

On the I/O Complexity of the Cocke-Younger-Kasami Algorithm and of a Family of Related Dynamic Programming Algorithms

Lorenzo De Stefani¹  

Department of Computer Science, Brown University, Providence, RI, USA

Vedant Gupta  

Department of Computer Science, Brown University, Providence, RI, USA

Abstract

Asymptotically tight lower bounds are derived for the Input/Output (I/O) complexity of a class of dynamic programming algorithms, including matrix chain multiplication, optimal polygon triangulation, and the construction of optimal binary search trees. Assuming no recomputation of intermediate values, we establish an $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ I/O lower bound, where n denotes the size of the input and M denotes the size of the available fast memory (cache). When recomputation is allowed, we show that the same bound holds for $M < cn$, where c is a positive constant. In the case where $M \geq 2n$, we show an $\Omega(n/B)$ I/O lower bound. We also discuss algorithms for which the number of executed I/O operations matches asymptotically each of the presented lower bounds, which are thus asymptotically tight.

Additionally, we refine our general method to obtain a lower bound for the I/O complexity of the Cocke-Younger-Kasami algorithm, where the size of the grammar impacts the I/O complexity. An upper bound with asymptotically matching performance in many cases is also provided.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms

Keywords and phrases I/O complexity, Dynamic Programming Algorithms, Lower Bounds, Recomputation, Cocke-Younger-Kasami

Digital Object Identifier 10.4230/LIPIcs.WADS.2025.49

Related Version *Full Version*: <https://arxiv.org/abs/2410.20337> [21]

1 Introduction

The performance of computing systems is significantly influenced by data movement, affecting both time and energy consumption. This ongoing technological trend [37] is expected to continue as fundamental limits on minimum device size and maximum message speed lead to inherent costs associated with data transfer, be it across the levels of a hierarchical memory system or between processing elements in a parallel system [10]. While the communication requirements of algorithms have been extensively studied, deriving significant lower bounds based on these requirements and matching upper bounds remains a critical challenge.

Dynamic Programming (DP) is a classic algorithmic framework introduced by Bellman [4] that is particularly effective in optimization tasks where problems can be broken down into overlapping subproblems with optimal substructure and storing the solutions to these subproblems to prevent redundant work. DP is widely used in computer science and other important fields such as control theory, operations research, and computational biology.

¹ Corresponding author



© Lorenzo De Stefani and Vedant Gupta;

licensed under Creative Commons License CC-BY 4.0

19th International Symposium on Algorithms and Data Structures (WADS 2025).

Editors: Pat Morin and Eunjin Oh; Article No. 49; pp. 49:1–49:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we analyze the memory access cost (*I/O complexity*) of a family of DP algorithms when executed in a two-level storage hierarchy with M words of fast memory (cache). The analyzed algorithms share a similar structure of dependence on the results of subproblems and include, among others, the classic DP algorithms for matrix chain multiplication, boolean parenthesization, optimal polygon triangulation, and construction of optimal binary search trees. We obtain asymptotically tight lower bounds for the I/O complexity of these algorithms both for schedules in which no intermediate values are computed more than once and for general schedules allowing recomputation. Our analysis reveals that, for a range of values of the size of the cache, recomputing intermediate values can lead to an asymptotically lower number of I/O operations. We further refine our analysis to derive an I/O lower bound for the Cocke-Younger-Kasami (CYK) algorithm [20, 47, 32] and an (almost) asymptotically matching upper bound.

Previous and related work. I/O complexity was introduced in the seminal work by Hong and Kung [31]; it denotes the number of data transfers between the two levels of a memory hierarchy with a cache of M words and a slow memory of unbounded size. Hong and Kung presented techniques to develop lower bounds for the I/O complexity of computations modeled by *Computational Directed Acyclic Graphs* (CDAGs).

While the “*edge expansion technique*” of [3], the “*path routing technique*” of [40], and the “*closed dichotomy width*” technique of [11] all yield I/O lower bounds that apply only to computational schedules for which no intermediate result is ever computed more than once (*nr-computations*) it is also important to investigate what can be achieved with recomputation. In fact, for some CDAGs, recomputing intermediate values reduces the space and/or the I/O complexity of an algorithm [39]. In [8], it is shown that some algorithms admit a *portable schedule* (i.e., a schedule that achieves optimal performance across memory hierarchies with different access costs) only if recomputation is allowed. Recomputation can also enhance the performance of simulations among networks (see [34] and references therein) and plays a key role in the design of efficient area-universal VLSI architectures. Several lower-bound techniques that allow recomputation have been presented in the literature, including the “*S-partition technique*” [31], the “*S-span technique*” [39], the “*S-covering technique*” [9] (which merges and extends aspects from both [31] and [39]), the “*G-flow technique*” [5, 6], and the “*visit partition technique*” [7]. However, to the best of our knowledge, none of these have been previously applied to the family of DP algorithms considered in this work.

A systematic study of *Dynamic Programming* was introduced by Bellman [4], and has since become a popular problem-solving methodology. Galil and Park [27] produce a survey and classifications of popular dynamic programming algorithms and optimization methods. Several results have been presented in the literature aimed at optimizing the running time of Dynamic Programming algorithms [28, 33, 46, 27]. Notably, Valiant [45] showed that context-free recognition can be carried out as fast as Boolean matrix multiplication.

Cherng and Ladner [13] provide a cache-oblivious divide-and-conquer algorithm and a cache-aware blocked algorithm for *simple dynamic programs*, which includes matrix chain multiplication and the construction of optimal binary search trees. Their approach is based on Valiant’s algorithm [45] and incurs $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$ cache misses. In [16, 17], Choudhury and Ramachandran introduce the *Gaussian Elimination Paradigm (GEP)* that provides cache-oblivious algorithms for problems including all-pairs shortest path. The authors also provide a way to transform simple DP problems into GEP problems. A proof of the optimality of the proposed algorithm for GEP problems is provided based on a similar result given in [31] for the matrix multiplication problem. As we show, this result does not extend to simple

DP algorithms, as fewer cache misses can be achieved in some scenarios. Cache-oblivious DP algorithms for other problems have been explored [36, 19, 22]. There has also been work on parallelizing cache-efficient DP algorithms [18, 14, 30, 44, 43, 25, 23, 12], experimental evaluations of the resulting performance improvement [38], and on the automated discovery of cache-efficient DP algorithms [15, 29].

Dunlop et al. [24] provide an empirical analysis of various strategies to reduce the number of accesses to the grammar for the *CYK* algorithm. There have also been other works on modifying a grammar representation to improve efficiency. Notably, Song et al. [42] provides an empirical analysis of different grammar binarization approaches to improve efficiency, while Lange et al. [35] argue for the merits of a modified version of CYK to work with grammars in *binary normal form* instead of *Chomsky normal form*.

Summary of results. In this work, we analyze the I/O complexity of a family of DP algorithms that follow the general strategy outlined in *Prototype Algorithm 1* when executed in a two-level storage hierarchy with M words of cache. The analyzed algorithms share a similar structure of dependence on the results of subproblems while allowing for diverse implementation choices and optimal problem substructure.

We analyze both schedules in which no intermediate values are computed more than once (no-recomputation schedules) and general ones that allow for recomputation. For both settings, we derive an $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ lower bound to the number of I/O operations required by sequential execution of these algorithms on an input of size n in a two-level storage hierarchy with a cache of size $M \leq cn$, for some positive constant value c and such that $B \geq 1$ memory words can be moved between consecutive memory locations of the cache and of the slow memory with a single I/O operation. However, we show that while no-recomputation schedules require $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ I/O operations even when using a cache of size at least $2n$ and $o(n^2)$, schedules using recomputation achieve an asymptotic reduction of the number of required I/O operations by a factor $\Theta\left(n^2/\sqrt{M}\right)$. This is particularly significant as in many cases of interest (*e.g.*, Matrix Multiplication [2, 5, 31], Fast Fourier Transform [31], Integer Multiplication [6]) recomputation has shown to enable a reduction of the I/O cost by at most a constant multiplicative factor. These results are obtained by analyzing a representation of the considered algorithms as Computational Directed Acyclic Graphs (CDAGs) whose structure captures the structure of dependence between subproblems, which is common to the DP algorithms considered in our work. We provide a general construction of such CDAGs and analyze their internal connection properties.

We refine our general method to obtain an $\Omega\left(\frac{n^3\Gamma}{\sqrt{MB}}\right)$ lower bound for the I/O complexity of the Cocke-Younger-Kasami algorithm when deciding whether an input string of length n is a member of the language generated by a given Context-Free Grammar with Γ rules with distinct right-hand-sides not including terminals. While the CYK algorithm exhibits a structure of subproblem dependencies similar to that of previously analyzed problems, it presents several challenging differences which we address by modifying our lower-bound methods by further refining its CDAG representation. Finally, we present a cache-oblivious implementation of the CYK algorithm for which, depending on the composition of the rules of the considered CFG, the number of I/O operations to be executed *almost* asymptotically matches the lower bound.

■ **Algorithm 1** Prototype DP algorithm \mathcal{A}^* .

```

1: Input  $\{x_0, x_1, \dots, x_n\}$ 
2: Output  $S(1, n)$ 
3: for  $i = 1$  to  $n$  do                                      $\triangleright$  Initialization of subproblems  $S(i, i)$ 
4:    $S(i, i) \leftarrow \text{INITIALIZATION\_VALUE}$ 
5: for  $l = 2$  to  $n$  do
6:   for  $i = 1$  to  $n - l + 1$  do
7:      $j \leftarrow i + l - 1$ 
8:      $S(i, j) \leftarrow \text{LEAST\_OPTIMAL\_VALUE}$             $\triangleright$  Initialization of accumulator subproblem  $S(i, j)$ 
9:     for  $k = i$  to  $j - 1$  do
10:       $q \leftarrow \text{COMBINE}(S(i, k), S(k + 1, j))$ 
11:       $S(i, j) \leftarrow \text{AGGREGATE}(S(i, j), q)$ 
12: return  $S(1, n)$ 

```

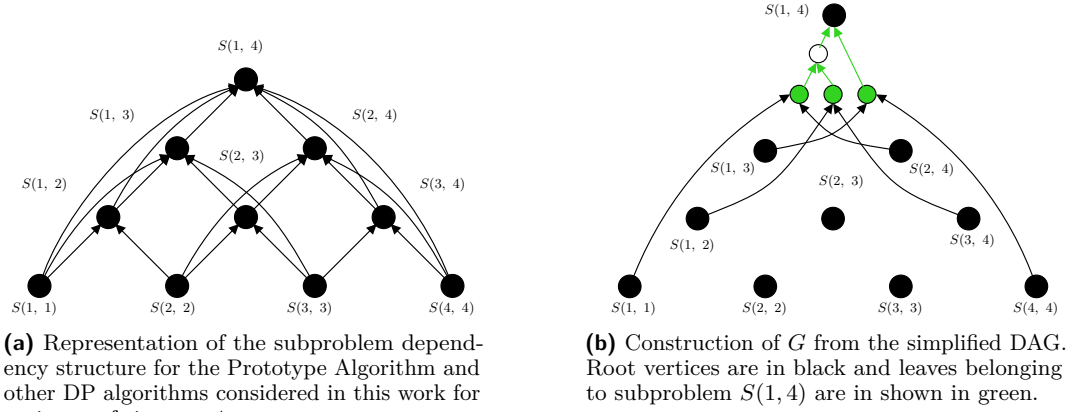
2 Preliminaries

We consider a family of Dynamic Programming (DP) algorithms following the general strategy outlined in *Prototype Algorithm 1*: Given an input of size n , the solution, denoted as $S(1, n)$ is computed bottom up. First, the values corresponding to the solution of subproblems of input size one, $S(i, i)$ for $i \in \{1, 2, \dots, n\}$, are initialized to some set default value. The results of subproblems corresponding to parts of the input of growing size ℓ , that is $S(i, i + \ell)$ for $i \in \{1, 2, \dots, n\}$ and $\ell \in \{1, 2, \dots, n - i\}$, are computed by first evaluating the ℓ compositions of pairs of subproblems $S(i, i + k)$ and $S(i + k + 1, i + \ell)$, for $k \in \{0, 1, \dots, \ell - 1\}$ and then composing them according to the optimal subproblem structure used in the algorithm.

Depending on the specific problems, the default initialization values (`INITIALIZATION_VALUE` in the Prototype Algorithm) of $S(i, i)$, the way that the pairs $S(i, i + k)$ and $S(i + k + 1, i + \ell)$ are combined (`COMBINE`), and the way these results are composed to evaluate $S(i, i + \ell)$ (`AGGREGATE`) may differ. However, `AGGREGATE` is assumed to be commutative and associative. Besides these differences, all the considered subproblems share the subproblem dependence structure outlined in the Prototype Algorithm, for which a visual representation is provided in Figure 1a. Examples of algorithms following such a structure include the classic DP algorithm for the matrix chain multiplication problem, the optimal convex polygon triangulation, the construction of optimal binary search trees, and the Cocke-Younger-Kasami algorithm (Algorithm 5).

We provide a general analysis of the I/O complexity of algorithms following the structure of the Prototype Algorithm \mathcal{A}^* , and we then refine our analysis for the mentioned algorithms of interest.

Computational Directed Acyclic Graphs. Our analysis is based on modeling the execution of the DP algorithms of interest as a *Computational Directed Acyclic Graph* (CDAG) $G = (V, E)$. Each vertex $v \in V$ represents either an input value or the result of an operation (i.e., an intermediate result or one of the output values) which is stored using a single memory word. The *directed* edges in E represent data dependencies. That is, vertices $u, v \in V$ are connected by an edge (u, v) directed from u to v if and only if the value corresponding to u is an operand of the unit time operation computing the value corresponding to v . Vertex u is said to be a *predecessor* of v and v is said to be a *successor* of u . For a given vertex v , the *set of its predecessors* (resp., *successors*) is defined as $\text{pre}(v) = \{u \in V \text{ s.t. } (u, v) \in E\}$ (resp., $\text{suc}(v) = \{u \in V \text{ s.t. } (v, u) \in E\}$). We refer to vertices with no predecessors (resp., successors) as the *input* (resp., *output*) vertices of a CDAG. We say that $G' = (V', E')$ is a *sub-CDAG* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. We say that two sub-CDAGs $G' = (V', E')$ and $G'' = (V'', E'')$ of G are *vertex-disjoint* if $V' \cap V'' = \emptyset$.



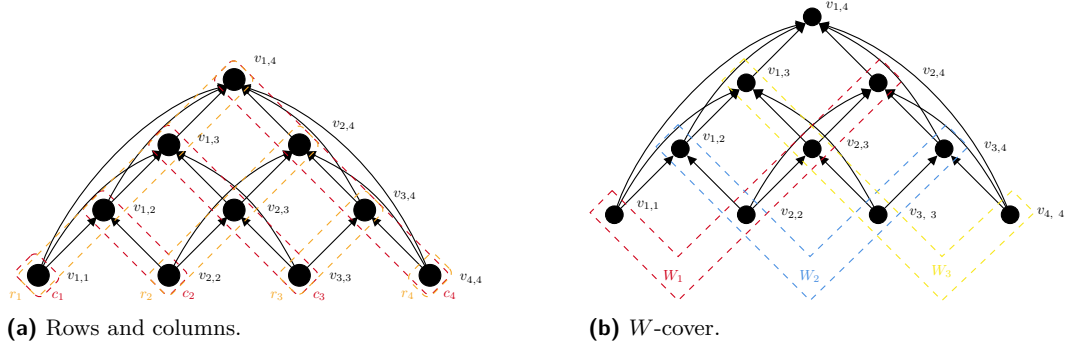
■ **Figure 1** Construction of CDAG in $G(n)$.

I/O model. We assume that sequential computations are executed on a system with a two-level memory hierarchy, consisting of a cache of size M (measured in memory words) and a *slow memory* of unlimited size. An operation can be executed only if all its operands are in the cache. We assume that the processor is equipped with standard elementary logic and algebraic operations. Data can be moved from the slow memory to the cache by **read** operations, and, in the other direction, by **write** operations. These operations are called *I/O operations*. We assume that the input data is stored in the slow memory at the beginning of the computation. The evaluation of a CDAG in this model can be analyzed using the “*red-blue pebble game*” [31]. The number of I/O operations executed when evaluating a CDAG depends on the “*computational schedule*”, that is, on the order in which vertices are evaluated and on which values are kept in/discarded from the cache. The *I/O complexity* of a CDAG G corresponding to the execution of algorithm \mathcal{A} for an input of size n , denoted as $IO_{\mathcal{A}}(M)$, is defined as the minimum number of I/O operations over all possible computational schedules. We further consider a generalization of this model known as the “*External Memory Model*” by Aggarwal and Vitter [1] where $B \geq 1$ memory words can be moved between consecutive memory locations of the cache and of the slow memory with a single I/O operation.

3 CDAG construction

In this section, we give the construction for a family of CDAGs representing the execution of a class of DP algorithms exhibiting a subproblem optimality structure as the one outlined for the Prototype Algorithm 1.

Given input size n , the CDAG G corresponding to the execution of \mathcal{A}^* is constructed using the CDAG in Figure 1a as the base structure. The actual CDAG is obtained by refining the basic structure by replacing each vertex corresponding to a subproblem $S(i, j)$ with a directed binary tree DAG with $j - i$ leaves with all edges directed towards the root. The *root vertex* (*R*-vertex for short) of the tree DAG, henceforth referred to as $v_{i,j}$, corresponds to the computation of the solution of the associated subproblem $S(i, j)$. The k -th “*leaf vertex*” (*L*-vertex for short) of the tree, for $k \in \{0, \dots, j - i - 1\}$ results from the combination of subproblems $S(i, i + k)$ and $S(i + k + 1, j)$ and has as predecessors the *R*-vertices of the tree DAGs corresponding respectively to $S(i, i + k)$ and $S(i + k + 1, j)$ (i.e., $v_{i,i+k}$ and $v_{i+k+1,j}$) (a visual representation is provided in Figure 1b). Notice that binary trees corresponding to different *R*-vertices are vertex disjoint. For each such tree DAG, we say that its $j - i$ *L*-vertices *belong to the root* $v_{i,j}$.



■ **Figure 2** Representation of rows, columns, and the W -cover for input size $n = 4$.

Our lower bound results hold for *any possible structure* of these tree DAGs. This is a *feature* of our model meant to accommodate a variety of possible implementations for \mathcal{A}^* and any other DP sharing the same substructure.

As an example, for the classic DP algorithm for the matrix chain multiplication problem (presented in the appendix of the extended online version [21]), the root vertex of each tree sub-CDAG $S(i, j)$ corresponds to the computation of the minimum number of operations required to compute the chain product of the matrices from the i -th to the j -th, and the k -th leaf, for $k \in \{0, \dots, j-i-1\}$, corresponds to the computation of $S(i, i+k) + S(i+k+1, j) + d_{i-1}d_{i+k}d_j$.

We denote the family of CDAGs constructed in such a way with n input vertices and differing in tree sub-DAGs used to accumulate the values associated to L -vertices into the values associated with the corresponding R -vertex as $\mathcal{G}(n)$. This family of CDAGs captures the dependence structure between subproblems common to the family of DP algorithms that are the focus of our analysis.

Analysis of CDAG properties. Given a CDAG G from the family $\mathcal{G}(n)$, let R (resp. L) denote the set its of R -vertices (resp. L -vertices). Then, $|R| = \frac{n(n+1)}{2}$ and $|L| = \frac{n^3-n}{6}$.

The “ i -th row” (resp., the “ j -th column”) of G , written r_i (resp., c_j), is defined as the subset of R including all of the R -vertices $v_{i,j}$ each corresponding to the computation of subproblem $S(i, j)$ for $j \in \{i, \dots, n\}$ (resp., $i \in \{1, \dots, j\}$). For any $v \in R$, if $v \in r_i$ and $v \in c_j$ then $i \leq j$. By construction, the rows (r_1, r_2, \dots, r_n) (resp., the columns (c_1, c_2, \dots, c_n)) partition R , and $|r_i| = n - i + 1$ (resp., $|c_i| = i$) for $1 \leq i \leq n$. A visual representation of rows and columns can be found in Figure 2a.

By construction, each L -vertex has a distinct pair of R -vertices as its predecessors. Such a pair is said to *interact* in the computation. Given $v_{i,j} \in R$, it interacts with all and only the vertices $v_{k,i-1}$ for $1 \leq k \leq i-1$, and the vertices $v_{j+1,l}$ for $j+1 \leq l \leq n$. All such $n+i-j-1$ interactions correspond to the computation of a L -vertex belonging to a distinct sub-tree corresponding to the subproblem $S(k, j)$ or $S(i, l)$ (i.e., to the R -vertex $v_{k,j}$ or $v_{i,l}$). Pairs of vertices in the same row (or column) do not interact.

We define $W_i = r_{i+1} \cup c_i$, for $1 \leq i \leq n-1$ as the subsets of R such that each vertex in c_i interacts with all the vertices of r_{i+1} (a visual representation can be found in Figure 2b). By construction, $|W_i| = n$. A vertex $v_{1,j} \in r_1$ (resp., $v_{i,n} \in c_n$) is only a member of W_j (resp., W_{i-1}). All remaining vertices $v_{i,j} \in R$ are members of both W_j and W_{i-1} , which are distinct as, by construction, for any $v_{i,j} \in R$ we have $i \leq j$. We refer to the collection of sets $(W_1, W_2, \dots, W_{n-1})$ as the “ W -cover” of the R -vertices of CDAG G . By its definition, and by the properties of the interactions between vertices according to placement in rows and columns discussed in the previous paragraphs, we have the following property:

► **Lemma 1.** *Given a CDAG $G \in \mathcal{G}(n)$, let W_1, W_2, \dots, W_{n-1} denote the W -cover of its R -vertices. For any set $X \subseteq W_i$ there are at most $\frac{|X|^2}{4} \leq \frac{n^2}{4}$ pairs of vertices (u, v) , where $u \in r_{i+1}$ and $v \in c_i$, that interact as predecessors of distinct L -vertices each of which belongs to a distinct R -vertex.*

4 I/O analysis for computations of CDAGs in $\mathcal{G}(n)$

We analyze the I/O complexity of CDAGs in the class $\mathcal{G}(n)$ according to the *Red-Blue Pebble Game* by Hong and Kung [31]. To simplify the presentation, we assume that n and M are powers of 2, i.e., $n = 2^i$ and $M = 2^j$ for integers $i, j \geq 3$. If that is not the case, our analysis can be adapted with minor, albeit tedious, modifications resulting in different constant terms.

4.1 I/O lower bound for computations with no recomputation

In this section, we present a lower bound to the I/O complexity of algorithms corresponding to CDAGs in $\mathcal{G}(n)$, assuming that no intermediate value is ever recomputed. (i.e., under the “no-recomputation assumption”). We first introduce a useful technical Lemma:

► **Lemma 2.** *Given $n \in \mathbb{N}^+$ non-negative real numbers a_1, a_2, \dots, a_n such that $\sum_{i=1}^n a_i^2 \geq T$, where $T \in \mathbb{R}^+$, and $\max_{i=1}^n a_i^2 \leq L$, where $L \in \mathbb{R}^+$ and $L < T$, then, $\sum_{i=1}^n a_i \geq \frac{T}{\sqrt{L}}$.*

Proof. The lower bound comes from the intuitive strategy of making each a_i as large as possible, i.e. if $q = \lfloor \frac{T}{L} \rfloor$, then assign $a_1, \dots, a_q = \sqrt{L}$ and $a_{q+1} = \sqrt{T - qL}$. In this case, $\sum_{i=1}^{q+1} a_i = q\sqrt{L} + \sqrt{T - qL} = q\sqrt{L} + \sqrt{\frac{T}{L} - q}\sqrt{L} \geq q\sqrt{L} + (\frac{T}{L} - q)\sqrt{L}$ (as $\frac{T}{L} - q < 1$) = $\frac{T}{\sqrt{L}}$. To show the optimality of this strategy, we show that any other configuration is suboptimal.

Consider any other assignment of a_1, \dots, a_n that is not just a permutation of the same values. By assumption, any such assignment must have at least two non-zero values $a_i, a_j < \sqrt{L}$. Let $k = a_i + a_j$. If $k \leq \sqrt{L}$, since $k^2 > a_i^2 + a_j^2$, there exists some $c \in \mathbb{R}^+$ such that $c^2 = a_i^2 + a_j^2$ and $c < a_i + a_j$, showing that our assignment a_1, \dots, a_n does not minimize $\sum_{i=1}^n a_i$.

Alternatively, if $k > \sqrt{L}$, then $a_i^2 + a_j^2 = a_i^2 + (k - a_i)^2 < \sqrt{L}^2 + (k - \sqrt{L})^2$ (the last step follows from the fact that $f(x) = x^2 + (k - x)^2$ defines an upwards-facing parabola with an axis of symmetry about $x = \frac{k}{2}$). This means that there exists some $c < k - \sqrt{L}$ such that $\sqrt{L}^2 + c^2 = a_i^2 + b^2$ and $L + c < a + b$, again showing that our assignment a_1, \dots, a_n does not minimize $\sum_{i=1}^n a_i$. ◀

The following property captures the relation between subsets of L -vertices and their number of distinct R -vertex predecessors:

► **Lemma 3.** *Let $L' \subset L$ such that vertices in L' belong to the tree sub-CDAGs of at most $0 < \rho \leq |L'|$ distinct R -vertices. The set $R' \subseteq R$ containing all R -vertices that are predecessors of at least one vertex in L' has cardinality at least $\lfloor |L'|/\sqrt{\rho} \rfloor$.*

Proof. Consider the W -cover of G as defined in Section 3. As any R -vertex can be a member of at most two distinct W_i 's we have:

$$|R'| \geq \frac{1}{2} \sum_{i=1}^{n-1} |R' \cap W_i|. \quad (1)$$

By definition, each vertex in L has a distinct pair of R -vertex predecessors which, by construction, are both included in exactly one set W_i . Let a_i be the number of vertices from L' with both predecessors in W_i . By the assumptions we have:

$$\sum_{i=1}^{n-1} a_i = |L'| \quad (2)$$

By Lemma 1, for any W_i , each interacting pair of R -vertices in it are the predecessors of a distinct L -vertex belonging to the tree sub-CDAG of a distinct R -vertex. From the assumption that vertices in L' belong to at most ρ distinct roots, we have:

$$\max_{i=1, \dots, n-1} a_i \leq \rho, \quad (3)$$

Furthermore, from Lemma 1, we know that vertices in $|R' \cap W_i|$ can interact to produce at most $|R' \cap W_i|^2/4$ vertices in L' . Thus,

$$\sum_{i=1}^{n-1} \frac{|R' \cap W_i|^2}{4} \geq \sum_{i=1}^{n-1} a_i \Rightarrow \sum_{i=1}^{n-1} |R' \cap W_i| \geq \sum_{i=1}^{n-1} 2\sqrt{a_i} \quad (4)$$

By (1), (4), (2), (3), and Lemma 2 (in this order) we have:

$$|R'| \geq \frac{1}{2} \sum_{i=1}^{n-1} |R' \cap W_i| \geq \sum_{i=1}^{n-1} \sqrt{a_i} \geq \frac{|L'|}{\sqrt{\rho}}. \quad \blacktriangleleft$$

Lemma 3 allows us to claim that to compute a set of L -vertices, it is necessary to access a minimum number of R vertices, which, due to the non-recomputation assumption, must either be in the cache or loaded into it using a **read** I/O operation.

► **Theorem 4.** *For any CDAG $G = (V, E) \in \mathcal{G}(n)$ let \mathcal{A} denote an algorithm corresponding to it. The I/O complexity of an Algorithm \mathcal{A} when run without recomputing any intermediate value using a cache memory of size M and where for each I/O operation it is possible to move up to B memory words stored in consecutive memory locations from the cache to slow memory or vice versa, is:*

$$IO_G(n, M, B) \geq \max \left\{ \frac{(n^3 - n)}{16\sqrt{M}} - \frac{n(n+1)}{2} - 3M, n \right\} \frac{1}{B} \quad (5)$$

Proof. We prove the result for $B = 1$. The result then trivially generalizes for a generic B .

The fact that $IO_G(n, M, 1) \geq n$ follows trivially from our assumption that the input is initially stored in the slow memory and therefore must be loaded into the cache at least once using n **read** I/O operations.

Let \mathcal{C} be any computation schedule for the sequential execution of \mathcal{A} using a cache of size M in which no value corresponding to any vertex of G is computed more than once. We partition \mathcal{C} into non-overlapping segments $\mathcal{C}_1, \mathcal{C}_2, \dots$ such that during each \mathcal{C}_i exactly $8M^{1.5}$ values corresponding to distinct L -vertices, denoted as L_i , are evaluated. Since $|L| = (n^3 - n)/6$ there are $\left\lfloor \left(\frac{n^3}{6} - \frac{n}{6} \right) \frac{1}{8M^{1.5}} \right\rfloor$ such segments. Given L_i , we refer to the set of R -vertices to whom at least one of the vertices in L_i belongs as R_i . These correspond to vertices which are *active* during \mathcal{C}_i . For each interval, we denote as h_i the number of R -vertices whose value is completely evaluated during \mathcal{C}_i . As no value is computed more than once $\sum_i h_i = |R| = n(n+1)/2$. Let g_i denote the number of I/O operations executed during \mathcal{C}_i . We consider two cases:

- The vertices of L_i belong to at least $4M + 1$ distinct R -vertices: For each of the $4M + 1 - h_i$ values corresponding to R -vertices that were *partially computed* in \mathcal{C}_i , a partial accumulator of the values corresponding to their leaves computed during \mathcal{C}_i must either be in the cache at the end of \mathcal{C}_i or must be written into slow memory by means of a **write** I/O operation. Hence, $g_i \geq 4M + 1 - M - h_i$.
- The vertices of L_i belong to at most $4M$ distinct R -vertices: By Lemma 3, the vertices in L_i have at least $4M$ R -vertex predecessors, of whom at most h_i are computed during \mathcal{C}_i . As the considered schedule's values are not computed more than once, each of the remaining $4M - h_i$ values corresponding to predecessor vertices of L_i must either be in the cache at the beginning of \mathcal{C}_i , or read into the cache by means of a read I/O operation. Hence, $g_i \geq 4M - M - h_i$.

The I/O requirement of the algorithm is obtained by combining the cost of each non-overlapping segment:

$$\begin{aligned}
 IO_G(n, M, 1) &\geq \sum_{i=1}^{\lfloor \frac{n^3-n}{48M^{1.5}} \rfloor} g_i \geq \sum_{i=1}^{\lfloor \frac{n^3-n}{48M^{1.5}} \rfloor} (3M - h_i) \geq \left\lfloor \frac{n^3-n}{48M^{1.5}} \right\rfloor 3M - \sum_{i=1}^{\lfloor \frac{n^3-n}{48M^{1.5}} \rfloor} h_i \\
 &\geq \frac{(n^3-n)}{16\sqrt{M}} - 3M - |R|. \quad \blacktriangleleft
 \end{aligned}$$

By Theorem 4, we have that for $M \leq cn^2$, for a sufficiently small constant value $c > 0$, computations of CDAGs in $\mathcal{G}(n)$ in which no value is recomputed require $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ I/O operations.

On the tightness of the bound

In [13], Cherng and Ladner presented the *Valiant's DP-Closure Algorithm*, henceforth referred to as VDP, which allows computing algorithms following the general structure outlined in Prototype Algorithm 1 *cache-obliviously* and without recomputation. When $M < |R| = \frac{n(n+1)}{2}$, VDP executes $\mathcal{O}\left(\frac{n^3}{\sqrt{MB}}\right)$ I/O operations, and for $M \geq |R|$ it executes $\mathcal{O}(n/B)$ I/O operations. Thus, we can conclude that our I/O lower bound for schedules with no recomputation provided in Theorem 4 is asymptotically tight.

VDP utilizes a divide and conquer approach, making recursive calls to smaller subproblems. Here we give a simplified presentation of the Valiant's DP-Closure Algorithm. Please refer to Cherng and Ladner [13] for a detailed presentation.

Consider any algorithm following the Prototype Algorithm 1 with input of size $n - 1$, where we assume $n = 2^a$ for $a \in \mathbb{N}$. All subproblems $S(i, j)$ computed by the algorithm are associated with entries of an $n \times n$ matrix \mathbf{X} so that $S(i, j)$ is represented at $\mathbf{X}[i, j + 1]$. Initially, the values $\mathbf{X}[i, i + 1]$ corresponding to values $S(i, i)$ for $1 \leq i \leq n - 1$ are computed from the input while the remaining entries in \mathbf{X} are set to the **LEAST_OPTIMAL_VALUE** (e.g., 0). To fully compute all the subproblems (i.e., to compute the *DP-closure*), VDP splits \mathbf{X} into 16 equally-sized submatrices of dimension $\frac{n}{4}^2$ labeled as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{12} & \mathbf{X}_{13} & \mathbf{X}_{14} \\ & \mathbf{X}_{22} & \mathbf{X}_{23} & \mathbf{X}_{24} \\ & & \mathbf{X}_{33} & \mathbf{X}_{34} \\ & & & \mathbf{X}_{44} \end{bmatrix}$$

² We assume that n is a power of 2. This can always be made the case by padding \mathbf{X}

49:10 On the I/O Complexity of the CYK Algorithm and DP Algorithms

We omit the representation of the lower-triangular submatrices as these will have all of their entries set to 0. Valiant's DP-Closure Algorithm (whose pseudocode is presented in Algorithm 2) is given the matrix \mathbf{X} as input, with the only non-zero values corresponding to subproblems $S(i, i)$ for $1 \leq i \leq n - 1$. The values of the $S(i, j)$ (*i.e.*, the R -vertices) corresponding to the top-left and bottom-right submatrices are computed through recursive calls, exploiting the fact that these submatrices correspond to a smaller instantiation of the same DP problem. The remaining values in \mathbf{X} are then computed by invoking a subroutine called “*Valiant's Star Algorithm*” (VSTAR).

■ **Algorithm 2** Valiant's DP-Closure Algorithm (VDP).

```

1: Input  $X$ 
2: Output  $X$  ▷ DP-Closure of  $X$  is computed in-place
3: if  $\dim(X) = 2$  then ▷ Input is already computed
4:   return
5:    $\begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{12} \\ & \mathbf{X}_{22} \end{bmatrix} \leftarrow \text{VDP} \left( \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{12} \\ & \mathbf{X}_{22} \end{bmatrix} \right)$ 
6:    $\begin{bmatrix} \mathbf{X}_{33} & \mathbf{X}_{34} \\ & \mathbf{X}_{44} \end{bmatrix} \leftarrow \text{VDP} \left( \begin{bmatrix} \mathbf{X}_{33} & \mathbf{X}_{34} \\ & \mathbf{X}_{44} \end{bmatrix} \right)$ 
7:  $\mathbf{X} \leftarrow \text{VSTAR}(\mathbf{X})$ 

```

VSTAR is given as input a matrix \mathbf{X} where the subproblems in the top-left and bottom-right quadrants are already computed. The algorithm then recursively computes the values corresponding to the remaining entries of \mathbf{X} (*i.e.*, the values of the R -vertices). The pseudocode for this function can be found in Algorithm 3. We represent COMBINE with “ \cdot ” and AGGREGATE with “ $+$ ”. Operations involving these operators in Lines 6, 8, 10, and 11 are computed using a subroute called “*Matrix multiply and accumulate*” that, given three equally sized square matrices \mathbf{A} , \mathbf{B} , and \mathbf{C} , computes $\mathbf{A} + \mathbf{B} \cdot \mathbf{C}$. In the context of VDP, \mathbf{A} contains partially computed values $S(i, j)$'s (*i.e.*, corresponding to R -vertices), and $\mathbf{B} \cdot \mathbf{C}$ represents the computation of combinations of subproblems contributing to $S(i, j)$ (*i.e.*, $S(i, k) \cdot S(k + 1, j)$ for $i \leq k < j$), that is, the values corresponding to the L -vertices, which are then used to update \mathbf{A} . A recursive in-place implementation of this function is presented in detail in [13, 26].

4.2 I/O lower bound for computations with allowed recomputation

We extend our I/O lower bound analysis to general computations in which values *may* be recomputed by combining it with the *dominator* I/O lower bound technique by Hong and Kung [31].

► **Definition 5** ([31]). *Given $G = (V, E)$, a set $D \subseteq V$ is a dominator set for $V' \subseteq V$ if every path directed from any input vertex of G to a vertex in V' contains a vertex of D .*

Dominator sets of R -vertices of a CDAG $G \in \mathcal{G}(n)$ must satisfy the following property:

► **Lemma 6.** *Given $G = (V, E) \in \mathcal{G}(n)$, let $A = \{v_{i_1, j_1}, v_{i_2, j_2}, \dots\}$ be a subset of R -vertices where v_{i_k, j_k} is the R -vertex corresponding to subproblem $S(i_k, j_k)$. Any dominator set D of A has minimum cardinality $|D| \geq \min(|A|, \min j_k - i_k)$*

■ **Algorithm 3** Valiant's Star Algorithm (VSTAR).

```

1: Input  $\mathbf{X}$                                 ▷ Top-left and bottom-right are assumed to be fully computed
2: Output  $\mathbf{X}$                                 ▷ DP-Closure of  $\mathbf{X}$  is computed in-place
3: if  $\dim(\mathbf{X}) = 2$  then                      ▷ No Computation required
4:   return
5:    $\begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{23} \\ & \mathbf{X}_{33} \end{bmatrix} \leftarrow \text{VSTAR} \left( \begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{23} \\ & \mathbf{X}_{33} \end{bmatrix} \right)$ 
6:    $\bar{X}_{13} \leftarrow X_{13} + X_{12} \cdot X_{13}$ 
7:    $\begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{13} \\ & \mathbf{X}_{33} \end{bmatrix} \leftarrow \text{VSTAR} \left( \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{13} \\ & \mathbf{X}_{33} \end{bmatrix} \right)$ 
8:    $\bar{X}_{24} \leftarrow X_{24} + X_{23} \cdot X_{34}$ 
9:    $\begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{24} \\ & \mathbf{X}_{44} \end{bmatrix} \leftarrow \text{VSTAR} \left( \begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{24} \\ & \mathbf{X}_{44} \end{bmatrix} \right)$ 
10:   $\bar{X}_{14} \leftarrow X_{14} + X_{12} \cdot X_{24}$ 
11:   $\bar{X}_{14} \leftarrow X_{14} + X_{13} \cdot X_{34}$ 
12:   $\begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{14} \\ & \mathbf{X}_{44} \end{bmatrix} \leftarrow \text{VSTAR} \left( \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{14} \\ & \mathbf{X}_{44} \end{bmatrix} \right)$ 

```

Proof. Suppose for every $v_{i,j} \in A$, D contains a vertex internal to the tree-CDAG rooted at v_{i_k, j_k} . As the tree-CDAGs of different R -vertices are disjoint, this would imply $|D| \geq |A|$.

Otherwise, there must be some $v_{i,j} \in A$, such that D does not contain any of its internal vertices. Let $L_{i,j} = \{l_0, l_1, \dots, l_{j-i-1}\}$ denote the set of L -vertices corresponding to $v_{i,j}$. It is sufficient to show that there are $j-i$ vertex disjoint paths from the inputs of G to vertices in $L_{i,j}$. By construction, each leaf $l_k \in L_{i,j}$ has a predecessor belonging to a unique column c_{i+k} . In turn, each of these predecessors is connected through its internal tree-CDAG to the input vertex in the same column ($v_{i+k, i+k}$). In this manner, $j-i$ paths ($v_{i+k, i+k} \rightarrow v_{i, i+k} \rightarrow l_k$) for $0 \leq k \leq j-i-1$ are obtained. Since each path contains R -vertices from a different column, and the tree-CDAGs of different R -vertices are disjoint, the paths outlined above are vertex-disjoint. ◀

► **Theorem 7.** For any CDAG $G = (V, E) \in \mathcal{G}(n)$ let \mathcal{A} denote an algorithm corresponding to it. The I/O-complexity of \mathcal{A} when run using a cache memory of size M and where for each I/O operation it is possible to move up to B memory words stored in consecutive memory locations from cache to slow memory or vice versa, is:

$$IO_G(n, M, B) \geq \max \left\{ \frac{(n - 6M - 1)^3}{18\sqrt{M}} - 2M, n \right\} \frac{1}{B} \quad (6)$$

Proof. To simplify our analysis, we only consider *parsimonious execution schedules* such that each time an intermediate result is computed, the result is then used to compute at least one of the values of which it is an operand before being removed from the memory (either the cache or slow memory). Any non-parsimonious schedule \mathcal{C} can be reduced to a parsimonious schedule \mathcal{C}' by removing all the steps that violate the definition of parsimonious computation. \mathcal{C}' has therefore less computational or I/O operations than \mathcal{C} . Hence, restricting the analysis to parsimonious computations leads to no loss of generality.

We prove the result for the case $B = 1$. The result then trivially generalizes for a generic B . The fact that $IO_{\mathcal{A}}(n, M, 1) \geq n$ follows trivially from our assumption that input is initially stored in slow memory and therefore must be loaded into the cache at least once using n read I/O operations.

Let $L(x)$ be the set of L -vertices in G whose predecessor R -vertices both correspond to some subproblems $S(i, j)$ such that $x \leq j - i$. By the construction, for each subproblem $S(i, j)$ the corresponding tree sub-CDAG has $\max\{0, j - i - 2x\}$ such L -vertices. Since the sub-CDAGs corresponding to each $S(i, j)$ do not share vertices, we have:

$$\begin{aligned}
 |L(x)| &= \sum_{j-i=1}^{n-1} (n - (j - i)) \max\{0, j - i - 2x\} \\
 &= \sum_{j-i=2x+1}^{n-1} (n - (j - i)) (j - i - 2x) \\
 &= \sum_{y=1}^{n-1-2x} (n - 2x - y) y \\
 &= \frac{(n - 2x - 1)(n - 2x)(n - 2x + 1)}{6} \\
 &> \frac{(n - 2x - 1)^3}{6}.
 \end{aligned}$$

Let \mathcal{C} be any computation schedule for the sequential execution of \mathcal{A} using a cache of size M . We partition \mathcal{C} into segments $\mathcal{C}_1, \mathcal{C}_2, \dots$ such that during each \mathcal{C}_l exactly $6M^{1.5}$ values corresponding to *distinct vertices* in $L(3M)$ are computed from their operands (*i.e.*, not read from the cache). Let L_l denote the set of vertices corresponding to these values. Below we argue that the number g_l of I/O operations executed during each of the $\lfloor |L(3M)|/6M^{1.5} \rfloor$ non-overlapping segments \mathcal{C}_l is at least $2M$, from whence the statement follows.

Let A denote the set of vertices corresponding to the values computed during \mathcal{C}_l . Clearly $L_l \subseteq A$. Let D denote the set of vertices corresponding to either the at most M values that are either stored in the cache or the values that are read into the cache during \mathcal{C}_l by means of **read** I/O operations. Thus $g_l = |D| - M$. In order to be possible to compute the values corresponding to vertices in A during \mathcal{C}_l there must be no paths from input vertices of G to vertices in A , that is, D must be a dominator set of A .

We refer to the set of R -vertices to whom at least one of the vertices in L_l belongs as R_l . These correspond to vertices which are *active* during \mathcal{C}_l . We consider three mutually exclusive cases:

- (a) At least one of the values corresponding to a vertex v' in R_l is *entirely computed* during \mathcal{C}_l . That is no accumulator of the values corresponding to the leaves of v' is in the cache or it is loaded in it by means of an **read** I/O operation during \mathcal{C}_l . Thus, $v' \in A$ and D do not include any non-leaf vertex internal to the tree-sub-CDAG rooted in v' . Since $L_l \subseteq L(3M)$, by definition, v corresponds to a subproblem $S(i, j)$ such that $j - i \geq 3M$. From Lemma 6, $|D| \geq 3M$ which implies $g_l \geq 2M$.
- (b) $|R_l| > 4M$: As none of the values in R_l is entirely computed during \mathcal{C}_l (case (a)), and as we are considering parsimonious computations, for each value R_l at least one partial accumulator of its value is either in the cache at the beginning (resp., end) of \mathcal{C}_l or is loaded into the cache during (resp., saved to the slow memory) during it. Hence $g_l \geq 4M - 2M$.
- (c) $|R_l| \leq 4M$: By Lemma 3, the vertices in L_l have at least $3M$ distinct R -vertices predecessors. Since $L_l \subseteq L(3M)$ by definition, all such R vertices corresponds to subproblems $S(i, j)$ where $j - i \geq 3M$. As the values of these vertices are either in the cache at the beginning of \mathcal{C}_l , or **read** to the cache during \mathcal{C}_l , or computed during \mathcal{C}_l , D must be dominator set for the corresponding vertices. From Lemma 6, we can thus conclude $|D| \geq 3M$ from whence $g_l \geq 2M$. \blacktriangleleft

■ **Algorithm 4** Top-Down recursive computation schedule for an algorithm \mathcal{A} corresponding to $G \in \mathcal{G}(n)$.

```

1: Input  $\{x_0, x_1, \dots, x_n\}$ 
2: Output  $S(1, n)$ 
3: for  $i = 1$  to  $n$  do                                      $\triangleright$  Initialization of subproblems  $S(i, i)$ 
4:    $S(i, i) \leftarrow \text{READ } i\text{-TH INPUT FROM SLOW MEMORY}$ 
5: procedure  $\text{COMPUTE}(i, j)$ 
6:   if  $i = j$  then
7:     return  $S(i, j)$ 
8:   else
9:      $S(i, j) \leftarrow \text{LEAST\_OPTIMAL\_VALUE}$                 $\triangleright$  Initialization of accumulator subproblem  $S(i, j)$ 
10:    for  $k = i$  to  $j - 1$  do
11:      if  $k - i \geq j - k - 1$  then
12:         $S(i, k) \leftarrow \text{COMPUTE}(i, k)$ 
13:         $S(k + 1, j) \leftarrow \text{COMPUTE}(k + 1, j)$ 
14:      else
15:         $S(k + 1, j) \leftarrow \text{COMPUTE}(k + 1, k)$ 
16:         $S(i, k) \leftarrow \text{COMPUTE}(i, k)$ 
17:       $S(i, j) \leftarrow \text{AGGREGATE}(S(i, j), \text{COMBINE}(S(i, k), S(k + 1, j)))$ 
18:    REMOVE FROM CACHE ALL VALUES USED BY THE PROCEDURE EXCEPT FOR  $S(i, j)$ 
19:    return  $S(i, j)$ 
20: return  $\text{COMPUTE}(1, n)$ 

```

By Theorem 7, we have that for $M \leq c_1 n$, where c_1 is a sufficiently small positive constant value, computations of CDAGs in $\mathcal{G}(n)$ require $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ I/O operations. This is in contrast with the result given in Theorem 4 for schedules with no recomputation that exhibit I/O complexity $\Omega\left(\frac{n^3}{\sqrt{MB}}\right)$ for values of M up to $c_2 n^2$ for an appropriately chosen constant c_2 .

On the tightness of the bound. When *allowing recomputation*, if $M < c_1 n$, where c_1 is a sufficiently small positive constant, the number of I/O operations executed by Valiant's DP-Closure Algorithm asymptotically matches our lower bound in Theorem 7. Thus, for $M < c_1 n$, allowing recomputation does not asymptotically change the required number of I/O operations. On the other hand, if $M > 2n$, our lower bound simplifies to $\Omega(n/B)$, diverging from the complexity of Valiant's DP-closure Algorithm.

In Theorem 8, we show how for any algorithm corresponding to $G = \mathcal{G}(n)$ given a cache memory of suitable size $M \geq 2n$, there exists an execution schedule in which the results of subproblems are computed according to a recursive *top-down* order that executes only $\lceil n/B \rceil + 1$ I/O operations, thus asymptotically matching the general lower bound given in Theorem 7. Crucially, such a schedule involves the recomputation of intermediate results in order to reduce the number of I/O operations otherwise necessary, as established in Theorem 4.

► **Theorem 8.** *Using a cache memory of size $M \geq 2n$, there exists a computational schedule executing \mathcal{A}^* incurring $\lceil n/B \rceil + 1$ I/O operations, where for each I/O operation it is possible to move up to B memory words stored in consecutive memory locations from the cache to slow memory or vice versa.*

Proof. The proposed schedule follows the steps outlined in Algorithm 4: in the initialization phase the n I/O values are read into the cache using $\lceil n/B \rceil$ **read** I/O operations. These values are held in the cache until the very end of the schedule, occupying n out of the $M \geq 2n$ available memory locations. Once the final result $S(1, n)$ is computed, it is written to the secondary memory by means of a **write** I/O operation.

We argue that the invocation of the procedure $\text{COMPUTE}(i, n)$ uses at most n cache memory locations, thus not requiring the execution of any additional I/O operation. Let $\mathcal{M}(j - i)$ denote the memory usage of an invocation of $\text{COMPUTE}(i, j)$. In the base case $\mathcal{M}(1) = 1$, as COMPUTE simply combines and returns the values $S(i, i)$ and $S(j, j)$ which are already stored in the cache after the initialization phase. In the general case, due to the use of accumulators and to the sequence of recursive calls and the memory management used by the schedule, we have:

$$\begin{aligned}\mathcal{M}(j - i) &= 1 + \max\{\mathcal{M}(j - i - 1), 1 + \mathcal{M}(j - i - 2)\} \\ &= 1 + \mathcal{M}(j - i - 1) \\ &= j - i - 1 + \mathcal{M}(1) \\ &= j - i.\end{aligned}$$

The lemma follows as $\mathcal{M}(n - 1) = n - 1$. ◀

Interestingly, while the proposed computation schedule allows to achieve an asymptotic polynomial reduction of the number of I/O operations by a $\Theta\left(\frac{n^2}{\sqrt{M}}\right)$ multiplicative factor using recomputation, this is at the cost of an exponential increase of the number of computation operations executed ($2^{\mathcal{O}(n)}$), in comparison with Valiant's DP-Closure Algorithm which requires only $\mathcal{O}(n^3)$ computations.

4.3 I/O analysis for notable Dynamic Programming algorithms

The general result in Section 4 can be adapted to derive asymptotically tight bounds on the I/O complexity of classic Dynamic Programming algorithms for notable problems such as Matrix Chain Multiplication, Optimal Polygon Triangulation, and the construction of optimal binary search trees. This is achieved by mapping each of these algorithms onto Prototype Algorithm 1, defining the `COMBINE`, `AGGREGATE`, and `LEAST_OPTIMAL_VALUE` appropriately. Due to space limitations, these results are presented in the online full version of this work [21, Appendix A].

5 I/O analysis for the Cocke-Younger-Kasami algorithm

In this section, we present I/O bounds for the Cocke-Younger-Kasami algorithm (CYK). First, we provide some background on the algorithm. We then discuss how our lower bound analysis method must be enhanced to better capture the complexity of this algorithm. Finally, we present an upper bound.

5.1 Background on the CYK algorithm

- **Definition 9** ([41]). *A Context-Free Grammar is a 4-tuple $(V, \mathcal{R}, \Sigma, T)$ where*
- *V is a finite set called the variables;*
 - *Σ is a finite set, disjoint from V , called the terminals;*
 - *\mathcal{R} is a finite relation in $V \times (V \cup \Sigma \cup \epsilon)^*$, where the $*$ represents the Kleene star operation and ϵ denotes the empty string;*
 - *$T \in V$ is the start variable.*

Elements of \mathcal{R} are referred to as *rules* and are generally given in the form $A \rightarrow w$, where $A \in V$, and $w \in (V \cup \Sigma \cup \epsilon)^*$. If $u, v, w \in (V \cup \Sigma \cup \epsilon)^*$ and if $A \rightarrow w$ is a rule of the grammar, we say that uAv yields uwv , written as $uAv \Rightarrow uwv$. We say that u derives v , written $u \Rightarrow^* v$,

■ **Algorithm 5** Pseudocode for the Cocke-Younger-Kasami algorithm.

```

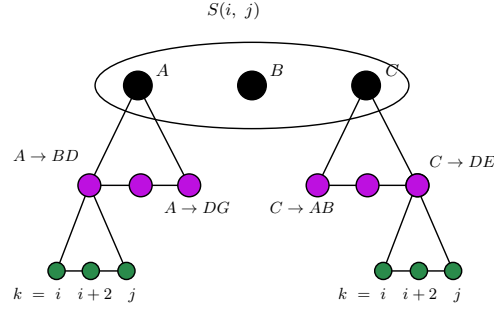
1: Input  $\{w_1, \dots, w_n\}, (\mathcal{V}, \Sigma, \mathcal{R}, T)$ 
2: Output parse
3: parse  $\leftarrow$  False
4:  $S \leftarrow$  False
5: if  $n = 0$  then
6:   if  $(T \rightarrow \epsilon) \in \mathcal{R}$  then
7:     parse  $\leftarrow$  True
8:   return
9: for  $i = 1$  to  $n$  do
10:  for each unit production  $(A \rightarrow a) \in \mathcal{R}$  do
11:    if  $a = w_i$  then
12:       $S(i, i)[A] \leftarrow$  True
13:  for  $l = 2$  to  $n$  do
14:    for  $i = 1$  to  $n - l + 1$  do
15:       $j \leftarrow i + l - 1$ 
16:      for  $k = i$  to  $j - 1$  do
17:        for each binary production  $(A \rightarrow BC) \in \mathcal{R}$  do
18:          if  $S(i, k)[B]$  and  $S(k + 1, j)[C]$  then
19:             $S(i, j)[A] \leftarrow$  True
20: if  $T \in S(1, n)$  then
21:   parse  $\leftarrow$  True

```

if $u = v$ or if there exists a sequence u_1, u_2, \dots, u_k for $k \geq 1$ such that $u_i \in (V \cup \Sigma \cup \epsilon)^*$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$. The *language* of a CFG is the set of all strings $w \in \Sigma^*$ that can be derived from the starting variable T , that is $\{w \in \Sigma^* \text{ s.t. } T \Rightarrow^* w\}$. A CFG is in Chomsky's Normal Form (CNF) if every rule is in the form: $A \rightarrow BC$, or $A \rightarrow a$, or $T \rightarrow \epsilon$, where A is any variable in V , B and C can be any pair in $(V \setminus S) \times (V \setminus S)$, and a can be any terminal in Σ . Any CFG can be transformed into an equivalent one in CNF through a simple iterative algorithm [41]. We refer to the number of rules $|\mathcal{R}|$ as the *size* of a CFG. We denote the set of binary rules $(A \rightarrow BC)$ in a CFG with $\mathcal{R}_B \subset \mathcal{R}$, the set of terminal rules $(A \rightarrow a)$ with $\mathcal{R}_T \subset \mathcal{R}$, and the number of distinct right-hand sides in \mathcal{R}_B with Γ .

Given a CFG $(V, \mathcal{R}, \Sigma, T)$ in CNF, and a string $w = w_1 w_2 \dots w_n$, the Cocke-Younger-Kasami (CYK) algorithm, henceforth referred to as \mathcal{A}_{CYK} , decides whether w is part of the language of the CFG. We refer to the standard implementation provided in Algorithm 5. Let $S(i, j)$ be the set of variables that derive the substring $w_i w_{i+1} \dots w_j$. CYK uses a DP approach to compute all sets $S(i, j)$ for $1 \leq i \leq j \leq n$ using an adaptation of prototype algorithm \mathcal{A}^* . \mathcal{A}_{CYK} determines whether w is a member of the language generated by the given CFG by checking whether $T \in S(1, n)$ (i.e., whether T derives w).

Memory representation model. We assume that the variables in V can be referenced by index and that such index can be stored in a single memory word. We represent subsets of V (the results of subproblems $S(i, j)$) using a one-hot encoding with $\lceil |V|/s \rceil$ memory words, where s denotes the number of bits in a memory word. Thus, given the representation of one such subset in the memory, to check whether a variable V_i is a member of the subset it is sufficient to access the $(i \bmod s)$ -th bit of the $\lfloor i/s \rfloor$ -th memory word.



■ **Figure 3** Each subproblem $S(i, j)$ is represented by multiple *variable roots* (in black), which is the result of the composition multiple *grammar roots* (in purple), which in turn are the composition of multiple *leaves* (in green).

We assume that each grammar rule is represented utilizing at most three memory words each used to represent the index of the left-hand side and at most two to represent the right-hand side (either the index of the two variables or the code for a terminal symbol).

5.2 I/O lower bound

While the CYK algorithm shares the underlying structure of relations between subproblems of the prototype algorithm \mathcal{A}^* , it presents several non-trivial complications due to the application of the grammar rules that, in turn, require a refinement of our analysis methods and, in particular, of the CDAG construction. In this section, we will assume that the set of grammar rules \mathcal{R} are such that no two rules have the same right-hand side. While this might not always hold in practice, our analysis can still be applied by considering a subset of rules of the considered grammar which satisfies this condition.

CYK CDAG construction. The construction of the CDAG representing CYK’s execution for an input string w with n characters and a CFG in CNF $(V, \mathcal{R}, \Sigma, T)$, called $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$, follows a recursive structure which uses as a basis $G = (V, E) \in \mathcal{G}(n)$. We describe the necessary modification focusing on the sub-CDAG for a single $S(i, j)$ subproblem. A visual depiction of the ensuing description is provided in Figure 3:

- For the base of the construction, for each $S(i, i)$, the CDAG has $|V|/s$ vertices, each corresponding to the encoding of one of the possible variables of the grammar. These vertices serve as the input of the CDAG.
- Each R -vertex of G , corresponding to the computation of the solution of one of the subproblems $S(i, j)$, is replaced by $|V|/s$ vertices, each representing a part of the encoding of the set of variables V . We refer to this set of vertices as “*variable roots*” VR .
- Consider a variable root v corresponding to subproblem $S(i, j)$ and variable $A \in V$. v forms the root of a binary tree, which has as its leaves vertices for each rule having A as its left-hand side. We denote the set of these vertices as “*grammar roots*” GR . Note that as only one variable can appear on the left-hand side of a rule, the trees composed in this way are vertex disjoint. Further, for each $S(i, j)$ there will be at least Γ/s such vertices.
- For each subproblem $S(i, j)$, consider one of its grammar roots u corresponding to rule $\gamma \in \mathcal{R}_B$. u forms the root of a binary tree combining the results of $j - i$ “leaf-vertices”. Each leaf vertex corresponds to a combination of the subproblems $S(i, k)$ and $S(k + 1, j)$, for $i \leq k < j$. Such a vertex has as predecessors the variable root vertices of subproblems $S(i, k)$ and $S(k + 1, j)$ encoding the variables appearing in the right-hand side of rule γ .

Given the CDAG $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$, we generalize the notion of rows (resp., columns) introduced in Section 3 by having row r_i (resp. column c_j) include all vertices in VR corresponding to the encoding of the results of subproblems $S(i, j)$ for $1 \leq j \leq n$ (resp., for $1 \leq i \leq n$). The definition of W -cover generalizes to $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$. We observe that a property analogous to that outlined in Lemma 1 holds for $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$ as well:

► **Lemma 10.** *Given a CDAG $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$, let W_1, \dots, W_{n-1} denote its W -cover. Any set of x VR -vertices in W_i contain at most $x^2/4$ interacting pairs, each of which forms the predecessors of a unique L -vertex belonging to a unique GR -vertex.*

Proof. Consider any set of x VR -vertices in W_i containing x_c vertices in column i and $x - x_c$ vertices in row $i + 1$. Since vertices in the same row (or column) don't interact, there are at most $x(x - x_c) \leq x^2/4$ interacting pairs.

Consider any two such distinct interacting pairs: $(v_{k,i}^A, v_{i+1,l}^B)$ and $(v_{m,i}^C, v_{i+1,n}^D)$ (where $v_{k,i}^A$ is a VR -vertex corresponding to variable A in row k and column i). Any leaf produced by $(v_{k,i}^A, v_{i+1,l}^B)$ must correspond to a rule with right-hand side “AB”. From the assumption that every rule in \mathcal{R} has a unique right-hand side, it follows that there is only one such rule and therefore $(v_{k,i}^A, v_{i+1,l}^B)$ must produce a single leaf.

To show that the leaves produced by $(v_{k,i}^A, v_{i+1,l}^B)$ and $(v_{m,i}^C, v_{i+1,n}^D)$ belong to distinct GR -vertices, consider the following two cases:

1. If $k \neq m$ or $l \neq i$, it follows from Lemma 1 that the leaves produced belong to different roots, and therefore must also belong to different grammar roots.
2. Otherwise, it must be the case that $A \neq C$ or $B \neq D$ (since the two interacting pairs are assumed to be distinct). It follows that the two leaves produced correspond to different rules in \mathcal{R} and therefore belong to distinct grammar roots. ◀

I/O lower bound proof. We analyze the I/O complexity of any sequential computation for the CYK algorithm by analyzing the CDAG corresponding to the given CFG and the input string to be considered. We first state two results that correspond each to a modification of the results in Lemma 3 and, respectively, of Lemma 6 for the family of CDAG corresponding to execution of the CYK algorithm:

► **Lemma 11.** *Given a CDAG $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$, let $L' \subset L$ be such that vertices in L' belong to the tree sub-DAGs of at most ρ distinct GR -vertices. The set $R' \subseteq VR$ that includes all VR -vertices that are predecessors of at least one vertex in L' has cardinality at least $|L'|/\sqrt{\rho}$.*

Proof. Consider the W -cover of $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$ with the adapted definition for the CYK CDAG discussed in Section 5.2. As any VR -vertex can be a member of at most two distinct W_i 's we have:

$$|R'| \geq \frac{1}{2} \sum_{i=1}^{n-1} |R' \cap W_i|. \quad (7)$$

By definition, each vertex in L has a distinct pair of VR -vertex predecessors which, by construction, are both included in exactly one set W_i . Let a_i be the number of vertices from L' with both predecessors in W_i . By the assumptions we have:

$$\sum_{i=1}^{n-1} a_i = |L'| \quad (8)$$

49:18 On the I/O Complexity of the CYK Algorithm and DP Algorithms

By Lemma 10, for any W_i , each interacting pair of VR -vertices in it are the predecessors of a distinct L -vertex belonging to the tree sub-CDAG of a distinct GR -vertex. From the assumption that vertices in L' belong to at most ρ distinct roots, we have:

$$\max_{i=1, \dots, n-1} a_i \leq \rho, \quad (9)$$

Furthermore, from Lemma 10, we know that vertices in $|R' \cap W_i|$ can interact to produce at most $|R' \cap W_i|^2/4$ vertices in L' . Thus,

$$\sum_{i=1}^{n-1} \frac{|R' \cap W_i|^2}{4} \geq \sum_{i=1}^{n-1} a_i \Rightarrow \sum_{i=1}^{n-1} |R' \cap W_i| \geq \sum_{i=1}^{n-1} 2\sqrt{a_i} \quad (10)$$

By (7), (10), (8), (9), and Lemma 2 (in this order) we have:

$$|R'| \geq \frac{1}{2} \sum_{i=1}^{n-1} |R' \cap W_i| \geq \sum_{i=1}^{n-1} \sqrt{a_i} \geq \frac{|L'|}{\sqrt{\rho}}. \quad \blacktriangleleft$$

The proof follows a reasoning analogous to that used to prove Lemma 3 adapted for the CDAG $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$ by using the property of L -vertices of the CYK CDAG states in Lemma 10 in place of Lemma 1.

► **Lemma 12.** *Given $G_{CYK}(n, (V, \mathcal{R}, \Sigma, T))$, let $A = \{v_{i_1, j_1}, v_{i_2, j_2}, \dots\}$ be a subset of GR -vertices where v_{i_k, j_k} is a GR -vertex corresponding to subproblem $S(i_k, j_k)$. Any dominator set D of A has cardinality at least $|D| \geq \min(|A|, \min j_k - i_k)$.*

The proof corresponds to the one of Lemma 6 where the tree sub-CDAGs rooted in GR -vertices are considered instead of the R vertices of $\mathcal{G}(n)$.

A lower bound to the I/O complexity of the CYK algorithm can be obtained by following steps analogous to those used in the proof of Theorem 7 with opportune adjustments matching the modifications in the *enhanced* CDAG described in this section.

► **Theorem 13.** *Consider a string w of length n and a CFG $(V, \mathcal{R}, \Sigma, T)$ in CNF. The number of I/O operations executed by \mathcal{A}_{CYK} when deciding whether w is a member of the language of the given grammar using a machine equipped with a cache memory of size M and such that B contiguous can be moved with each I/O operation is bounded as:*

$$IO_{G_{CYK}}(n, m, B) \geq \max \left\{ \frac{(n - 6M - 1)^3}{18\sqrt{M}} \Gamma - 2M\Gamma, n \right\} \frac{1}{B}$$

where Γ denotes the number of distinct right-hand sides in \mathcal{R}_B .

By Theorem 13, we have that for $M \leq c_1 n$, where c_1 is a sufficiently small positive constant value, computations of the CYK algorithm require $\Omega\left(\frac{n^3 \Gamma}{\sqrt{M} B}\right)$ I/O operations. Note that if the considered CFG is such that $\Gamma = \Theta(|\mathcal{R}_B|)$ our lower bound shows a direct dependence of the I/O complexity of execution of the CYK algorithm on the *size* of the considered grammar. Indeed there are many possible grammars for which this is the case. Finally, while the result of Theorem 7 holds for computations with recomputation, it is also possible to obtain an I/O lower bound for schedules in which recomputation is not allowed by modifying the analysis given in Theorem 4.

5.3 On the tightness of the bounds

We refine Valiant's DP-Closure Algorithm [13] to obtain an I/O efficient recursive divide-and-conquer implementation of the CYK algorithm, henceforth referred to as \mathcal{A}_{CYK} . While the CYK algorithm follows the general structure of the Prototype Algorithm 1, and is hence amenable to the approach in [13], the analysis in [13] does not directly apply to the case in which the size of the given CNF cannot be considered as constant with respect to the size of the cache. Our algorithm, \mathcal{A}_{CYK} , manages I/O efficiently by opportunistically iterating over the grammar rules and by removing redundant computations due to grammar rules with the same right-hand side.

Given an input string w of length $n - 1$ and a CFG $(V, \mathcal{R}, \Sigma, T)$ in CNF, we use an $n \times n$ matrix \mathbf{X} to represent the values of the subproblems computed by the CYK algorithm (corresponding to R -vertices as defined in Section 5.2): $\mathbf{X}[i, j + 1]$ corresponds to the solution of subproblem $S(i, j)$, i.e., a sequence of length $|V|$ with binary values, where the index corresponding to variable $A \in V$ represents whether variable A can derive the substring $w_i w_{i+1} \dots w_j$ of the input.

First, \mathcal{A}_{CYK} computes the input values corresponding to subproblems $S(i, i)$ for $1 \leq i \leq n$ (i.e., the values of $\mathbf{X}[i, i + 1]$): Each character in the input string w is compared with the right-hand side of each terminal rule in \mathcal{R}_T , and the appropriate values are stored. The rest of the computation involves executing Valiant's DP-Closure Algorithm (Algorithm 2) using a modified version of Valiant's Star Algorithm (Algorithm 3, called "*Valiant's Star Algorithm for CYK*") (\mathcal{A}_{CYK}^*) which is presented in Algorithm 6.

■ **Algorithm 6** \mathcal{A}_{CYK}^* , Valiant's Star Algorithm for CYK.

```

1: Input  $X$                                 ▷ Top-left and bottom-right are assumed to be fully computed
2: Output  $X$                                 ▷ DP-Closure of  $X$  is computed in-place
3: if  $\dim(X) = 2$  then                      ▷ Root has been fully computed. Update all variable roots
4:   for  $A \in V$  matched with placeholder  $P$  do
5:      $X^A \leftarrow X^P$ 
6:   return
7:  $\begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{23} \\ & \mathbf{X}_{33} \end{bmatrix} \leftarrow \mathcal{A}_{CYK}^* \left( \begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{23} \\ & \mathbf{X}_{33} \end{bmatrix} \right)$ 
8: for  $P \rightarrow BC$  in  $\mathcal{R}_B^\Gamma$  do
9:    $X_{13}^P \leftarrow X_{13}^P + X_{12}^B \cdot X_{13}^C$     ▷ Computed using Matrix Multiply and Accumulate
10:  $\begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{13} \\ & \mathbf{X}_{33} \end{bmatrix} \leftarrow \mathcal{A}_{CYK}^* \left( \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{13} \\ & \mathbf{X}_{33} \end{bmatrix} \right)$ 
11: for  $P \rightarrow BC$  in  $\mathcal{R}_B^\Gamma$  do
12:    $X_{24}^P \leftarrow X_{24}^P + X_{23}^B \cdot X_{34}^C$ 
13:  $\begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{24} \\ & \mathbf{X}_{44} \end{bmatrix} \leftarrow \mathcal{A}_{CYK}^* \left( \begin{bmatrix} \mathbf{X}_{22} & \mathbf{X}_{24} \\ & \mathbf{X}_{44} \end{bmatrix} \right)$ 
14: for  $P \rightarrow BC$  in  $\mathcal{R}_B^\Gamma$  do
15:    $X_{14}^P \leftarrow X_{14}^P + X_{12}^B \cdot X_{24}^C$ 
16:    $X_{14}^P \leftarrow X_{14}^P + X_{13}^B \cdot X_{34}^C$ 
17:  $\begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{14} \\ & \mathbf{X}_{44} \end{bmatrix} \leftarrow \mathcal{A}_{CYK}^* \left( \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{14} \\ & \mathbf{X}_{44} \end{bmatrix} \right)$ 

```

\mathcal{A}_{CYK}^* introduces two modifications over Valiant's Star Algorithm: First, the algorithm iterates over the rules of the grammar with distinct right-hand-sides in lines 8, 11, and 14, calling the subroutine Matrix Multiply and Accumulate with the variable roots corresponding to each rule in the grammar. Second, to avoid computing rules with identical right-hand sides more than once, let \mathcal{R}_B^Γ denote a set of binary rules each having a unique right-hand side from \mathcal{R}_B of maximum cardinality. In this set, the corresponding left-hand sides are replaced with placeholder variables. For any subproblem, once the value corresponding to a

rule in \mathcal{R}_B^Γ is fully computed, it can be used to update the value corresponding to all variable roots producing the same right-hand side through a rule in \mathcal{R}_B . This is done in lines 4 and 5 as the base case implies that all leaves for the subproblem have been computed.

► **Theorem 14.** *Given an input string w of length n and a CFG $(V, \mathcal{R}, \Sigma, T)$ in CNF, algorithm \mathcal{A}_{CYK} decides whether w is a member of the language of the given grammar. When run on a machine equipped with a cache of size M , the number of I/O operations executed by \mathcal{A}_{CYK} can be bound as:*

$$IO_{\mathcal{A}_{CYK}}(n, M) \leq \mathcal{O}\left(\left(\frac{n^3\Gamma}{\sqrt{M}} + n^2\Gamma \log M + n^2|\mathcal{R}_B| + n|\mathcal{R}_T|\right) \frac{1}{B}\right).$$

Proof. When computing $S(i, i)$ for $1 \leq i \leq n$, \mathcal{A}_{CYK} compares each character of input string w with the right-hand side of each terminal rule in \mathcal{R}_T . Doing so incurs at most $\mathcal{O}(n|\mathcal{R}_T|/B)$ I/O operations.

In the following, we use n to denote the dimension of the input (i.e., $n = \dim(X)$). The number of I/O operations executed by Valiant's Star Algorithm for CYK is at most:

$$IO_{\mathcal{A}_{CYK}^*}(n) \leq \begin{cases} \mathcal{O}\left(\frac{|\mathcal{R}_B|}{B}\right) & \text{if } n \leq 2, \\ 4IO_{\mathcal{A}_{CYK}^*}\left(\frac{n}{2}\right) + 4\Gamma(IO_{MMA}\left(\frac{n}{4}\right) + \mathcal{O}(1)) & \text{otherwise} \end{cases} \quad (11)$$

By [13, Lemma 2.1], the number of I/O operations executed by **Matrix Multiply and Accumulate** is:

$$IO_{MMA}(n) \leq \begin{cases} \mathcal{O}\left(\frac{n^2}{B}\right) & \text{if } n^2 \leq M, \\ \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right) & \text{otherwise} \end{cases} \quad (12)$$

Solving the recursion in (11) using (12) yields:

$$\begin{aligned} IO_{\mathcal{A}_{CYK}^*}(n) &\leq \mathcal{O}\left(\frac{n^3\Gamma}{B\sqrt{M}}\right) + 4^{\log(n/\sqrt{M})} \mathcal{O}\left(\frac{M\Gamma}{B} \log M + \frac{M|\mathcal{R}_B|}{B}\right) \\ &\leq \mathcal{O}\left(\frac{n^3\Gamma}{B\sqrt{M}} + \frac{n^2\Gamma}{B} \log M + \frac{n^2|\mathcal{R}_B|}{B}\right) \end{aligned} \quad (13)$$

Finally, by adapting that analysis in [13] we can bound the number of I/O operations executed by the proposed algorithm \mathcal{A}_{CYK} as:

$$IO_{\mathcal{A}_{CYK}}(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n \leq 2, \\ 2IO_{\mathcal{A}_{CYK}}(n/2) + IO_{\mathcal{A}_{CYK}^*}(n) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

Expanding the recurrence, by (13), and adding the cost of processing the input, we have:

$$IO_{\mathcal{A}_{CYK}}(n) \leq \mathcal{O}\left(\frac{n^3\Gamma}{B\sqrt{M}} + \frac{n^2\Gamma}{B} \log M + \frac{n^2|\mathcal{R}_B|}{B} + \frac{n|\mathcal{R}_T|}{B}\right) \quad \blacktriangleleft$$

When $M < c_1 n$, where c_1 is a sufficiently small positive constant, the lower bound in Theorem 13 simplifies to $\Omega\left(\frac{n^3\Gamma}{B\sqrt{M}}\right)$, while the upper bound in Theorem 14 becomes $\mathcal{O}\left(\frac{n^3\Gamma}{B\sqrt{M}} + \frac{n^2|\mathcal{R}_B|}{B} + \frac{n|\mathcal{R}_T|}{B}\right)$. Hence, if $(n|\mathcal{R}_B| + |\mathcal{R}_T|)/\Gamma \in \mathcal{O}(n^2/\sqrt{M})$, the lower-bound asymptotically matches the upper-bound. When disallowing recomputation, a similar analysis shows our upper and lower bounds match under the same condition for $M < c_2 n$, where c_2 is a sufficiently small positive constant.

6 Conclusion

This work contributed to the characterization of the I/O complexity of Dynamic Programming algorithms by establishing asymptotically tight lower bounds for a general class of DP algorithms sharing a common structure of sub-problem dependence. Our technique exploits common properties of the CDAGs corresponding to said algorithms, which makes it promising for the analysis of other families of recursive algorithms, in particular other families of DP algorithms of interest. The generality of our technique is further showcased by the ability to extend it to more complex algorithms, such as the Cocke-Younger-Kasami, for which we provide an (almost) asymptotically tight I/O lower bound and a matching algorithm.

Our analysis yields lower bounds both for computations in which no intermediate value is computed more than once and for general computations allowing recomputation. By doing so we reveal how when the size of the available cache is greater than $2n$ and $o(n^2)$, schedules using recomputation can achieve an asymptotic reduction of the number of required I/O operations by a factor $\Theta(n^2/\sqrt{M})$. This is particularly significant as in many cases of interest (e.g., Matrix Multiplication, Fast Fourier Transform) recomputation has been shown to enable a reduction of the I/O cost by at most a constant multiplicative factor.

Although it is known that recomputation can decrease the space and I/O complexity of certain CDAGs, we are still far from characterizing those CDAGs for which recomputation proves effective. This overarching objective remains a challenge for any efforts aimed at developing a general theory of the communication requirements of computations.

References

- 1 Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
- 2 Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 245–252, 2009.
- 3 Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)*, 59(6):1–23, 2013. doi:10.1145/2395116.2395121.
- 4 Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.
- 5 Gianfranco Bilardi and Lorenzo De Stefani. The I/O complexity of Strassen’s matrix multiplication with recomputation. In *Workshop on Algorithms and Data Structures*, pages 181–192. Springer, 2017. doi:10.1007/978-3-319-62127-2_16.
- 6 Gianfranco Bilardi and Lorenzo De Stefani. The I/O complexity of Toom-Cook integer multiplication. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2034–2052. SIAM, 2019. doi:10.1137/1.9781611975482.123.
- 7 Gianfranco Bilardi and Lorenzo De Stefani. The DAG Visit Approach for Pebbling and I/O Lower Bounds. In *In Proc. FSTTCS 2022*, volume 250, pages 7:1–7:23, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2022.7.
- 8 Gianfranco Bilardi and Enoch Peserico. A characterization of temporal locality and its portability across memory hierarchies. In *International Colloquium on Automata, Languages, and Programming*, pages 128–139. Springer, 2001. doi:10.1007/3-540-48224-5_11.
- 9 Gianfranco Bilardi, Andrea Pietracaprina, and Paolo D’Alberto. On the space and access complexity of computation DAGs. In *Graph-Theoretic Concepts in Computer Science: 26th International Workshop, WG 2000 Konstanz, Germany, June 15–17, 2000 Proceedings 26*, pages 47–58. Springer, 2000. doi:10.1007/3-540-40064-8_6.

- 10 Gianfranco Bilardi and Franco P Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27(2):172–182, 1995. doi:10.1006/JPCD.1995.1080.
- 11 Gianfranco Bilardi and Franco P Preparata. Processor—time tradeoffs under bounded-speed message propagation: Part II, lower bounds. *Theory of Computing Systems*, 32(5):531–559, 1999. doi:10.1007/S002240000131.
- 12 Guy E Blleloch and Yan Gu. Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra. *arXiv preprint*, 2018. arXiv:1809.09330.
- 13 Cary Cherng and Richard E Ladner. Cache efficient simple dynamic programming. *Discrete Mathematics & Theoretical Computer Science*, 2005.
- 14 Rezaul Chowdhury, Pramod Ganapathi, Yuan Tang, and Jesmin Jahan Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 339–350, 2017. doi:10.1145/3087556.3087586.
- 15 Rezaul Chowdhury, Pramod Ganapathi, Stephen Tschudi, Jesmin Jahan Tithi, Charles Bachmeier, Charles E Leiserson, Armando Solar-Lezama, Bradley C Kuszmaul, and Yuan Tang. Autogen: Automatic discovery of efficient recursive divide-8-conquer algorithms for solving dynamic programming problems. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):1–30, 2017. doi:10.1145/3125632.
- 16 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 591–600. Citeseer, 2006. URL: <http://dl.acm.org/citation.cfm?id=1109557.1109622>.
- 17 Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *Proceedings of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 71–80, 2007. doi:10.1145/1248377.1248392.
- 18 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 207–216, 2008. doi:10.1145/1378533.1378574.
- 19 Rezaul Alan Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, 2008.
- 20 John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.
- 21 Lorenzo De Stefani and Vedant Gupta. On the I/O complexity of the CYK algorithm and of a family of related DP algorithms. *arXiv preprint arXiv:2410.20337*, 2024. doi:10.48550/arXiv.2410.20337.
- 22 Erik D Demaine, Andrea Lincoln, Quanquan C Liu, Jayson Lynch, and Virginia Vassilevska Williams. Fine-grained I/O complexity via reductions: New lower bounds, faster algorithms, and a time hierarchy. *arXiv preprint arXiv:1711.07960*, 2017. arXiv:1711.07960.
- 23 Xiangyun Ding, Yan Gu, and Yihan Sun. Parallel and (nearly) work-efficient dynamic programming. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 219–232, 2024. doi:10.1145/3626183.3659958.
- 24 Aaron Dunlop, Nathan Bodendab, and Brian Roark. Reducing the grammar constant: an analysis of CYK parsing efficiency. Technical report, Technical report CSLU-2010-02, OHSU, 2010.
- 25 Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H Campbell. An algorithm for fast edit distance computation on GPUs. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9. IEEE, 2012.
- 26 Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 285–297. IEEE, 1999. doi:10.1109/SFFCS.1999.814600.

- 27 Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, 1992. doi:10.1016/0304-3975(92)90135-3.
- 28 Te Chiang Hu and Man Tak Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982. doi:10.1137/0211028.
- 29 Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. *ACM SIGPLAN Notices*, 51(10):145–164, 2016.
- 30 Mohammad Mahdi Javanmard, Pramod Ganapathi, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. Toward efficient architecture-independent algorithms for dynamic programs. In *High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34*, pages 143–164. Springer, 2019. doi:10.1007/978-3-030-20656-7_8.
- 31 Hong Jia-Wei. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, 1981.
- 32 Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966.
- 33 Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1:14–25, 1971. doi:10.1007/BF00264289.
- 34 Richard Koch, Tom Leighton, Bruce Maggs, and Satish Rao. Work-preserving emulations of fixed-connection networks. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 227–240, 1989.
- 35 Martin Lange and Hans Leiß. To CNF or not to CNF? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8(2009):1–21, 2009. URL: <http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009>.
- 36 J-S Park, Michael Penner, and Viktor K Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on parallel and distributed systems*, 15(9):769–782, 2004. doi:10.1109/TPDS.2004.44.
- 37 Cynthia A Patterson, Marc Snir, and Susan L Graham. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.
- 38 Vijaya Ramachandran. Cache-oblivious computation: Algorithms and experimental evaluation. In *2007 International Conference on Computing: Theory and Applications (ICCTA'07)*, pages 20–26. IEEE, 2007. doi:10.1109/ICCTA.2007.34.
- 39 John E Savage. Extending the Hong-Kung model to memory hierarchies. In *International Computing and Combinatorics Conference*, pages 270–281. Springer, 1995. doi:10.1007/BFB0030842.
- 40 Jacob Scott, Olga Holtz, and Oded Schwartz. Matrix multiplication I/O-complexity by path routing. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 35–45, 2015. doi:10.1145/2755573.2755594.
- 41 Michael Sipser. Introduction to the Theory of Computation. *ACM Sigact News*, 27(1):27–29, 1996. doi:10.1145/230514.571645.
- 42 Xinying Song, Shilin Ding, and Chin-Yew Lin. Better binarization for the CYK parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 167–176, 2008. URL: <https://aclanthology.org/D08-1018/>.
- 43 Shanjiang Tang, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, and Huabei Wu. Easypdp: An efficient parallel dynamic programming runtime system for computational biology. *IEEE Transactions on Parallel and Distributed Systems*, 23(5):862–872, 2011. doi:10.1109/TPDS.2011.218.
- 44 Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A Chowdhury. Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–214, 2015.

49:24 On the I/O Complexity of the CYK Algorithm and DP Algorithms

- 45 Leslie Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 1974.
- 46 F Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 429–435, 1980. doi:10.1145/800141.804691.
- 47 Daniel H Younger. Recognition and parsing of context-free languages in time n^3 . *Information and control*, 10(2):189–208, 1967. doi:10.1016/S0019-9958(67)80007-X.