	1	0	0	0	1	3
50	3	0	1	1	0	0	1	0	3
52	3	0	1	1	0	1	0	0	3
56	3	0	1	1	1	0	0	0	3
67	3	1	0	0	0	0	1	1	3
69	3	1	0	0	0	1	0	1	3
70	3	1	0	0	0	1	1	0	3
73	3	1	0	0	1	0	0	1	3
74	3	1	0	0	1	0	1	0	3
76	3	1	0	0	1	1	0	0	3
81	3	1	0	1	0	0	0	1	3
82	3	1	0	1	0	0	1	0	3
84	3	1	0	1	0	1	0	0	3
88	3	1	0	1	1	0	0	0	2
97	3	1	1	0	0	0	0	1	3
98	3	1	1	0	0	0	1	0	3
100	3	1	1	0	0	1	0	0	3

104	3	1	1	0	1	0	0	0	3
112	3	1	1	1	0	0	0	0	3
15	4	0	0	0	1	1	1	1	4
23	4	0	0	1	0	1	1	1	4
27	4	0	0	1	1	0	1	1	4
29	4	0	0	1	1	1	0	1	3
30	4	0	0	1	1	1	1	0	4
39	4	0	1	0	0	1	1	1	4
43	4	0	1	0	1	0	1	1	4
45	4	0	1	0	1	1	0	1	4
46	4	0	1	0	1	1	1	0	4
51	4	0	1	1	0	0	1	1	4
53	4	0	1	1	0	1	0	1	4
54	4	0	1	1	0	1	1	0	4
57	4	0	1	1	1	0	0	1	4
58	4	0	1	1	1	0	1	0	3
60	4	0	1	1	1	1	0	0	4
71	4	1	0	0	0	1	1	1	4
75	4	1	0	0	1	0	1	1	4
77	4	1	0	0	1	1	0	1	3
78	4	1	0	0	1	1	1	0	4
83	4	1	0	1	0	0	1	1	4
85	4	1	0	1	0	1	0	1	3
86	4	1	0	1	0	1	1	0	4
89	4	1	0	1	1	0	0	1	3
90	4	1	0	1	1	0	1	0	3
92	4	1	0	1	1	1	0	0	3
99	4	1	1	0	0	0	1	1	4
101	4	1	1	0	0	1	0	1	4
102	4	1	1	0	0	1	1	0	4
105	4	1	1	0	1	0	0	1	4
106	4	1	1	0	1	0	1	0	3
108	4	1	1	0	1	1	0	0	4
113	4	1	1	1	0	0	0	1	4
114	4	1	1	1	0	0	1	0	3
116	4	1	1	1	0	1	0	0	4
120	4	1	1	1	1	0	0	0	3

31	5	0	0	1	1	1	1	1	4
47	5	0	1	0	1	1	1	1	4
55	5	0	1	1	0	1	1	1	4
59	5	0	1	1	1	0	1	1	4
61	5	0	1	1	1	1	0	1	4
62	5	0	1	1	1	1	1	0	4
79	5	1	0	0	1	1	1	1	4
87	5	1	0	1	0	1	1	1	4
91	5	1	0	1	1	0	1	1	4
93	5	1	0	1	1	1	0	1	3
94	5	1	0	1	1	1	1	0	4
103	5	1	1	0	0	1	1	1	4
107	5	1	1	0	1	0	1	1	4
109	5	1	1	0	1	1	0	1	4
110	5	1	1	0	1	1	1	0	4
115	5	1	1	1	0	0	1	1	4
117	5	1	1	1	0	1	0	1	4
118	5	1	1	1	0	1	1	0	4
121	5	1	1	1	1	0	0	1	4
122	5	1	1	1	1	0	1	0	3
124	5	1	1	1	1	1	0	0	4
63	6	0	1	1	1	1	1	1	4
95	6	1	0	1	1	1	1	1	4
111	6	1	1	0	1	1	1	1	4
119	6	1	1	1	0	1	1	1	4
123	6	1	1	1	1	0	1	1	4
125	6	1	1	1	1	1	0	1	4
126	6	1	1	1	1	1	1	0	4
127	7	1	1	1	1	1	1	1	4