Università degli Studi di Padova

FACOLTÀ DI DI INGEGNERIA
Corso di Laurea in in Ingegneria Informatica

Tesi di Laurea Magistrale

On the space complexity of DAG computations

Candidato:
Lorenzo De Stefani
Matricola 621842

Relatore:
Ch.mo Prof. Gianfranco Bilardi

Anno Accademico 2011-2012

Contents

List of Figures

Abstract

In this thesis we have studied some issues related to the space complexity of Directed Acyclic Graphs (DAG) computations and in particular the possibility of obtaining a reduction of the amount of memory necessary to the evaluation of a DAG using computations with multiple assessments of the same vertex rather than strictly without recalculations. In the main result of the thesis, we introduce a method to obtain a significant upper bound for the space complexity of a DAG, based on the concept of DAG-vertex separator. By further developing this result according to the divide and conquer paradigm we obtain a decomposition of the DAG through which it is possible to observe a relationship between topological characteristics of the graph and its space complexity.

Introduction

Since the advent of the digital computer and trough the steady and impressive growth of its supporting technology, the memory used in the computing system has accounted for a substantial part of the cost of computing systems. This fact has motivated the study of space-efficient computations aimed to achieve an optimal utilization of the registers of a CPU and/or of the random access memory used in common general-purpose computer.

Even in situations where the cost of memory is negligible, the need for efficient use of the available memory space in computations still arises from the pursuit of performance. Although most parameters are still being improved, there is a general consensus that physical limitations to signal propagation speed and device size are becoming more and more significant [4]. Therefore, in a scenario where access time is bound to increase with the size of memory, the utilization of smaller memory will allow to achieve faster computation thus making space efficiency a crucial objective.

In this thesis we will focus on the analysis of the memory space needed for computations done with straight-line programs in a data-independent fashion which can be modeled by means of a *Computational Directed Acyclic Graph* (CDAG). We will attempt in particular to find significant bounds for the space requirement, possibly pointing out relations between them and some graph-theoretic properties of the DAG.

In Chapter 1 we provide an accurate characterization of the problem under analysis, together with a series of definitions and concepts repeatedly used in the thesis, and we establish the main goals of this work.

In Chapter 2 we discuss the approach based on marking rule introduced by Bilardi, Pietracaprina and D'Alberto [6], as an example of a general framework for analyzing the space complexity of DAGs.

The main result of the thesis is then presented in Chapter 3, where we intro-

duce a new method to obtain a significant upper bound for the space complexity of a DAG based on the concept of DAG-vertex separator. By further developing this result according to the paradigm divide and conquer we obtain a decomposition of the DAG through which it is possible to observe a relationship between topological characteristics of the graph and its space complexity.

In Chapter 4 we go to apply the previous results to the classes of finite genus DAGs and planar DAGs, obtaining quantitative bounds on the space complexity related just to the dimension of the vertices set.

We then conclude in Chapter 5 with remarks on the results obtained with respect to the initial objectives of the thesis accompanied by indications on possible future developments of the presented work.

Chapter 1

Problem definition and introductory concepts

In this first chapter we will present an in-depth analysis of the problem to be explored and lay down the main theoretical definitions and concepts extensively used in this work. We will also introduce the pebbling game model as to produce an effective model of the task at hand. We will then conclude stating the goals for this thesis.

1.1 Problem description

In most computations the memory space available, be it the number of CPU registers or the RAM size, is not sufficient to hold all the data on which a program operates. Thus the same memory locations must be reused or the available space must be increased leading respectively generally to an increase or to a reduction of the number of the necessary computational steps (time) [19].

This study is focused 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).

Definition 1.1 (Straight-line program)**.** A straight-line program is a set of steps each of which is an input step, denoted as (s READ x), an output step, denoted (s OUTPUT i), or a computation step, denoted (s OP $i...k$). Here s is the number of a step, x denotes an input variable and the keyword READ, OUTPUT, OP identify steps where an input is read, an output produced and the operation OP

is performed. In particular at the s-th computation step the arguments to OP are the results produced at steps $i, ..., k$. It is required that these steps precede the sth step, that is $s \geq i, ..., k$.

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

The requirement that each computation step operates on results produced in preceding steps insures 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 processing type) and whose arcs represent data dependencies.

The problem we consider is the optimization of the implementation of a computation which has been specified in terms of a DAG assigning to the implementor essentially two degrees of freedom: the definition of the schedule of execution of the operations, possibly including recalculations, 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.

In particular we will focus on the study of the minimum memory space required for the evaluation of a given DAG, called *space complexity*.

This problem has been extensively developed in literature since the seventies, typically formulated in terms of the so-called *pebble game* (see e.g., [7], [11], [14], [17]) which is presented later.

1.2 Preliminaries

This section provides some basic definitions concerning graph theory and DAG computations which are widely used in the remainder of the work.

1.2.1 Definitions on DAG theoretical properties

Let $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ be a DAG where the directed edges in the set $\overrightarrow{E_V}$ represent data dependencies and the vertices in the set V represent values produced by unit-time operations requiring unitary memory space. We assume that there is no directed loop in $\overrightarrow{G_V}$. Sometimes we will use the lighter notation $\overrightarrow{G}$ instead of $\overrightarrow{G_V}(V, \overrightarrow{E_V})$

when we are referring to a generic DAG without a specific vertex set V associated to it.

We say that two vertices u and v in V are *adjacent* in $\overrightarrow{G_V}$ if there is an edge connecting them. For every directed edge $\langle u, v\rangle$ in $\overrightarrow{E_V}$ 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.

A *path* l between two distinct vertices u and v in 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 $l = (c_0 = u, c_1 \ldots, c_{m-1}, c_m = v)$ where $\langle c_{i-1}, c_i\rangle$ or $\langle c_i, c_{i-1}\rangle$ are edges in $\overrightarrow{E}$ for $i = 1, \ldots, m$ and $c_i \neq c_j$ for all $i \neq j$. We say that a path l_d 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 $\overrightarrow{G}$. The set of all ancestors of v will be denoted as $an(v)$.

The *in-degree* (resp., *out-degree*) $deg^-(v)$ (resp., $deg^+(v)$) of a vertex v in $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ is the number of its predecessors (resp., successors) $deg^-(v) = |pa(v)|$(resp., $deg^+(v) = |ch(v)|$. Vertices of in-degree (resp., out-degree) 0 constitute the set $I \subset V$ of the inputs (resp., $O \subset V$ of the outputs) of the DAG.

The *total-degree* of a vertex v corresponds to the total number of its adjacent vertices: $deg(v) = deg^-(v) + deg^+(v)$. We shall refer to the maximum degree of a DAG $\overrightarrow{G}$ as $Deg(\overrightarrow{G}) = \max_{v \in V} (deg^-(v) + deg^+(v))$.

Given a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ we say that $\overrightarrow{G}$ is *connected* if for every couple of distinct vertices u and v in V there is either a directed path from u to v or vice versa. If this condition is not verified, but there are still path connecting couple of distinct vertices u and v, $\overrightarrow{G_V}$ is said to be *weakly connected*. DAGs that are not weakly connected are said to be *not connected*.

The notation $G(V, A)$ is used to refer to the *undirected graph* defined over the node set V with the set of undirected arcs A.

1.2.2 Definitions concerning DAG computations

A *computation* (or *schedule*) of $\overrightarrow{G}$ specifies a particular scheduling of the operations associated with its vertices, which satisfies data dependencies, and a

particular memory management.

In this work we study DAG computations on the RAM model with a memory of unbounded size whose cells are addressed by the natural numbers starting from 0 [19]. A standard computation of a DAG $\overrightarrow{G}$ starts with the values of all input vertices 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, and the memorization of the value computed for v in memory.

In general a vertex can be evaluated more than once, however without loss of generality it is useful to restrict the analysis to *parsimonious computations* where output vertices are evaluated exactly one time and between two consecutive evaluations of the vertex v at least one successor $u \in ch(v)$ must be evaluated. We will thus assume that after the value of vertex v is produced, it remains in memory until the last time a node $u \in ch(v)$ is evaluated before the re-calculation of v. If v is an output we can safely assume that its value may be removed from the memory immediately after its unique evaluation.

The space required by a computation corresponds to the maximum number of values which are stored in memory at any one time during the computation. The *space complexity* of a DAG $\overrightarrow{G}$, denoted by $S(\overrightarrow{G})$ is defined as the minimum space strictly required by any *standard computation* for $\overrightarrow{G}$.

Since for standard computations all inputs need to be in memory at the start of the computation for every DAG the following lower bound holds:

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

We will consider also another class of computations, called *free-input computations*, which start with an initially empty memory, and every time an input value is needed it can be produced invoking a special load instruction. It is easy to argue that the space complexity defined over the free-input computations of $\overrightarrow{G}$, which will be denoted as $S^{free}(\overrightarrow{G})$ is never higher than $S(\overrightarrow{G})$. I should be remarked how the $S^{free}(\overrightarrow{G})$ corresponds to the space measure captured by the *Pebble Game* model [6].

On the other side the time required by a computation corresponds to the number of vertex evaluation performed within it. It is easy to see how for any n-vertex DAG there are computations which require no more than n steps.

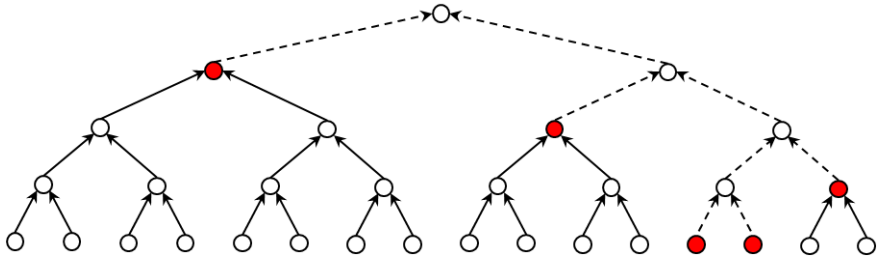

Figure 1.1: Example of pebble game played on a binary tree DAG

1.3 The pebble game

In this section we will briefly present the *pebble game* (also called *black pebble game* [11]) which is a simple yet useful model which allows us to study various types of computations and enables us to investigate the time and space requirements for the evaluation of a DAG, and the relation between them. This method is also an useful tool to get a better understanding of the problem of the space complexity estimation for DAG, through a simple yet powerful approach.

The pebble game is a game played on directed acyclic graphs which captures the dependencies of straight-line programs. In the pebble game pebbles are placed on vertices of a DAG in a data-independent order to indicate that the value associated with a certain node is currently stored in memory.

The rules for the pebble game are the following:

- (Initialization) A pebble can be placed on an input vertex at any time
- (Computation step) A pebble can placed on (or slided to) any non-input vertex only if all its immediate predecessors carry pebbles
- (Pebble deletion) A pebble can be removed at any time
- (Goal) Each output vertex must be pebbled at least once

The placement of a pebble on an input vertex 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. The removal of a pebble models the deletion or the overwriting of the value previously stored in memory corresponding to the vertex carrying a pebble.

Allowing pebbles to be placed on input vertices at any time reflects the assumption that inputs are readily available; this 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 situation in which all input values must actually reside in memory. The model, however, maintains however a high degree of interest since it provides some kind of lower bound to space complexity 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 models the natural requirement that an operation can be performed only if all arguments of the operation are available and located 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.

The execution of the rules of the pebble game on the vertices of a DAG $\overrightarrow{G}$ is called a *pebble strategy*. It is easy to argue that each pebble strategy corresponds to a free-input computation for $\overrightarrow{G}$. In particular each step of the strategy is associated to each placement of a pebble, ignoring steps on which pebbles are removed, and numbered consecutively from 1 to T, where T corresponds to the time required by the strategy. The space, S, used by a pebbling strategy is the maximum number of pebbles it uses. The goal of the pebble game is to pebble a graph with values of space and time that are minimal, that is, the necessary space cannot be reduced for the given value of time and vice versa. In our analysis we will focus mainly on the optimal (minimal) space requirements.

It should be remarked that, in general, it is very hard to determine the minimum number of pebbles needed to pebble a graph and, in order to achieve the optimal result, rather than a general approach, specific pebbling strategies tailored on the particular DAG structure are to be devised [19].

In particular, in terms of the traditional hierarchy of complexity classes, the problem of finding the minimum number of pebbles needed to pebble a DAG can be modeled as a language consisting of strings each of which contains the description of a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ a vertex $v \in V$ and an integer S with the property that v can be pebbled with S or fewer pebbles. The language of these strings is 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

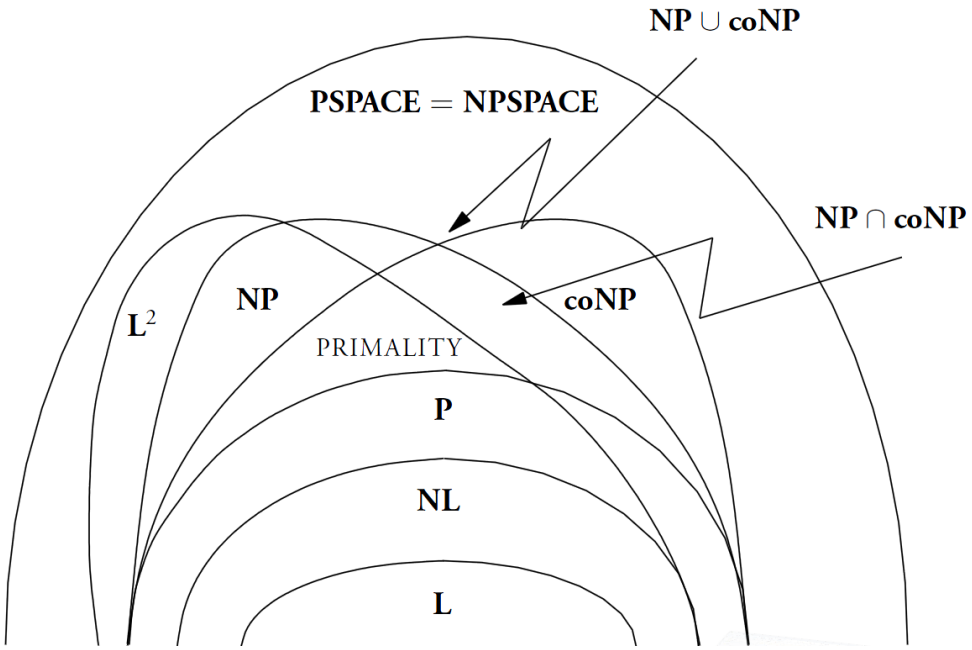

Figure 1.2: The relationships among complexity classes

problem can be reduced to a PSPACE-complete problem in polynomial time by a Turing machine. 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 [19].

Besides the study of the space complexity of DAGs, variations of the basic pebble game have been successfully used to analyze problems linked to the access complexity in two-level and hierarchical memory. In particular the Red-Blue pebble game by Hong and Kung [12] remains to this date the main point of departure of most lower bound analysis for hierarchical memory performance.

1.4 Goals of the thesis

As discussed in the first part of this chapter, the optimization of memory space for programs that can be represented through DAGs is an important but hard task. This difficulty is mainly related to the need of taking into account the computations available using recalculations. Over the years, several results have been proposed in literature to show how a clever use of recalculations allows to achieve minimum space complexity for DAG computations otherwise not reachable with

single evaluation of each vertex. This observation leads to a trade-off between the number of operations executed T and the space used S, generally expressed in the form $ST = \Omega(f(n))$, where $f(n)$ is a function of the input size of the problem n, related to the specific problem at hand.

On the other hand, the possibility of re-evaluating some values conveniently during the steps of a computation opens the door to the need of devising specific strategies to meet the particular features of each DAG in order to meet the optimal memory requirement.

This approach, however, proves to be very time consuming and leads to very specific results which can not generally be extended to wider classes of DAGs.

These difficulties lead to the fact that, in general, given a DAG $\overrightarrow{G}$ it is hard to give a good estimate of its space complexity and to determine whether the use of recalculations may prove useful to achieve a better memory usage.

This work is aimed to find a relation between some graph theoretical properties of a given DAG $\overrightarrow{G}$ and the fact that $\overrightarrow{G}$ may or may non benefit, in terms of minimal space needed for its execution, from the employment of computations with recalculations. In particular, once these properties are found, we expect to be able to obtain some new and significant bounds to the space complexity of DAGs. In addition, this result will allow us to obtain a general criteria to estimate the space complexity founded on the DAG structure and properties rather than its peculiar possible computations.

Chapter 2

The role of the recalculation in DAG computations

In this chapter we will discuss some results already known in literature, concerning bounds on the space complexity of DAGs. In particular, we will distinct observations related to computations in which recalculations are never performed from those in which they are.

We will then try to get a better understanding of why recalculations may prove so helpful in some situations and present a technique especially devised to estimate a lower bound on space complexity of DAGs by capturing the memory space used by computations with recalculations.

2.1 Computations strictly without recalculations

Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ it is always possible to devise a computation for which the operations associated to each node are executed exactly one time. This class of computations is referred as *strictly without recalculations* and corresponds to non *re-pebbling strategies* in the pebble game. Any such computation is particularly useful in all those situations in whom the main priority is given to achieving the minimum execution time n.

The key observation concerning this class of computations is that every time the value of a vertex is loaded in memory or calculated it must remain available until each one it successors has been evaluated, still among these computations some will provide a better memory usage than others, using a clever ordering of vertex evaluations.

Bilardi and Preparata proposed an approach to estimate the lower bounds on memory usage achievable using computations strictly without recalculations based on the notion of *dicothomy-width* [3]. This results is of particular interest since it shows an interesting relation between a property of the topological structure of a DAG and the best minimum memory requirements achievable using computations strictly without recalculations.

Definition 2.1 (Closed subset)**.** Given a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, a subset $W \subseteq V$ is said to be *closed* if whenever $v \in W$ then $an(v) \in W$.

In the present context it is important to notice that, because of the dependencies between the vertices of a DAG, in each step i of a computation strictly without recalculations the set of vertices evaluated during the steps $1, \dots i$ is always a closed subset of V.

Definition 2.2 (Closed dichotomy)**.** Given a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, and a subset $W \subseteq V$, let $B_{out}(W)$ denote the set of vertices in W that have a successor in $V \setminus W$. The closed dichotomy size of $\overrightarrow{G}$ is the function β of the integer value $w \in \{0, \dots, |V|\}$ defined as:

$$\beta(w) = \min\{|B_{out}(W)| : W \subseteq V,\ W \text{ closed},\ |W| = w\}.$$

The closed dichotomy size, intuitively, indicates that after w values have already been computed, at least $\beta(m)$ of them will be reused in further computational steps ant should therefore be kept in memory. Using this notion we can thus obtain a lower bound on the memory space necessary for computations in which recalculations are not allowed $S^{wr}(\overrightarrow{G_V})$ as:

$$S^{wr}(\overrightarrow{G_V}) = \max_{w \in \{0,\dots,|V|\}} \beta(w). \tag{2.1}$$

The closed dichotomy proves effective since it captures in B_{out} the presence of *communication vertices* which have already been evaluated but that still have at least one successor which has not yet been computed [5].

The study of this class of computations is of note as the minimum memory space needed by them marks the border of the optimum performance achievable without recalculations and is therefore the standard of comparison for computations with multiple vertex-evaluation.

Obviously only those schedules that offer better performance than $S^{wr}(\overrightarrow{G_V})$ will be considered of interest and their usefulness will be directly related to the gap from the optimum performance achievable without recalculations.

We can therefore say that $S^{wr}(\overrightarrow{G_V})$ is actually an upper bound for the space complexity of $\overrightarrow{G_V}$:

$$S(\overrightarrow{G_V}) \leq S^{wr}(\overrightarrow{G_V}). \tag{2.2}$$

If for some DAG this bound is tight, it can be concluded that the execution with recalculations does not lead to any benefit in terms of reduction of memory space necessary for the calculation of $\overrightarrow{G_V}$.

2.2 The importance of the recalculation

Through the next sections we will focus on computations which allow recalculations and we will try to understand how and why they can prove useful for some DAGs. However, a legitimate doubt concerning the usefulness and the advisability of designing computations with a number of steps higher than the minimum may arise.

In the introduction, we hinted that the optimization of memory utilization has been strongly motivated by the cost of the memory component in computing system. This cost should however be intended just in terms of the economic cost of components. In particular in the context of embedded programming, situations may arise in which for project needs the physical space for memory may be strongly restricted, and optimization of memory management becomes then crucial in order to perform some function, even at the cost of a considerable increase of execution time.

Another important consideration descends from the fact that although most parameters are still being improved, there is an emerging consensus that physical limitations are becoming increasingly significant [**?**]. In particular, concerning memory hierarchies, the gap between the access time to memory layers close to the processing units (register, cache) and to the slower levels (RAM, disc) is expected to grow larger and larger. In such context, a better management of the fastest memory levels, even if achieved at the cost of a substantial increase of the number of computational steps, may still provide better performances than computation with less operations, but needing to access a wider memory space.

Recalculations may also prove very useful in the context of parallel and distributed computing. The general approach adopted whenever more than one computing unit is available, is to divide the workload equally between them and having them collaborate among themselves. However, in a model in which the communication time between computation units, due to physical limits, cannot be considered instantaneous and constitutes a bottleneck for the overall performance, it will actually make more sense to repeat some calculations locally, in order to minimize the communication between different CPUs.

Thus, the execution of recalculations brings a degree of flexibility which may prove useful in many practical situations and may present a different approach to many traditional problems.

2.3 The marking rule approach

In this section, we will present a method to obtain a general lower bound for DAG space complexity based on a framework developed by Bilardi, Pietracaprina and D'Alberto [6], that allows to model arbitrary executions by suitable permutations of the vertices, where each operation appears exactly once, while maintaining a grip on space requirements.

2.3.1 Main theorem

If we rule out multiple executions of the same operation, the computations of a DAG are in one-to-one correspondence with the *topological orderings of its vertices*. In particular, given a specific topological ordering $\phi = \phi_1, \ldots, \phi_n$, consistently with what discussed in section 2.1, after the execution of vertex ϕ_i at step i, all the vertices $v \in \{\phi_1, \ldots, \phi_i\}$ with at last one successor in $\{\phi_{i+1}, \ldots, \phi_n\}$ must be in memory. It may also be noted that $pa(\phi_i) \subseteq \{\phi_1, \ldots, \phi_i\}$ and that the subset of vertices $\{\phi_1, \ldots, \phi_i\}$ is always closed.

It is interesting to note how for all connected DAGs there will be exactly one possible topological ordering of its vertices, and thus it will be possible to calculate the exact amount of memory space necessary for schedules strictly without recalculations in linear time.

The possibility to repeat operations, however, 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 this method, the authors aim to show a correspondence between each possible computation of the DAG to a permutation of its nodes, generally not corresponding to a topological ordering, using a *marking rule* that is a criterion to associate to each vertex $v \in V$ a family of subsets of it successors.

In particular, a marking rule for a given n-vertex DAG $\overrightarrow{G_V}$ is any function $f : V \to 2^{2^V}$ for which:

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

Given a linear arrangement $\phi = \phi_1 \phi_2 \dots \phi_n$ of all the vertices in V so that $\{\phi_i : 1 \leq i \leq n\} = V$. ϕ is a f-marking for a marking rule f iff 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 defined as the set $B^f_\phi(i)$ of all the vertices $v \in V \backslash O$ that satisfy the following properties:

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

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

Is thus possible to shown a relation between the space complexity of the free-input computations of a DAG $\overrightarrow{G_V}$ and the size of the boundaries of its f-marking. Let $F_{\overrightarrow{G}}$ denote the set of marking rules for $\overrightarrow{G_V}$ and $\Phi(f)$ the set of f-markings of $\overrightarrow{G_V}$.

Theorem 2.1 (Lower bound for space complexity)**.**
The space complexity of the free-input computations of $\overrightarrow{G_V}$ is:

$$S_{free}(\overrightarrow{G_V}) \geq \max_{f \in F_{\overrightarrow{G}}} \min_{\phi \in \Phi(f)} \max_{1 \leq i \leq n} \left| B^f_\phi(i) \right|. \tag{2.3}$$

Proof. Consider an arbitrary marking function $f \in F_{\overrightarrow{G}}$ and a T-step free-input parsimonious computation C for $\overrightarrow{G_V}$. Let v_t be the vertex evaluated at step t of C, for $1 \leq t \leq T$. It is possible to obtain the corresponding f-marking of C $\phi = \phi_1 \phi_2 \dots \phi_n$ by sweeping backward the steps of the computations 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 $\overrightarrow{G_V}$. 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 C. Let $v \in B^f_\phi(i)$. The definition of $B^f_\phi(i)$ and the fact that the computation 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 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 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 $\phi \in \Phi(f)$ and by maximizing over all possible $f \in F_G$. □

Note that the lower bound obtained is generally not tight. In fact, as explained in the demonstration, 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, is not yet possible to conclude that all the nodes that are in memory at that step of the computation will actually appear in the boundary $B^f_\phi(i)$.

Another circumstance, which can lead to the lower bound being non-strict is given by the fact that the instants of computation in which the vertices are marked, and that are actually used to build the f-marking, do not correspond to the points of the computation for which it has the greatest need for space.

This means that all possible computations C for $\overrightarrow{G_V}$ will require memory space satisfying the lower bound obtained by 2.3, but it will not always be possible to produce a schedule that matches exactly the value given by 2.3.

Please note that the fundamental difference between topological permutations and the generic f-marking obtained from a certain computation C using a specific marking rule f is that, while in the first each node must appear before all of his successors, in the second the only constraint that arises on the occurrence of a node in the permutation is that this appears before at least one of its successors.

One disadvantage of this approach is given by the high number of possible

marking rule to be analyzed $\left|F_{\overrightarrow{G}}\right|$. Among these, however, one of particular importance is the *singleton marking rule* $f^{(sing)}$ defined as:

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

The application of the conversion from computations into $f^{(sing)}$-makings allows to find all f-markings obtainable using any other marking rule $f \in F_{\overrightarrow{G}}$ and for any f-marking $\phi \in \Phi(f)$ will be:

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

2.3.2 A criterion for identifying DAG that do not benefit form recalculations

The computations used for the evaluation of a CDAG $\overrightarrow{G_V}$ without recalculations correspond each to the possible topological orderings of the vertices in V, indicated as Φ_V. It will therefore be possible to evaluate the memory space required for the evaluation of $\overrightarrow{G_V}$ without recalculations $S^{wr}(\overrightarrow{G_V})$ as:

$$S(\overrightarrow{G_V}) \leq \min_{\phi \in \Phi_V} \max_{1 \leq i \leq n} |B_{\phi}(i)| = S^{wr}(\overrightarrow{G_V}), \tag{2.4}$$

where $B_{\phi}(i)$ will be consisting of all the vertices in the prefix $\phi_1 \phi_2 \dots \phi_i$ which have at least one successor in the suffix $\phi_{i+1} \dots \phi_n$.

As discussed before, the minimum memory space necessary for computations strictly without recalculations constitutes a demarcation point to determine if the execution of recalculations can be effectively useful for lowering the minimum memory space necessary. From 2.3 and 2.4 we can obtain:

$$\max_{f \in F_{\overrightarrow{G}}} \min_{\phi \in \Phi(f)} \max_{1 \leq i \leq n} \left| B_{\phi}^{f}(i) \right| \leq \min_{\phi \in \Phi_V} \max_{1 \leq i \leq n} |B_{\phi}(i)|.$$

Proposition 2.2.
Given a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, if it is verified:

$$\max_{f \in F_{\overrightarrow{G}}} \min_{\phi \in \Phi(f)} \max_{1 \leq i \leq n} \left| B_{\phi}^{f}(i) \right| = \min_{\phi \in \Phi_V} \max_{1 \leq i \leq n} |B_{\phi}(i)|, \tag{2.5}$$

this implies that the optimal space complexity can be achieved by computations without recalculations.

Proof. The demonstration of this proposition is obtained directly from the properties of the lower bound identified in Theorem 2.1. □

In particular, the previous proposition implies that if there is a marking function $f \in F_{\overrightarrow{G}}$ for which it is:

$$\min_{\phi\in\Phi(f)} \max_{1\le i\le n} \left|B^f_\phi(i)\right| = \min_{\phi\in\Phi_V} \max_{1\le i\le n} |B_\phi(i)|$$

then again this implies that the optimal space complexity can be achieved by computations without recalculations.

The lower bound 2.3 can also be used in order to obtain an estimate of the order of magnitude of the maximum obtainable reduction in terms of necessary memory space for computations using recalculation with respect to those strictly without recalculation, indicated as $\zeta(\overrightarrow{G_V})$:

$$\zeta(\overrightarrow{G_V}) = O\left(\frac{\max_{f\in F_{\overrightarrow{G}}} \min_{\phi\in\Phi(f)} \max_{1\le i\le n} \left|B^f_\phi(i)\right|}{\min_{\phi\in\Phi_V} \max_{1\le i\le n} |B_\phi(i)|}\right). \tag{2.6}$$

2.3.3 A criterion for composed DAG analysis

The result of Theorem 2.1 provides also the means to assess the space complexity of a DAG through the analysis of its sub-DAGs. In particular, let $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ be a DAG and $\overrightarrow{G_{V'}}(V', \overrightarrow{E_{V'}})$ a sub-DAG of $\overrightarrow{G_V}$, where $V' \subseteq V$ and $|V| = n \ge m = |V'|$. Any free-input computation of $\overrightarrow{G_V}$ includes (at least) one free-input computation of $\overrightarrow{G_{V'}}$, hence $S_{free}(\overrightarrow{G_V}) \ge S_{free}(\overrightarrow{G_{V'}})$.

In terms of marking rules, it is easy to see that every marking rule $f_{\overrightarrow{G_{V'}}}$ for $\overrightarrow{G_{V'}}$ can be extended to a corresponding marking rule $f_{\overrightarrow{G_V}}$ for $\overrightarrow{G_V}$ such that $f_{\overrightarrow{G_V}}(v) = f_{\overrightarrow{G_{V'}}}(v)$ for every $v \in V'$. This leads to:

$$\max_{f\in F_{\overrightarrow{G_V}}} \min_{\phi\in\Phi(f_{\overrightarrow{G_V}})} \max_{1\le i\le n} \left|B_\phi^{f_{\overrightarrow{G_V}}}(i)\right| \ge \max_{f\in F_{\overrightarrow{G_{V'}}}} \min_{\phi\in\Phi(f_{\overrightarrow{G_{V'}}})} \max_{1\le i\le n} \left|B_\phi^{f_{\overrightarrow{G_{V'}}}}(i)\right|. \tag{2.7}$$

The above considerations somehow suggest that meaningful lower bounds on the space complexity of a DAG are provided by marking rules that are carefully tailored to the DAG structure and, in particular, that bring forward the presence of space demanding sub-DAGs [6].

In Chapter 3 we will see how the proposed method based on the concept of separation provides a general framework for analyzing the space complexity of DAGs, referring only to their underlying undirected graph.

2.4 Examples of significant DAGs

In this section we will propose some DAGs particularly significant in order to show whether or not the execution of recalculations allows to obtain benefits in terms of reduction of the minimum memory space required for their evaluation.

2.4.1 DAGs that do not benefit from the performance of recalculations

Tree DAG

Tree DAGs are a particular class of planar DAGs for which every vertex $v \in V$ has $deg^-(v) = 1$, except one output vertex u (the *root*) for which $deg^-(u) = 0$, and all the edges are oriented from each vertex to its only successor. The input vertices of a tree DAG, also called *leaves*, are those for which $deg^+ = 0$.

In the study of the marking rule approach we pointed out that the main difference between topological orderings and the general f-markings is that, while in the first each vertex should occur before all its successors, for general f-markings a vertex must occur before at least one of its successors. However, since all vertices have just one successor, all possible f-markings will actually be topological orderings.

Thus the condition of Proposition 2.2 is verified and it is possible to conclude that tree DAGs will never benefit of a reduction of the minimum necessary memory space by using computations with multiple evaluations of the same vertices.

In particular, through the marking rule approach, it can be shown that the space complexity of complete balanced binary tree DAG (see Figure 1.2) $\overrightarrow{G_V}$ with m leaves will be $S_{free}\left(\overrightarrow{G_V}\right) \geq \log_2 m + 1$, which is tight as shown in [16].

Pyramid

A pyramid DAG can be obtained by taking the half of a $m \times m$ *mesh graph* above, including the nodes on the main diagonal, and directing all the edges toward the upper-right corner of the mesh. The DAG will have $|V| = n$ vertices with m

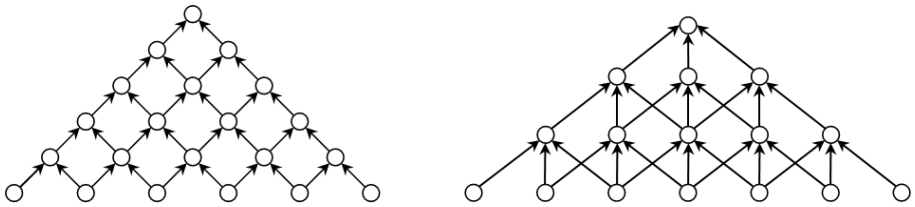

Figure 2.1: Pyramid DAG and 3-pyramid DAG

inputs (the vertices of the mesh diagonal) and one output vertex (the node on the upper-right corner of the mesh) (see Figure 2.1a).

The space complexity of a pyramid DAG $\overrightarrow{G_V}$ can be evaluated by reformulating the argument in [7], using the marking rule approach and in particular the singleton marking rule $f^{(sing)}_{\overrightarrow{G_V}}$. Let $\phi = \phi_1\phi_2 \dots \phi_n$ be an arbitrary $f^{(sing)}$-marking for $\overrightarrow{G_V}$ and let $\phi_1\phi_2 \dots \phi_j$ be the smallest prefix that contains all inputs (hence ϕ_j must be an input). Among the possible directed paths from ϕ_j to the output, there will be at least one path π whose vertices must all be included in the suffix $\phi_{j+1} \dots \phi_n$, and regardless of how π was chosen there will be at most m paths that go from the leaves to vertices in π and intersect only in π. Since any such path starts from a leave in the prefix and ends with a vertex in the suffix, it must contain a vertex in $B_\phi^{f^{(sing)}_{\overrightarrow{G_V}}}(j)$ Theorem 2.1 leads to obtain:

$$S_{free}(\overrightarrow{G_V}) \geq m.$$

Since it will be possible to produce computations without recalculations that evaluate $\overrightarrow{G_V}$ using memory space m (e.g., the bottom up execution by levels starting from the leaves), we can therefore conclude that the bound previously shown is tight and that the execution of recalculations does not grant a reduction in the amount of minimum memory space necessary for the assessment of pyramid DAGs.

The previous considerations can be extended to the class of r-pyramid DAGs with respect to which the case discussed above is a 2-pyramid (see Figure 2.1b for an example of 3-pyramid) [18]. In particular, using the same procedure based on the analysis of $f^{(sing)}$-marking, it is possible to conclude that the space complexity is always equal to the number m of the leaves, and that therefore there are never benefits obtainable using recalculations.

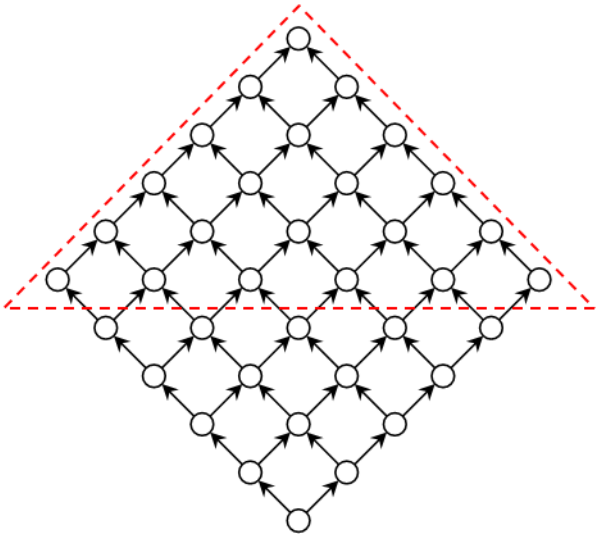

Figure 2.2: Diamond DAG

Diamond

Diamond DAGs are an example of composed DAG which can be obtained by gluing together two $m\,(m+1)\,/2$-vertex pyramids coalescing the corresponding inputs in one node. The diamond DAG will have one input vertex (the former output of the lower pyramid) and one output (the output of the upper pyramid) and all the direction of the edges of the lower pyramid will be reversed. Since one such DAG $\overrightarrow{G_V}$ will have an $m-$based pyramid as a sub-DAG rule 2.7 combined with Theorem 2.1 implies $S_{free}(\overrightarrow{G_V}) \geq m$.

Again, since it is possible to find computations without recalculations which require exactly m memory spaces, it is possible to conclude that the previous bound is tight and the optimal space complexity is achievable without the need to employ recalculations.

This case can be further generalized to $r \times c$ directed mesh, for which it can be shown that the space complexity $\min(r, c)$ can again be achieved by computations without recalculations.

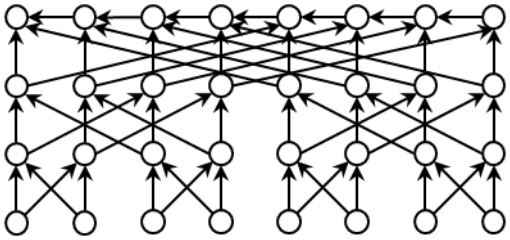

Figure 2.3: FFT DAG with 8 inputs

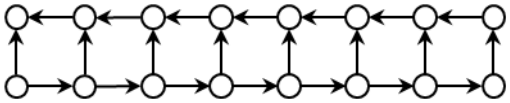

Figure 2.4: Snake-like DAG

2.4.2 DAGs that benefit from the performance of recalculations

FFT

An FFT DAG with m inputs (see Figure 2.3) and m outputs has the property that the set of directed paths from input vertices to an output vertex forms a complete balanced binary tree with m leaves. Thus, any such DAG $\overrightarrow{G_V}$ contains m copies of a complete balanced binary tree with m leaves as a sub-DAG. Relation 2.7, combined with the previous result obtained for binary tree space complexity, allows to conclude $S_{free}\left(\overrightarrow{G_V}\right) \geq \log_2 m + 1$.

It is possible to show a computation which actually matches the bound above by evaluating each of the binary tree sub-DAGs one at a time. This strategy, however, employs the use of recalculations: in particular, besides output vertices which are evaluated just one time, a vertex v for which the shortest directed path to an output vertex has length i will be evaluated 2^{i-1} times [19].

On the other hand, sticking to computations without recalculations, the minimum memory space required will be $S^{wr}\left(\overrightarrow{G_V}\right) \geq m$.

Snake-like DAG

The n-vertex Snake DAG has been obtained by taking an $\frac{n}{2} \times 2$ directed mesh and reversing the direction of the arcs in the second horizontal line. This modifications makes the DAG connected and thus it is possible to evaluate the minimum space needed for computations strictly without recalculations through the analysis of

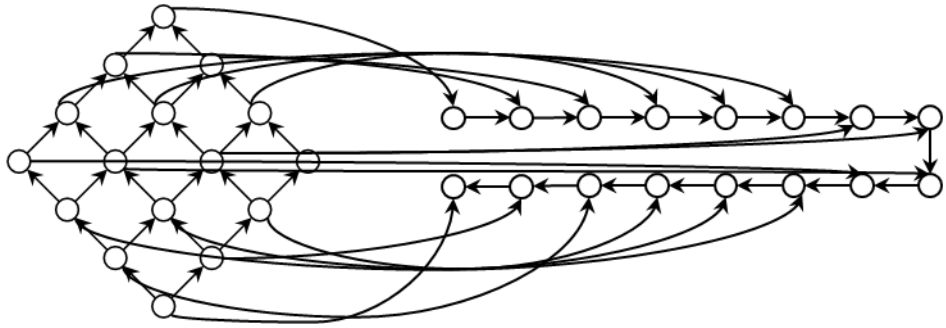

Figure 2.5: DAG Diamond + Array

the total topological ordering of the vertices as $S^{wr}\left(\overrightarrow{G^{B1}}\right) \geq \frac{n}{2}$.

It is possible, however, to find a computational strategy for which at most two results reside in memory at the same time in any of the steps of the computation.

With reference to the pebbling game model, we can see that any vertex in the first row can be pebbled using just one pebble, by discarding the predecessor value as soon as it has been used, while the vertices of the second column will need both of their predecessors to be pebbled in order to be evaluated. In particular, the vertex v in the upper right corner (the first vertex of the upper row) will need just the vertex in the bottom left corner, and will thus be evaluable using at most one pebble. To proceed further in the computation of v successor u, the value associated with v should be kept in memory, while the other predecessor of u in the first row shall be evaluated using one more pebble, this way u will be evaluated using just two pebbles as well. By repeating this pattern, it is easy to see that it will be possible to evaluate the entire DAG using just two pebbles.

Since it is clearly not possible to evaluate the DAG using just one pebble, we can conclude that the bound $S_{free}\left(\overrightarrow{G^{B1}}\right) \geq 2$ is tight.

This is a nice example of a situation in which the employment of recalculations allows to obtain a consistent reduction of the memory space needed from $O(n)$ to $O(1)$. It is also interesting to observe ho in this case the number of operations needed for the re-pebbling strategy is $O(n^2)$ while the execution without recalculations requires just n vertex evaluations.

Diamond + Array

Let $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ be a $n = 2m$ vertex DAG formed by an m-vertex diamond $\overrightarrow{G_D}(D, \overrightarrow{E_D})$ and an m-vertex array $\overrightarrow{G_L}(L, \overrightarrow{E_L})$ connected as follows. Let $v_1 v_2 \dots v_m$

be a topological ordering of $\overrightarrow{G_D}$ and $u_1 u_2 \dots u_m$ be the unique topological ordering of $\overrightarrow{G_L}$. The set of directed edges $\overrightarrow{E_V}$ will be constituted by $\overrightarrow{E_D} \cup \overrightarrow{E_L}$ plus the edges $\langle v_i, u_{m-i+1} \rangle$. The resulting DAG will have one input vertex (the input vertex of the diamond DAG) and one output (the output of the array DAG).

Is easy to see that the space complexity of $\overrightarrow{G_V}$ is determined by the diamond component, and using rule 2.7 one can show $S_{free}\left(\overrightarrow{G_V}\right) \geq \sqrt{m}$, which is tight. However, if we consider computations strictly without recalculations, we see that each of them will have a suffix corresponding to the topological ordering of the vertices of L. Since there will be one edge directed to a vertex in the suffix from a vertex in the prefix $\langle v_i, u_{m-i+1} \rangle$, $B_\phi^{f^{(sing)}_{\overrightarrow{G_V}}}(m) = m^2$ and thus $S^{wr}\left(\overrightarrow{G_V}\right) \geq m$.

Once again the use of recalculation allows to obtain a reduction of order $(O\sqrt{m})$ in terms of the memory space required for the evaluation of the DAG.

Chapter 3

An estimation of DAG space complexity through separators analysis

In this chapter the main result of this work is presented. We will describe a divide-and-conquer technique which allows to find significant upper bounds for the space complexity of generic computations of DAGs. In order to achieve this result, we will exploit the concept of separator and tree decomposition for the undirected graph extractable from a DAG.

3.1 Separators in graphs and DAGs

In literature, the concept of the separator for graphs, and the many definitions connected to it, are usually introduced with reference to undirected graphs with weights assigned to nodes and arcs. In our presentation, we will however refer to the basic model provided by the pebble game, in which the operations associated with each vertex of the DAG produce a value which takes up unit memory space. Thus, without loss of generality, we will assign unitary weight to each vertex and each edge of the graph.

3.1.1 Definitions for undirected graphs

The basic idea behind the separator concept is to remove *few* vertices or edges and separate the original graph in pieces whose size is *balanced* with respect to

the original graph, in that the number of vertices for each of the pieces obtained is at most a fraction of the original graph.

Definition 3.1 (Node Separator)**.** Given a graph $G_V(V, A_V)$, we define a *node separator* for G a subset $X \subseteq V$ which splits the graph in two parts L and R such that there are no edges in E which join a vertex in L with a vertex in R. The separator X generates a partition (L, X, R) for V. The *cost of a node separator* $c(X)$ is given by the number of nodes removed in X ($c(X) = |X|$).

Definition 3.2 (Separator Balance)**.** Consider a node separator X that generates the partition (L, X, R) for V. The *balance* of X with respect to G is given by:

$$b(X_G) = \frac{\min\{|L|, |R|\}}{|V|}.$$

$b(X_G)$ can assume values in the interval $\left[0, \frac{1}{2}\left(1 - \frac{|X|}{|V|}\right)\right]$.

A node separator X for G is *b-balanced* if it achieves a balance of b.

Note that if the removal of a separator splits the graph in more than two connected components, these pieces can be partitioned in two classes so as to achieve a desirable balance.

Definition 3.3 (Node Cut Ratio of a Separator)**.** Consider a separator X that generates the partition (L, X, R) for V. The node cut ratio of X with respect to G is given by:

$$r(X_G) = \frac{|X|}{\min(|L \cup X|, |X \cup R|)}.$$

The **sparsest node cut ratio** of a graph $R(X_G)$ is the lower cut ratio achievable by any node separator of the graph.

We will now try to state similar definitions for DAGs.

3.1.2 Definitions for DAGs

While considering DAGs, the orientation of the edges makes necessary to specify some new definitions concerning separators.

Definition 3.4 (DAG-vertex separator)**.** Given a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, we define a *DAG-vertex separator* for $\overrightarrow{G_V}$ a subset $X \subseteq V$ which splits the graph in two parts L and R, such that there are no edges in E_V which join a vertex in L with a vertex in R. The separator X generates therefore a partition (L, X, R) for V. The *cost* of a DAG-vertex separator $c(X)$ is given by the number of nodes removed in X ($c(X) = |X|$).

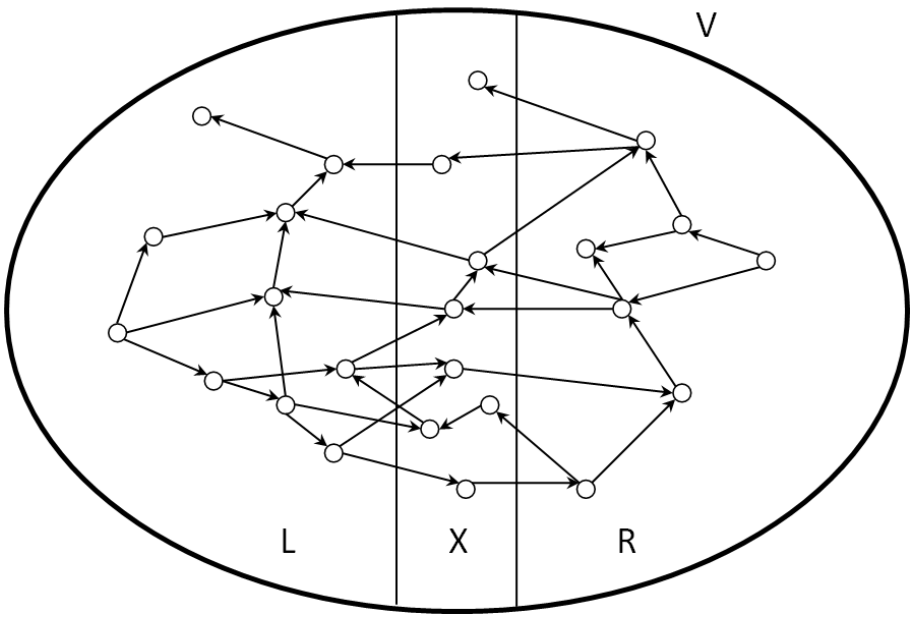

Figure 3.1: Example of DAG-vertex separator

In Figure 3.1 it is proposed an example of DAG-vertex separator for a DAG. The definitions of *balance* and *b-balanced* separator previously stated for undirected graphs can still be used for DAG-vertex separators. Observing a partition generated by a DAG-node separator, we can see that there will generally be vertices in L (R) which are head of edges whose tail is in X, and vice versa.

We will denote the set of all DAG-vertex separators for $\overrightarrow{G_V}$ as $\Xi(\overrightarrow{G_V})$.

Later in Chapter 4 we will introduce a particular sub-class of DAG-vertex separators, called *topological separators*, which exhibits a regularity in the direction of the edges connecting X to R and L.

As of now, and in the following paragraphs, we will consider all DAG-vertex separators according to the previous definition.

3.2 A divide-and-conquer approach

In computer science, divide and conquer (D&C) is an important algorithm design paradigm, based on multi-branched recursion. The main idea behind it is to recursively break down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original

problem [8].

This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

In particular, we will show how to decompose a DAG according to the divide and conquer approach in order to obtain problems of smaller size and thus manageable with possibly smaller memory space. In order to achieve this result we will exploit the concept of DAG-separator previously proposed, and the freedom concerning the schedule construction, obtainable through recalculations.

Definition 3.5 (Undirected Intrinsic Graph of a DAG)**.** Given an input n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ we can extract its undirected intrinsic graph $G_V(V, A_V)$ whose nodes set corresponds to the vertices set V of the DAG and whose arcs set A_V can be obtained simply ignoring the orientation of the edges in $\overrightarrow{E_V}$.

It is interesting to see how, according to this definition, different DAGs may have the same undirected intrinsic graph.

A graph G_V can be decomposed in smaller parts through the extraction of a b-balanced node separator X, obtaining the partition (L, X, R). Since the set of vertices for whom the undirected graph G has been defined corresponds exactly with the set of vertices for $\overrightarrow{G_V}$, we can easily see how any separator X in G_V corresponds to a DAG-node separator in $\overrightarrow{G_V}$ which will be constituted by the same vertices.

The idea behind the use of the separator is to try to execute the DAG holding just one of the two parts in memory at each time. This way, whenever we are able to find a separator which splits $\overrightarrow{G}$ in two balanced components, the required memory will decrease accordingly. Generally, the first time we consider each part, only a limited subset of their vertices will be evaluable, specifically just those whose predecessors have already been computed and are presently stored in memory. Once all the possible evaluations have been performed, in order to proceed in the computation we shall switch to the evaluation of the other part. However, in doing so, great attention needs to be used in order not to lose any of the useful information accumulated in the previous computational steps and still necessary for the remainder of the DAG evaluation.

This is where the properties of the separator X come into play. Since there are no edges connecting vertices in L to vertices in R, for each vertex $v \in L$ (resp.,

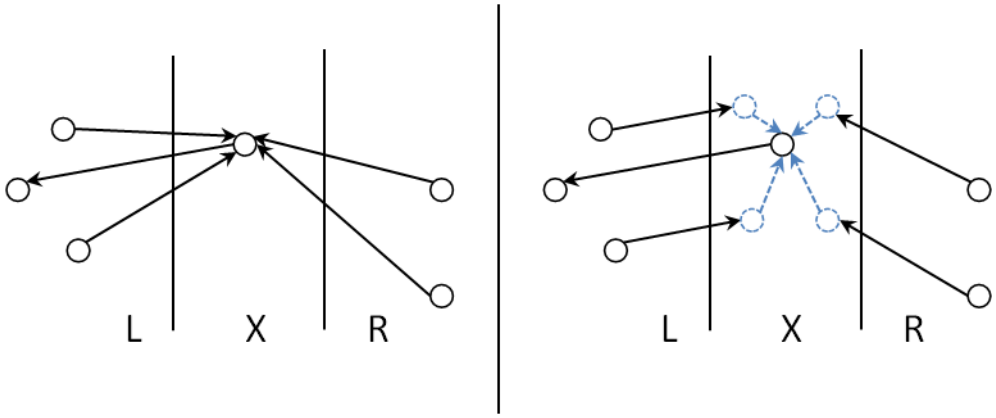

Figure 3.2: Example of possible blocking situation

$v \in R$) all its predecessors $u \in pa(v)$ will be $u \in L \cup X$ (resp., $v \in R \cup X$). Thus, while switching between the evaluation of $L \cup X$ and $X \cup R$ (and vice versa) the only vertices that should be kept in memories are those in separator X. We can safely discard all the other vertices (except for the input nodes), since whenever they will be needed again in the prosecution of the schedule, it will always be possible to recalculate them form the inputs and the vertices in X. This way, recalculations grant a high level of freedom in memory management.

Intuitively, proceeding in the alternate execution of $L \cup X$ and $R \cup X$ will allow the evaluation of an increasing number of vertices at each iteration, ultimately enabling the computation of all the output values of $\overrightarrow{G}$.

In the next section we will accurately demonstrate the correctness of this approach, showing how is possible to obtain a computational schedule which evaluates $\overrightarrow{G}$ in a finite number of steps using memory space $O(\max(|L \cup X|, |R \cup X|))$, without erroneous deadlocks.

3.3 Buffer space for separator vertices predecessors

The construction of a DAG-node separator, as presented in the previous section, may give rise to a situation which may lead to a block of the computation.

This situation occurs whenever in the separator X chosen for $\overrightarrow{G_V}$ there is a vertex v which has at least one predecessor in L and at least one predecessor in R. Since the vertices of L and R will never be present simultaneously in memory,

v predecessors will never be available at the same time, and thus v the operation associated with vertex v will never be executable (an example is proposed in Figure 3.2a).

To overcome this problem, additional memory space should be allocated to hold the value of the predecessors of the vertices in the separator. This way, whenever one of these values is available, it is stored in a *buffer memory* and is not affected by the switching between zones currently being executed.

In particular the additional space to be allocated is $O(Deg^-(\overrightarrow{G_V})\,|X|)$, where $Deg^-(\overrightarrow{G}) = \max_{v\in V} deg^-(v)$ is the maximum in degree of all the vertices in V.

This correction may be visualized on the DAG by replacing each directed edge $\langle u, v\rangle$,where $u \in L \cup R$ and $v \in V$, with a buffer vertex $u_v \in X$, and the directed edges $\langle u, u_v\rangle$ and $\langle u_v, v\rangle$ as shown in Figure 3.2b .

3.4 First separation level

We will then proceed to formalize the procedure previously described and to prove the correctness of proposed technique by showing computations which using bounded memory space allow to successfully evaluate the output vertices of the DAG. We also show an upper bound on the number of operations needed by such computations.

3.4.1 Main statement

We will now proceed to prove the following theorem:

Theorem 3.1 (DAG-separators Execution Theorem)**.**
Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ and DAG-node separator X which generates the partition (L, X, R) of V. The space complexity $S_{free}(\overrightarrow{G})$ satisfies the following constraint:

$$S_{free}(\overrightarrow{G}) \leq Deg^-(\overrightarrow{G})\,|X| + \max\left(|L|\,, |R|\right). \tag{3.1}$$

Proof. In order to prove the correctness of the statement, we show how given a DAG $\overrightarrow{G_V}$ and a DAG-vertex separator X it is possible to find a free-input computation C which computes $\overrightarrow{G_V}$ without erroneous blocking, using memory space at most $Deg^-(\overrightarrow{G})\,|X| + \max\left(|L|\,, |R|\right)$. In order to do so we will consider a topological ordering of the vertices of $\overrightarrow{G_V}$. Without loss of generality we will

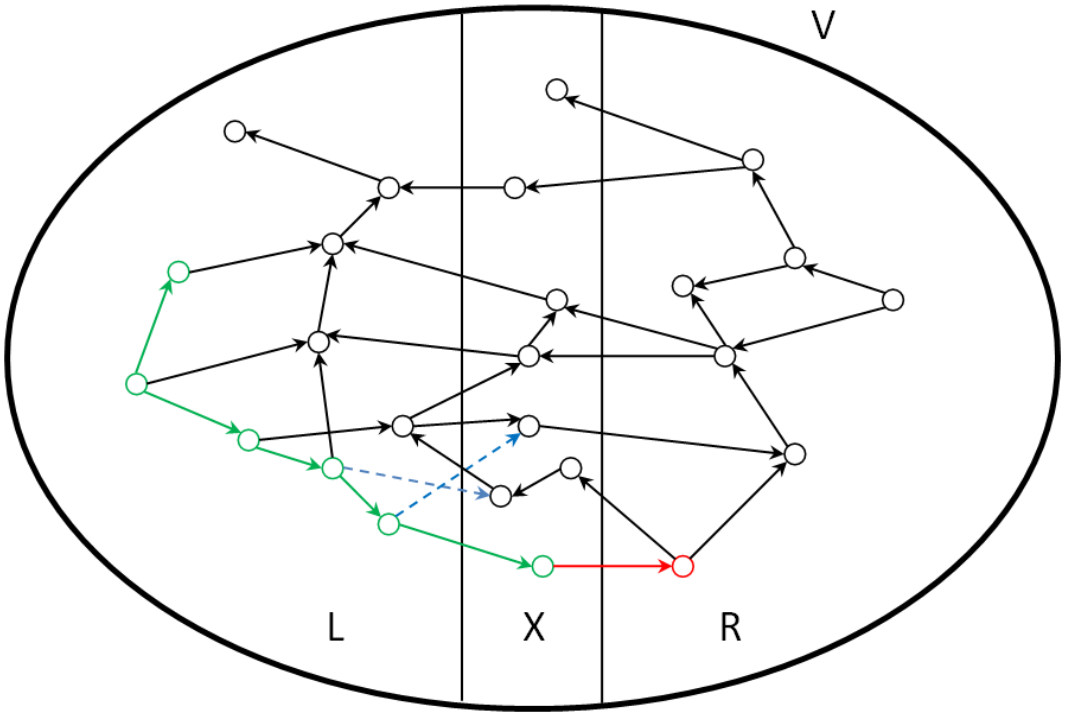

Figure 3.3: First execution phase

arbitrarily assume to the first vertex of this topological ordering is in $L \cup X$. This choice is made just for the sake of simplicity; an equivalent proof may be attained starting from $X \cup R$, replacing all occurrences of "R" with "L" in the remainder of the proof.

We then keep evaluating the vertices in topological order, this is particularly important since it ensures that all the predecessors of the vertex considered in the i-th step will have already been evaluated in the previous $i-1$ steps. It should be remarked that, since the available space is enough to hold the whole $L \cup X$, in this phase the vertices will be evaluated without recalculations.

However, after the evaluation of $1 \leq \iota \leq |L \cup X|$ vertices (highlighted in green in Figure 3.3), it will not be possible to proceed in the computation, since the next vertex w to be evaluated according to the topological ordering is in R (highlighted in red in Figure 3.3).

The limit situation $i = |L \cup X|$ may otherwise be verified in the further steps of the computation, and indicates that all the vertices in $L \cup X$ have been evaluated and it will not be necessary to *return* to $L \cup X$ in future calculations.

In the prosecution, we will assume that during the first execution of $L \cup X$

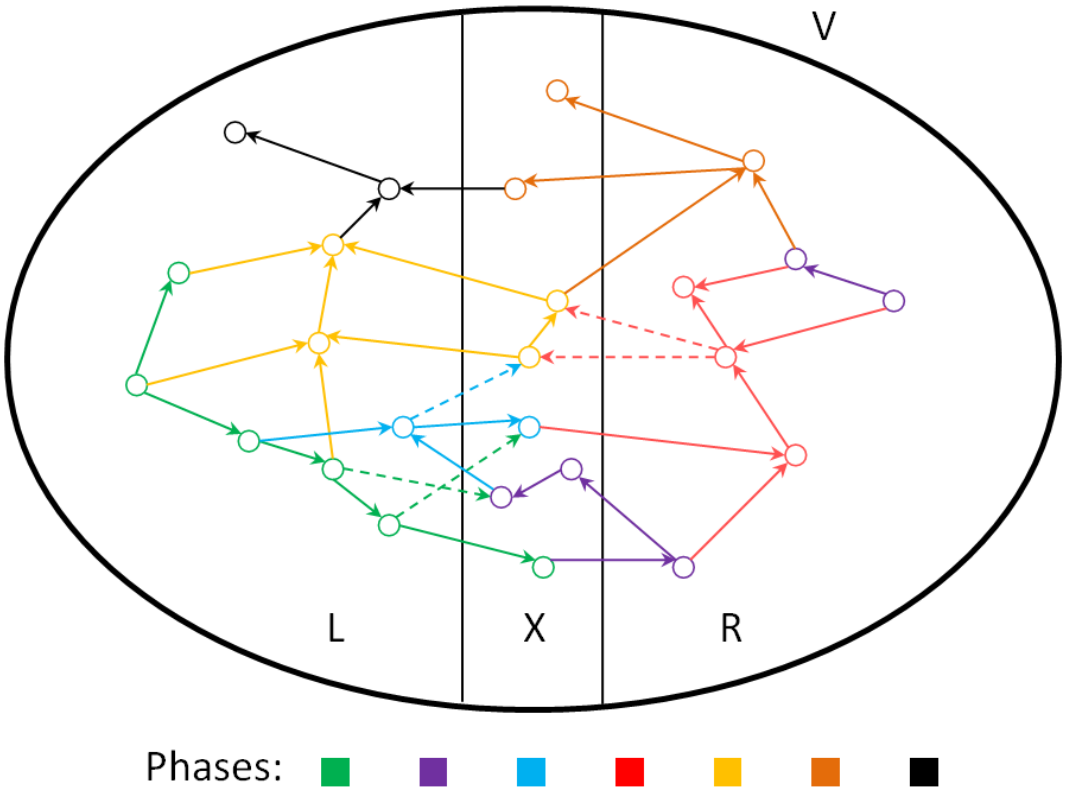

Figure 3.4: Phases of DAG evaluation

$1 \leq \iota < |L \cup X|$ vertices have been evaluated.

After the end of first phase, all vertices evaluated in L will be discarded, while the values in X shall remain in memory, and we will switch to the evaluation of the vertices in $X \cup R$.

The computation will then resume in topological order from the vertex w of $X \cup R$, using as "*input data*" not only the input vertices in R, but also the vertices in X evaluated in the previous step. It is important to understand that, since $L \cup X$ and $X \cup R$ communicate only through the vertices in X, all the information generated by the previous evaluation of one of the two parts which is relevant for the evaluation of the vertices of the other, is completely contained by the vertices of the separator.

In particular since the computation has proceeded evaluating vertices in a topological order is safe to assume that either $w \in I$, and therefore obviously immediately evaluable, or $pa(w) \subseteq X$. Since the values of vertices in X are kept in memory since their first evaluation and the calculation of the vertices in topological order guarantees that all vertices $pa(w)$ have already been evaluated,

we can conclude that all predecessors of w will be available in memory and thus w will be evaluable.

The computation will thus continue on the vertices in $X \cup R$ until the next vertex u to be evaluated in the topological ordering is in L (note that the vertex u could immediately follow w). The computation will then return to $L \cup X$, after discarding the vertices in L and keeping in memory all vertices in X evaluated that far.

In particular, $pa(u)$ can be partitioned in the two subsets $pa_L(u)$, the set of predecessors of u in L, and $pa_X(u)$, the set of predecessors of u in X.

- Since the computation has proceeded in topological order, all vertices in $pa_L(u)$ have already been evaluated in previous executions of $L \cup X$, and thus, using recalculation, they can be computed again;

- The fact that u is next to be evaluated assures that all its predecessors in X have already been calculated. Therefore, since the values of vertices in X are kept in memory since their first evaluation, all $pa_X(u)$ will be available.

We can thus conclude that u will actually be evaluable. The same reasoning can be applied to all subsequent vertices of the topological order, and this allows us to conclude that the entire DAG can be evaluated using at most $Deg^-(\overrightarrow{G})\,|X| + \max(|L|, |R|)$ memory space. The desired C will thus be composed by the finite succession of the vertex evaluations performed in each phase (see Figure 3.4 for an example).

□

The previous result allows us to obtain the following corollary:

Corollary 3.2.
Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, its space complexity $S_{free}\left(\overrightarrow{G_V}\right)$ will satisfy the bound:

$$S_{free}(\overrightarrow{G}) \leq \min_{X \in \Xi(\overrightarrow{G_V})} \left(Deg^-(\overrightarrow{G})\,|X| + \max(|L|, |R|) \right). \tag{3.2}$$

where X is a DAG-node separator for $\overrightarrow{G_V}$ which generates the partition (L, X, R) of V.

3.4.2 Accuracy of the bound in relation to separator cost and balance

The previous results allows us to find an upper bound on space complexity achievable by using recalculations. In particular, if for an n-vertex DAG $\overrightarrow{G}(V, \overrightarrow{E})$ a separator X is used, it is possible to express the memory space required in terms of its balance b:

$$S^X_{free}(\overrightarrow{G}) \leq Deg^-(\overrightarrow{G})\,|X| + (1-b) \tag{3.3}$$

While the term $|X|$ (the cost of the separator X) depends just on the size of the chosen separator, it is easy to see that the more balanced the division operated by X, the less space will be required by the term $(1-b)n$. In particular, for every separator X' which achieves the best balanced separation with $b(X'_{\overrightarrow{G}}) = \frac{1}{2}\left(1 - \frac{|X'|}{|V|}\right)$, the 3.3 will become:

$$S^{X'}_{free}(\overrightarrow{G}) \leq \frac{1}{2}n + |X'|\left(Deg^-(\overrightarrow{G}) - \frac{1}{2}\right).$$

Thus, in order to minimize the memory space used by a computation obtained via the separator X, the trade-off between the size of the separator X and the balance of the DAG division achievable by using it should be carefully analyzed.

A possible improvement of the tightness of the bound 3.1 is achievable through a more detailed analysis of the in-degrees of the vertices that constitute the separator X. In particular, the 3.1 can be rewrited as:

$$S^X_{free}(\overrightarrow{G}) \leq \sum_{v \in X} deg^-(v) + \max\left(|L|, |R|\right),$$

where the term $\sum_{v\in X} deg^-(v)$ represents the exact memory space which should be reserved in order to prevent problems that may arise from the circumstances discussed in Section 3.3.

Despite the fact that the last bound is more precise, the ones given in 3.1and 3.2 remain of interest because they relate the space complexity of $\overrightarrow{G}$ to the size of the separator.

3.4.3 Time complexity

In describing the way to build a computation γ which satisfies the bound 3.1, the possibility of re-evaluating some vertex was strongly exploited. Following

the phase execution scheme described in Theorem 3.1, is possible to obtain an upper bound for the number of vertex evaluations composing any computation γ generated with reference to the separator X, used for the decomposition of $\overrightarrow{G}$.

It has been seen that, following the topological order, it will be necessary to switch the part of the DAG being evaluated at each step. Thus, the maximum number of times each part may be considered is $\min(|L|, |R|)$, since when all vertices in one of two parts have been calculated it will no longer be necessary to return to that part. In the worst case, during each evaluation only one new vertex is computed, but it may be necessary to recalculate all the other vertices. Vertices in X are evaluated just one time.

The previous considerations lead to:

$$T_X(\overrightarrow{G}) \leq |X| + \sum_{i=1}^{\min(|L|,|R|)} i + \sum_{i=1}^{\min(|L|,|R|)} (\max(|L|, |R|) - i)$$

$$T_X(\overrightarrow{G}) \leq |X| + (\max(|L|, |R|) \min(|L|, |R|)). \tag{3.4}$$

In order to sharpen the previous upper bound, it is necessary to observe carefully the number of times each part is evaluated. This quantity is closely related to the topological ordering according to which the vertices are evaluated in $\overrightarrow{G_V}$.

Of particular interest is the topological ordering $\phi^\star = \phi_1\phi_2 \ldots \phi_n$ for which, whenever the part $L \cup X$ (resp. $X \cup R$), all vertices that are evaluable (that is those for which all predecessors are available or evaluable), are actually evaluated according to the topological ordering $\phi^\star$ before switching to $X \cup R$ (resp. $L \cup X$). It can be argued that $\phi^\star$ is indeed a topological ordering since, for each $1 \leq i \leq n$, $pa(\phi_i) \subseteq \{\phi_1, \ldots, \phi_{i-1}\}$ and $\{\phi_1, \ldots, \phi_i\}$ is a closed subset of V.

As shown in the proof of Theorem 3.1, following the proposed topological ordering $\phi^\star$ it will be possible to evaluate the entire DAG. Let w be a vertex in the topological ordering for which it will be necessary to switch from the evaluation of the current part of the DAG $\overrightarrow{G}$ (i.e. $L \cup X$) to the other (i.e. $X \cup R$). In the proof of Theorem 3.1, it has been shown that w is actually always evaluable.

However, since the vertices are being considered following the topological order $\phi^\star$, it is particularly important to notice that the vertex w must have at least one predecessor in X which was not available during the last execution of $L \cup X$. In fact, if this was not the case the vertex w could have been already

evaluated in a previous execution of $L \cup X$. Following the same reasoning will lead to conclude that this circumstance is also verified in all subsequent phases.

This observation is of particular significance, since it implies that during each computation of $L \cup X$ or $X \cup R$ (except for the last) at least one vertex of X will be evaluated for the first time.

Since once a vertex in X is calculated it is stored in memory, it is safe to conclude that after at most $|X|$ phases, all the vertices on X will have been evaluated and thus the two parts $L \cup X$ and $X \cup R$ will be calculated altogether no more than $|X|$ times.

This considerations allows to obtain the following upper bound, achievable by computations whose space requirement satisfy the bound in 3.1:

$$T_X(\overrightarrow{G}) \leq |X| + |X| \left(\max\left(|L|, |R|\right)\right) \tag{3.5}$$

which will be tighter than 3.4 whenever $|X| \leq \min\left(|L|, |R|\right)$.

3.4.4 Standard computations

The computation C, used in the proof of the main statement, is a free-input computation. It is however possible to achieve a similar result for a standard computation for which all input values reside in memory from the beginning of the computation until their last utilization. The 3.1 should thus be changed into:

$$S(\overrightarrow{G}) \leq |X| + \max\left(|L|, |R|\right) + |I| - |I \cap X| \tag{3.6}$$

where I is the set of input vertices of $\overrightarrow{G}$.

3.5 A recursive separator application

In this section we will show how to use the previous result in a divide and conquer recursive approach, aimed to achieve a stronger upper bound on space complexity for DAGs.

3.5.1 Recursive extension of the separator method

In the previous section we were able to obtain an upper bound for space complexity by using a vertex-separator for the DAG $\overrightarrow{G}$ extracted from the undirected

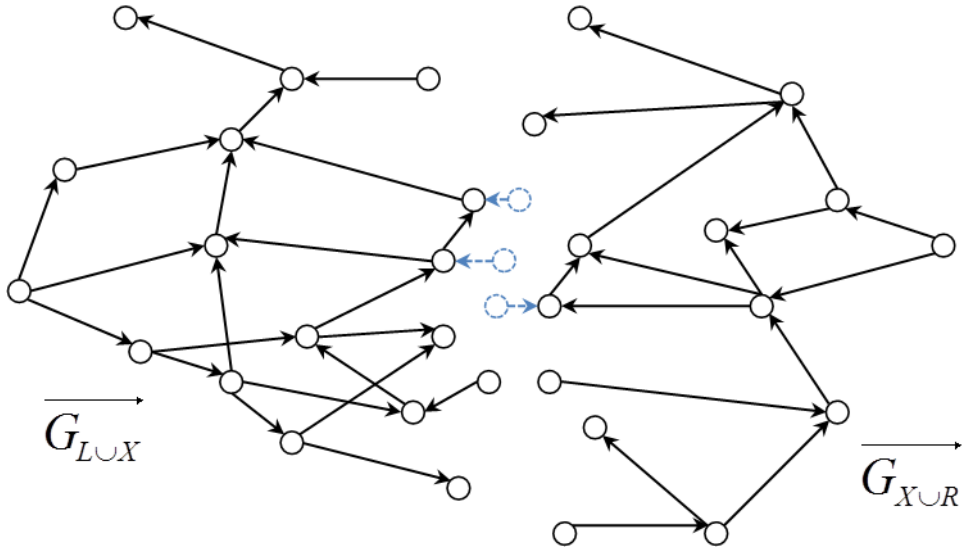

Figure 3.5: Extractions of sub-DAGs generated by a DAG-vertex separator

intrinsic graph G. However, concerning the computations of the two parts generated, $L \cup X$ and $X \cup R$, enough memory space had been set aside to hold the bigger one of them entirely in memory during each phase without further optimizations.

In order to obtain a reduction of the overall space requirement, we would like to apply recursively in the evaluation of $L \cup X$ and $X \cup R$ the same decomposition method (based on a vertex separator) used in the first level for the whole DAG $\overrightarrow{G}$. To do so, it will be necessary to carefully individuate the sub-DAGs generated by a DAG-vertex separator X.

Definition 3.6 (Sub-DAGs generated by a DAG-vertex separator X)**.** Given an n-vertex DAG $\overrightarrow{G_V}$, let X be a DAG-vertex separator which generates the partition (L, X, R). $\overrightarrow{G_{L\cup X}}(V', \overrightarrow{E'})$ (resp., $\overrightarrow{G_{X\cup R}}(V'', \overrightarrow{E''})$), it is a sub-DAG of $\overrightarrow{G_V}$ (shown in Figure 3.5) defined as follows:

- the vertices set V' (resp., V'') will be composed by all vertices in L (resp., R) and the vertices in X which are adjacent to at least one vertex in V'.
- the directed edges set $\overrightarrow{E'} = \{\langle u, v\rangle \,|\, \langle u, v\rangle \in E \wedge u, v \in V'\}$.

The inputs of this sub-DAG will be constituted of both the input vertices in $I \cap (L \cup X)$ and the vertices in $X \cap V'$ with at least one predecessor in R; among these there may be however vertices w with predecessors in V' which will thus not be considered *proper* input vertices for $\overrightarrow{G_{L\cup X}}$. For these vertices we can consider

as inputs the vertices $pa(w) \cap R$, which according to the matter presented in section 3.3 will be stored in memory as *buffer space*.

The outputs of this sub-DAG will be constituted of both the output vertices in $O \cap (L \cup X)$ and the vertices in $X \cap V'$ with no successors in V'.

Obviously $\overrightarrow{G_{X \cup R}}(V'', \overrightarrow{E''})$ will be defined exactly as $\overrightarrow{G_{L \cup X}}(V', \overrightarrow{E'})$, by replacing V' with V'', $\overrightarrow{E'}$ with $\overrightarrow{E''}$, L with R and R with L.

In Figure 3.5, we show the sub-DAGs extracted from the DAG $\overrightarrow{G_V}$ presented in Figure 3.1, by means of the DAG-vertex separator X.

Following this approach it will be possible to rewrite the 3.1 as a recursion.

Theorem 3.3 (Separator Space Complexity)**.**
Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, its space complexity $S_{free}\left(\overrightarrow{G_V}\right)$ satisfies the following bound:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^{-}(\overrightarrow{G_V})\,|X| + \max\left(S_{free}(\overrightarrow{G_{L \cup X}}), S_{free}(\overrightarrow{G_{X \cup R}})\right) \tag{3.7}$$

where X is a DAG-node separator for $\overrightarrow{G_V}$, which generates the partition (L, X, R) of V.

Proof. In order to prove the correctness of the statement, we show how given a DAG $\overrightarrow{G_V}$ and a DAG-node separator X, it is possible to find a free-input computation C which evaluates $\overrightarrow{G_V}$, without erroneous locks, using memory space at most $Deg^{-}(\overrightarrow{G_V})\,|X| + \max\left(S_{free}(\overrightarrow{G_{L \cup X}}), S_{free}(\overrightarrow{G_{X \cup R}})\right)$.

In the same way as in the proof of Theorem 3.1, we will consider a topological ordering of the vertices in $\overrightarrow{G_V}$, for the sake of simplicity we will assume that the first vertex of the ordering will be in $L \cup X$.

The computation will proceed in topological order, and for the construction of the sub-DAG previously discussed each evaluation of a vertex in $L \cup X$ (resp., $X \cup R$) in the proof of Theorem 3.1 corresponds here to a computation of a vertex in the sub-DAG $\overrightarrow{G_{L \cup X}}$ (resp., $\overrightarrow{G_{X \cup R}}$). Therefore, for the same reasons discussed in the demonstration of Theorem 3.1, here too it will be possible to evaluate the vertices in the topological order by exploiting the structure of the separator without the risk of erroneous blockings in the computation.

By definition of space complexity, $S_{free}(\overrightarrow{G_{L \cup X}})$ (resp., $S_{free}(\overrightarrow{G_{X \cup R}})$) is the minimum memory size sufficient to compute all nodes in the DAG $\overrightarrow{G_{L \cup X}}$ (resp., $\overrightarrow{G_{X \cup R}}$). Therefore, we can reduce the space reserved for the evaluation of two

parts from $|L|$to $S_{free}(\overrightarrow{G_{L\cup X}})$ (resp., from $|R|$to $S_{free}(\overrightarrow{G_{X\cup R}})$), without compromising the ability to calculate the whole DAG. Since only one of the two sub-DAGs will be evaluated in a certain instant of the computation, it will be sufficient to reserve memory space equal to the maximum space complexity between the two, in addition to the space already reserved for the separator X.

Therefore, we can conclude that it is possible to find a free-input computation C composed by the sequence of all vertex evaluations, which calculates $\overrightarrow{G_V}$ using at most $S_{free}(\overrightarrow{G_V}) \leq Deg^-(\overrightarrow{G_V})\,|X| + \max\left(S_{free}(\overrightarrow{G_{L\cup X}}), S_{free}(\overrightarrow{G_{X\cup R}})\right)$ memory space. □

Observation 3.5.1.
If the analysis is restricted only to standard computations, the previous bound must be rewritten as:

$$S(\overrightarrow{G_V}) \leq Deg^-(\overrightarrow{G_V})\,|X| + \max\left(S_{free}(\overrightarrow{G_{L\cup X}}), S_{free}(\overrightarrow{G_{X\cup R}})\right) + |I| - |I \cap X|\,. \quad (3.8)$$

Observation 3.5.2.
The estimated number of operations that must be performed by a schedule C, generated with reference to the separator X used for the decomposition of $\overrightarrow{G}$, can be expressed in a recursive form based on 3.5:

$$T_X(\overrightarrow{G}) \leq |X|\left(\max\left(T(\overrightarrow{G_{L\cup X}}), T(\overrightarrow{G_{X\cup R}})\right)\right). \quad (3.9)$$

From the result of Theorem 3.3 follows immediately:

Corollary 3.4.
Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$, its space complexity $S_{free}\left(\overrightarrow{G_V}\right)$ will satisfy the bound:

$$S_{free}(\overrightarrow{G}) \leq \min_{X\in\Xi(\overrightarrow{G_V})}\left(Deg^-(\overrightarrow{G})\,|X| + \left(S_{free}(\overrightarrow{G_{L\cup X}}), S_{free}(\overrightarrow{G_{X\cup R}})\right)\right). \quad (3.10)$$

where X is a DAG-node separator for $\overrightarrow{G_V}$ which generates the partition (L, X, R) of V.

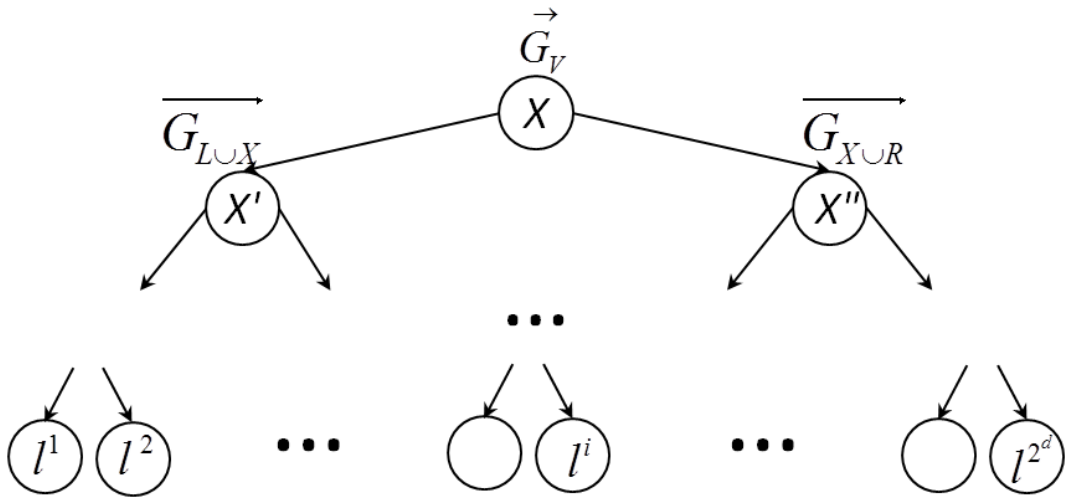

Figure 3.6: Separator hierarchy based DAG tree decomposition

3.5.2 A separator hierarchy-based DAG decomposition

In order to estimate the values $S_{free}(\overrightarrow{G_{L\cup X}})$ and $S_{free}(\overrightarrow{G_{X\cup R}})$, it will be possible to resort again to the separator approach, obtaining:

$$S_{free}(\overrightarrow{G_{L\cup X}}) \leq Deg^{-}(\overrightarrow{G_{L\cup X}})\,|X'| + \max\left(S_{free}(\overrightarrow{G_{L'\cup X}}), S_{free}(\overrightarrow{G_{X'\cup R'}})\right),$$

where X' is a node separator for $\overrightarrow{G_{L\cup X}}$ which generates the partition (L', X', R') of $L \cup X$.

For example, in the case in which $S_{free}(\overrightarrow{G_{L\cup X}}) \geq S_{free}(\overrightarrow{G_{X\cup R}})$, it is possible to expand the first recursion level for 3.7 and obtain:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^{-}(\overrightarrow{G_V})\,|X| + Deg^{-}(\overrightarrow{G_{L\cup X}})\,|X'| + \max\left(S_{free}(\overrightarrow{G_{L'\cup X}}), S_{free}(\overrightarrow{G_{X'\cup R'}})\right).$$

It is interesting to observe how both the separators defined for $\overrightarrow{G_V}$ and $\overrightarrow{G_{L\cup X}}$ appear in the recursion followed by the recursive term. However, since just one between $\overrightarrow{G_{L\cup X}}$ and $\overrightarrow{G_{X\cup R}}$ will be executed at each time, just the separator defined for the sub-DAG being executed should be kept in memory.

Proceeding in the application of this approach, the DAG $\overrightarrow{G_V}$ can be further decomposed in smaller and smaller sub-DAGs through a recursive extraction of a separator from the components obtained at the previous step, according to 3.7. The set of separators thus obtained constitutes a *separator hierarchy*.

We can use a separator hierarchy of this kind to develop a *tree decomposition* of $\overrightarrow{G_V}$ (shown in Figure 3.6). The root node will be associated to the vertices of a separator X, generating the partition (L, X, R) for V. The children of the root will then be used as the root of the recursively built decomposition of the sub-DAGs $\overrightarrow{G_{L\cup X}}$ and $\overrightarrow{G_{X\cup R}}$.

The decomposition will cease when the size of the components extracted by a separator will be at most m_L, and this components will constitute the *leaf nodes* of the tree decomposition, and we will refer to them as *leaf components*. Obviously, for any leaf components l^i, $S_{free}(l^i) \leq |l^i| = m_L$.

It should be noted that the arbitrary choice of the maximum size of the leaf components m_L will constitute a lower bound of the space usage achievable through the recursive separation, and will determine the number of decomposition levels (number of recursive levels) d of $\overrightarrow{G_V}$. In particular, if $b \in \left[\frac{1}{2}, 1\right)$ is the maximum acceptable value of balance for a separator chosen for any of the possible sub-DAG obtained in the decomposition, the number of levels d will be the minimum integer value for which $m_L \geq b^d n$, and thus:

$$d = \log_b \left(\frac{n}{m_L}\right).$$

In the tree decomposition thus obtained, each *path* P_{l^i} from a leaf component l^i to the root is constituted by the vertices in the union of all the separators extracted in each recursive level. We shall refer to the width of a path in the tree decomposition generated by a separator hierarchy as its *cardinality* $|P_{l^i}|$. A peculiar feature of the tree-decomposition thus obtained is that each vertex associated with a leaf node l^i will be adjacent only to vertices in the same leaf component and to vertices in the path $|P_{l^i}|$ from the root to the leaf component itself. Thus, we can conclude that vertices in a certain leaf component *communicate* with the rest of the DAG vertices only through P_{l^i}. In particular, vertices in a certain leaf component l^i communicate with vertices in another leaf component l^j only through the vertices in $P_{l^i} \cap P_{l^j}$.

Therefore it will possible to calculate the whole $\overrightarrow{G_V}$ by evaluating just one leaf component in each phase, while keeping in memory just the vertices in the corresponding path to the root of the tree decomposition (highlighted in red in Figure 3.7).

These considerations allow us to obtain the following result.

Proposition 3.5 (Upper bound based on separator hierarchy)**.**

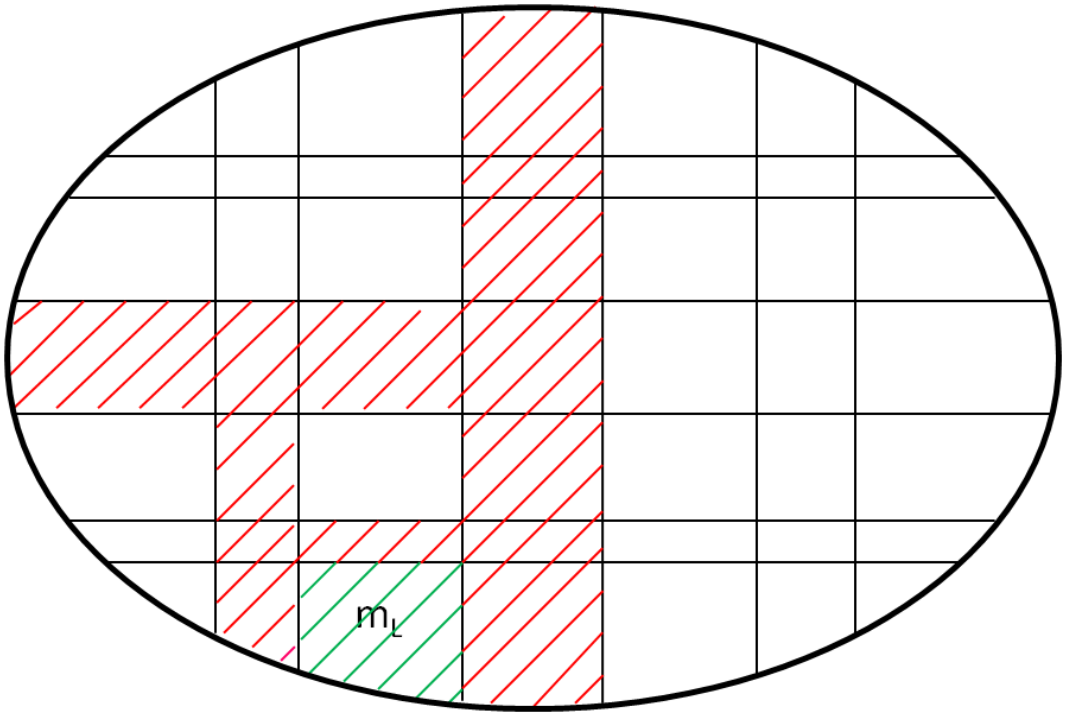

Figure 3.7: Leaf Component - Root path of DAG tree decomposition

Given an n-vertex DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$*, its space complexity* $S_{free}(\overrightarrow{G_V})$*satisfies the bound:*

$$S_{free}(\overrightarrow{G_V}) \leq Deg^-(\overrightarrow{G_V}) \max_{1 \leq i \leq 2^d} |P_{l^i}| + m_L, \tag{3.11}$$

where P_{l^i} *are the paths connecting each of the* 2^d *leaf component of size at most* m_L *of a tree separator decomposition of* $\overrightarrow{G_V}$ *with d levels.*

Proof. In the previous part of this section, we discussed how the tree structure is generated and why it is sufficient to keep in memory at any step of the computation just the vertices of the *widest path* from leaf to root. Since 3.11 is built by further developing the recursion in 3.7, it will still be possible to actually find a free-input computation C which evaluates $\overrightarrow{G_V}$ using at most the memory space given by 3.11. □

In particular, to highlight the separators identified at each level, the 3.11 can be rewritten as:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^-(\overrightarrow{G_V}) \max_{1 \leq i \leq 2^d} \left(\sum_{i=1}^{d} \left| X^i \right| \right) + m_L.$$

The bound proposed in the main statement is referred to free-input computations. It is however possible to achieve a similar result for standard computations:

$$S(\overrightarrow{G}) \leq Deg^{-}(\overrightarrow{G_V}) \max_{1 \leq i \leq 2^d} |P_{l^i}| + m_L + |I| \,.$$

where I is the set of input vertices of $\overrightarrow{G_V}$.

3.5.3 Time complexity analysis for computations based on DAG separator-based decompositions

Following the same approach used in Section 3.4.3, it is possible to obtain an estimation on the actual number of vertex evaluations executed by a computation for which the memory space used satisfies the bound given by 3.11. The basic bound given in 3.4 will be rewritten as the following recursion:

$$T(\overrightarrow{G_V}) \leq |X| \left(\max \left(T(\overrightarrow{G_{L \cup X}}), T(\overrightarrow{G_{X \cup R}}) \right) \right). \tag{3.12}$$

Supposing $T(\overrightarrow{G_{L \cup X}}) \geq T(\overrightarrow{G_{X \cup R}})$, the second level of the recursion will be:

$$T(\overrightarrow{G_V}) \leq |X|\,|X'| \left(\max \left(T(\overrightarrow{G_{L' \cup X'}}), T(\overrightarrow{G_{X' \cup R'}}) \right) \right),$$

where X' is a DAG-vertex separator for $\overrightarrow{G_{L \cup X}}$ which generates the partition (L', X', R') of $L \cup X$.

Following the subsequent recursive steps, according to the decomposition realized using a separator hierarchy (as explained in detail in the previous section), the previous recursion can be further developed:

$$T(\overrightarrow{G_V}) \leq 2 m_L \prod_{i=1}^{d} \left| X^i \right|, \tag{3.13}$$

where X^i is the separator defined at each of the d recursive level of the tree path $P = \max_{1 \leq i \leq 2^d} |P_{l^i}|$.

3.5.4 Observations on the previous results

In this chapter we have shown how to find an upper bound for the space complexity of DAGs, based on a divide and conquer approach.

This result allows us to find a relation between the space complexity of DAGs and the separator hierarchies which may be used to decompose the underlying

undirected intrinsic graph. The fewer vertex will be necessary to extract the separator at each level, the fewer memory locations will be necessary for the complete evaluation of the DAG in analysis using free-input computations.

In particular, Proposition 3.5 allows us to conclude that the space complexity of a generic DAG $\overrightarrow{G}$ is related to the separator decomposition by the relation $O\left(Deg^{-}(\overrightarrow{G_V})P\right)$, where the term P represents the total *weight* of the separator structure.

All the bounds obtained are thus directly related to the topological structure of the DAG in analysis, and not to its peculiar computations. This allows to use the separator decomposition method as a general framework which can be effectively employed to study the space complexity of any given DAG trough a standard approach constituted by the separator individuation.

It is very interesting to notice how the property of the DAG used to obtain the bound is actually obtained by its undirected intrinsic graph, while the orientation of the edges will have an impact only on the number of times that the parts of the graph will actually be computed and thus on the number of vertex evaluations performed.

This upper bound can be successfully used to achieve a worst-case estimate of the benefits obtainable in terms of reduction of the required memory space by using the recalculation, with respect to computations in which the vertices are assessed a single time.

In particular let, $S_X(\overrightarrow{G_V})$ be an upper bound obtained from one of the previous results (3.1,3.7,3.11) and $S^{wr}(\overrightarrow{G_V})$ the minimum space necessary to evaluate $\overrightarrow{G_V}$ using computations which do not resort to vertex re-evaluations (calculated as closed dichotomy or using the marking rule as discussed in Chapter 2).

The order of magnitude of the minimum obtainable reduction in terms of necessary memory space for computations using recalculation, with respect to those strictly without recalculations, indicated as $\zeta(\overrightarrow{G_V})$, is:

$$\zeta(\overrightarrow{G_V}) = \Omega\left(\frac{S_X(\overrightarrow{G_V})}{S^{wr}(\overrightarrow{G_V})}\right). \tag{3.14}$$

Therefore, the fact that $S_X(\overrightarrow{G_V}) < S^{wr}(\overrightarrow{G_V})$ is a *sufficient condition* to conclude that using recalculations for the evaluation of $\overrightarrow{G_V}$, a benefit at least of order 3.14 can be achieved in terms of necessary memory space. It should however be remarked that the condition is not necessary as well.

In the next chapter we will show how to use this results to obtain significant estimations of the space complexity of some important classes of DAGs.

Chapter 4

Applications of the separator based approach

In this chapter we go on to discuss some applications of the results presented in Chapter 3. In the first part we will revisit the concept of topological separator of a DAG and some known results concerning its relation with space requirements for computations stricly without recalculations. We will then show how to use the method based on the separator to obtain a quantitative estimate of the potential significant benefits to be gained by using the recalculation for some classes of DAGs.

4.1 Topological separators

A vertex v of a DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ can be executed only if the set $pa(v)$ of its immediate predecessor has been executed. Generalizing this observation, a set $U \subseteq V$ can be executed only after the execution of its preboundary $\Gamma_{in}(U) = \cup_{v \in U} pa(v) \backslash U$.

The following definition captures the conditions under which the execution of U can be decomposed into the successive executions of subsets $U_1, U_2, \ldots, U_q$.

Definition 4.1 (Topological partition)**.** An ordered partition $(U_1, U_2, \ldots, U_q)$ of $U \subseteq V$ is said to be a topological partition of U if, for $r = 1, 2, \ldots, q$

$$\Gamma_{in}(U_r) \subseteq \Gamma_{in}(U) \cup \left(\cup_{i=1}^{r-1} U_i \right).$$

It can be seen that a topological partition of U can be refined into a topological

sorting of U, and thus into a computation without recalculations, and that sets U_i's are convex in the following sense.

Definition 4.2 (Convex subset)**.** A subset $U \subseteq V$ is said to be convex if, whenever u and v are in U, so is any path from u to v.

It is thus possible to formulate a simple strategy to execute the vertices in U by exploiting its topological partitioning.

Proposition 4.1 (thm)**.** *Let $S(\overrightarrow{G_U})$ be the space required by the execution of the DAG whose vertex set will be a convex set $U \subseteq V$ with $\Gamma_{in}(U)$ initially stored in memory. If $U_1, U_2, \ldots, U_q$ is a topological partition of U, then:*

$$S(\overrightarrow{G_U}) \leq \max_{i=1}^{q} S(\overrightarrow{G_{U_i}}) + \varphi(U), \tag{4.1}$$

where $\varphi(U) = \sum_{i=1}^{q} |\Gamma_{in}(U_i)|$.

Proof. We assume $\Gamma_{in}(U)$ is initially in memory; note that if $U = V$ then $\Gamma_{in}(V) = \emptyset$. To execute U for $i = 1, 2, \ldots, q$ do:

- Evaluate $\overrightarrow{G_{U_i}}$; this can surely be possible, since $|\Gamma_{in}(U_i)|$ will be in memory, and will require at most $\max_{i=1}^{q} S(\overrightarrow{G_{U_i}})$ memory space
- Keep in memory the values correspondig to vertices $U_i \cap \Gamma_{in}\left(\cup_{j=1+1}^{q} U_j\right)$, while deleting the other vertex of U_1. The values kept in memory this way will be at most $\varphi(U)$.

The bound on $S(\overrightarrow{G_U})$ thus descends from those of the individual steps of the above procedure [4]. □

In particular, we can see how the set of the vertices $\Gamma_{in}\left(\cup_{j=1+1}^{q} U_j\right) = X$ is actually a vertex separator for $\overrightarrow{G_U}$ which generates the partition $\cup_{j=1}^{i} U_j \backslash X,\ X,\ \cup_{j=1+1}^{q} U_j$ of the vertices in U. A peculiar characteristic of one such partition is that all the edges from X and $\cup_{j=1+1}^{q} U_j$ are directed from X to $\cup_{j=1+1}^{q} U_j$. We shall refer to these separators as *topological separators*.

It is important to remark that the calculations considered here through topological partitions are strictly without recalculations. In the proof of Proposition 4.1, the computation proceeds form a convex subset to the other without the possibility of returning to vertices already evaluated.

This is particularly interesting because it indicates that between $S^{wr}(\overrightarrow{G_V})$ and topological separators (which are defined with attention to the orientation of

the edges of the DAG), there exists the same relation that has been previously analyzed between the DAG-vertex separators (which on the contrary are defined on the intrinsic undirected graph obtained by ignoring the orientations of the edges) and the space complexity $S(\overrightarrow{G_V})$.

The aim is to identify topological partitions of a convex set U whose components, although not necessarily *geometrically similar* to U, share its decomposability properties. This feature is captured by the notion of *topological separator*.

Definition 4.3 ($(g(x), \delta)$-topological separator)**.** Let $0 < \delta < 1$, let $q > 1$ be an integer, and let $g(x)$ be a real function. A convex set $U \subseteq V$ has a $(g(x), \delta)$-topological separator if either $|U| = 1$ or:

- $|\Gamma_{in}(U_i)| \leq g(|U|)$;
- U has a topological partition $(U_1, U_2, \ldots, U_q)$where for each $i = 1, 2, \ldots, q$, $|U_i| \leq \delta |U|$;
- for each $i = 1, 2, \ldots, q$, U_i has a $(g(x), \delta)$-topological separator.

We can now consider the worst case space to execute a set U of size $|U| = k$ when U has a topological separator; the bound in Proposition 4.1 can thus be rewritten as:

$$\sigma(|U|) \leq \sigma(\delta |U|) + qg(\delta |U|).$$

where bounding φ we assumed that $g(x)$ is monotone non decreasing.

The idea behind the use of general vertex separator, presented in Chapter 3, instead of topological ones, aims to introduce a greater degree of freedom by partitioning the DAG in zones which may not constitute a topological partition, and which may be evaluated multiple times thanks to the possibility of executing recalculations.

4.2 Applications to planar DAGs

We now attempt to use Theorem 3.3 in order to obtain a significant upper bound on the space complexity of planar DAGs by exploiting some known results concerning the property of separators for this class.

4.2.1 The planar separator theorem

Definition 4.4 (Planar DAG)**.** A DAG $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ or an undirected graph $G_V(V, A_V)$ is said to be *planar* if it can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

Since for a DAG $\overrightarrow{G_V}$ its associated undirected intrinsic graph G_V is obtained just by ignoring the direction on the edges, it safe to assume that $\overrightarrow{G_V}$ is planar if and only if G_V is planar.

In graph theory, an important property related to the existence of separators for this class of graphs is given by the Planar Separator Theorem, proposed by Lipton and Tarjan [15], which is a form of isoperimetrical inequality for planar graphs, that states that any planar graph can be split into smaller pieces by removing a small number of vertices.

Theorem 4.2 (Planar Separator Theorem)**.**
Let G_V be any n-vertex planar graph. The vertices in G_V can be partitioned into three sets L, X, R such that no edge joins a vertex in L with a vertex in R, neither L nor R contains more than $2n/3$ vertices, and X contains no more than $\sqrt{8}\sqrt{n}$ vertices.

For the proof see [15].

The theorem does not require L and R to be connected. In particular, it is important to notice that the sub-Graphs $G_{L\cup X}$ (resp., $G_{L\cup X}$) of $G_V(V, E_V)$, whose vertices set will be constituted by $L \cup X$ (resp., $X \cup R$) and whose undirected edges set will be constituted by $\{(u, v)| \left(\langle u, v\rangle \in \overrightarrow{E_V} \vee \langle v, u\rangle \in \overrightarrow{E_V}\right) \wedge u, v \in L \cup X\}$(resp., $\{\ldots u, v \in X \cup R\}$), will still be planar and therefore they will maintain the same separator properties of G_V.

It is interesting to note how the constant $\frac{2}{3}$ in the statement of the separator theorem is arbitrary, and it is still possible to find a b-balanced separator of size $O(\sqrt{n})$ for any value $b \in \left(\frac{1}{2}, 1\right)$. In fact, a partition into more equal subsets may be obtained from a less-even partition by repeatedly splitting the larger sets in the uneven partition and regrouping the resulting connected components [9]. This may however cause the term $\sqrt{8}$ to change.

Lipton and Tarjan provide also an algorithm which, given a graph G, determines a partition (L, X, R) of the nodes in V which satisfies the requirements expressed in Theorem 4.2 within linear time $O(n)$.

The planar separation theorem discussed this far has been defined with regards to an undirected graph such as the undirected intrinsic graph $G(V, E)$ extracted from $\overrightarrow{G}(V, \overrightarrow{E})$. However, since the set of vertices and edges, with the exception of the directions, are the same in G and $\overrightarrow{G}$, we can safely state that every partition (L, X, R) defined in G by means of a vertex separator X corresponds to a partition (L, X, R) in $\overrightarrow{G}$ generated by a DAG-vertex separator X, constituted by the same vertices, which achieves the same balance. In the same way, the sub-graphs $G_{L\cup X}$ and $G_{L\cup X}$ defined above correspond to the sub-DAGs $\overrightarrow{G_{L\cup X}}$ and $\overrightarrow{G_{L\cup X}}$, constructed as described in Chapter 3. In particular, each sub-DAG will have the same number of vertices of the corresponding sub-graphs with at most $Deg^{-}(\overrightarrow{G})\,|X|$ additional vertices, and will maintain the planarity property.

We will now proceed to exploit this result in order to demonstrate how any n-vertex planar dag $\overrightarrow{G_V}(V, \overrightarrow{E_V})$ can surely be calculated using a memory space whose size is at most $O(\sqrt{n})$, using the tools obtained in Chapter 3.

4.2.2 An upper bound for planar DAGs space complexity

Theorem 4.3 (Upper bound for planar DAG space complexity)**.**
Let $\overrightarrow{G_V}$ be any n-vertex planar graph. $S_{free}(\overrightarrow{G_V}) \in O(\sqrt{n})$.

Proof. The planar separator theorem allows us to assume that it is possible to find a b-balanced DAG-vertex separator X whose size will be $c\sqrt{n} \in O\left(\sqrt{n}\right)$ and that, following from the previous considerations, the sub-DAGs generated by the separator will have a size of at most bn vertices, where c is a constant linked to the separator. For the sake of simplicity and without loss of generality [9], we will assume that $b = 1/2$.
The space complexity of n-vertex planar DAG $\overrightarrow{G_V}$ can be estimated according to Theorem 3.3 as:

$$S_{free}(\overrightarrow{G_V}) = c\sqrt{n} + \max\left(S_{free}(\overrightarrow{G_{L\cup X}}), S_{free}(\overrightarrow{G_{X\cup R}})\right).$$

The previous result can be further developed by applying the method based on the separator extraction to the sub-DAGs (which have been defined in Section 3.3), whose size will be at most $n/2$, in order to estimate the space complexity. Subsequent applications of this method will lead to obtain a separator hierarchy

which will induce a tree decomposition of $\overrightarrow{G_V}$, as discussed in Proposition 3.5:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^{-}(\overrightarrow{G_V}) \max_{1 \leq i \leq 2^d} |P_{l^i}| + m_L$$

where each path P_{l^i} form a leaf component l^i to the root is constituted by the vertices of all the separators extracted in each recursive level.

The application of the planar separator theorem at each level of the proposed tree-decomposition leads to the fact that the sizes of the sub-DAGs go down by a constant factor at each level and it will therefore be possible to observe a corresponding lowering of the upper bound on the number of nodes required to identify a DAG-vertex separator. In particular, at the i-th depth level of the tree decomposition there will be at most 2^i sub-DAGs composed by at most $O\left(\frac{n}{2^i}\right)$ vertices and for whom a $\frac{1}{2}$-balanced separator of size $O\left(2^{-\frac{i}{2}}\sqrt{n}\right)$ may be found.

The decomposition will cease when the size of the sub-DAGs extracted by a separator will be at most $m_L \in O(\sqrt{n})$, and thus each of the leaf components will surely be evaluable using $O(\sqrt{n})$ memory space. It should be noted that the choice of the maximum size of the leaf components m_L is arbitrary. The demonstration of the existence of a valid computation for any planar DAG which requires at most $O\left(\sqrt{n}\right)$ memory space will still be achievable for any $m_L \in O\left(\sqrt{n}\right)$. A particular choice for m_L will determine the number of decomposition levels of $\overrightarrow{G_V}$ and the upper bound of the time complexity for a computation achieving the bound $O\left(\sqrt{n}\right)$ on space complexity.

The depth d of the tree decomposition can be estimated as the smallest integer value for which $2^d\sqrt{n} \geq n$, and thus $d = \lceil \log_2\left(\sqrt{n}\right) \rceil$. According to these observations, the size of each path P_{l^i} can be estimated as:

$$|P_{l^i}| \leq \sum_{i=0}^{log_2\sqrt{n}} c\sqrt{\frac{n}{2^i}} \leq c\sqrt{n}\sum_{i=0}^{\infty} 2^{-i/2} \leq c2\left(1+\sqrt{2}\right)\sqrt{n} \in O\left(\sqrt{n}\right) \tag{4.2}$$

Therefore, Proposition 3.5 leads to the following bound for the space complexity of $\overrightarrow{G_V}$:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^{-}(\overrightarrow{G_V})O\left(\sqrt{n}\right) + O\left(\sqrt{n}\right) \in O\left(\sqrt{n}\right).$$

□

4.2.3 Observations and refinements

The hypothesis according to which the separators are $\frac{1}{2}$-balanced can be generalized to the case in which separators extracted at each level are at least b-balanced.

This will influence the depth d of the decomposition, as d will be the smallest integer for which $m_L \geq b^d n$, and thus:

$$d = \left\lceil \log_{\frac{1}{b}} \frac{n}{m_L} \right\rceil = \left\lceil \frac{\log_2 \frac{n}{m_L}}{\log_2 \frac{1}{b}} \right\rceil = \left\lceil \frac{\log_2 n - \log_2 m_L}{\log_2 \frac{1}{b}} \right\rceil$$

Therefore in the case for which $m_L \in O(\sqrt{n})$, it will be $d = \left\lceil \frac{\log_2 \sqrt{n}}{\log_2 \frac{1}{b}} \right\rceil \in O(\log_2 \sqrt{n})$.

The considerations used in the proof of Theorem 4.3, together with the results presented in Chapter 3, can be used to obtain an estimate of the time complexity $T(n)$ associated with a computation that allows to evaluate a n-vertex planar DAG using at most $O(\sqrt{n})$ memory space. These considerations allows us to write:

$$T(n) \leq c\sqrt{n}T(\frac{n}{2}) = c^d m_L \sqrt{n}^d \prod_{i=0}^{d} 2^{-\frac{i}{2}} = c^d m_L \sqrt{n}^d 2^{-\frac{d(d+1)}{4}},$$

where c is a constant associated with the size of the separators defined at each of the d levels. In particular, if $d \in O(\log_2 \sqrt{n})$, it will be $T(n) \leq m_L \sqrt{n}^{2\log_2 \sqrt{n}}$.

The usefulness of the result given by Theorem 4.2 consists in the fact that, given a DAG $\overrightarrow{G_V}$, it is possible to identify a quantitative bound to the space complexity only related to the property of planarity, without the need to evaluate the possible decompositions obtainable with separators. Therefore, if it is known the memory space $S^{wr}(\overrightarrow{G_V})$ needed for the evaluation of a DAG $\overrightarrow{G_V}$ using computations using strictly no recalculations it will be possible to obtain immediately a first estimate of the possible benefits achievable with computations with multiple assessments of some vertices. In particular, if $S^{wr}(\overrightarrow{G_V}) < \sqrt{n}$, there will be a benefit of a reduction of the necessary memory space at least of order $\frac{S^{wr}(\overrightarrow{G_V})}{\sqrt{n}}$. It should however be emphasized that this fact does not allow to conclude in itself that a DAG does not benefit from the recalculation, since a more detailed analysis of the hierarchies of separators connected to the particular structure of the intrinsic undirected graph can help identify the most significant bound.

Among planar DAGs, the Tree class consists of all connected DAGs for which all vertices have at most one successor. Possibly the earliest known separator theorem is a result of Jordan [13] which states that any tree can be partitioned into sub-trees of at most $2n/3$ vertices each, by the removal of a single vertex. In particular, the vertex that minimizes the maximum component size has this property, for if it did not, then its neighbor in the unique large sub-tree would form an even better partition.

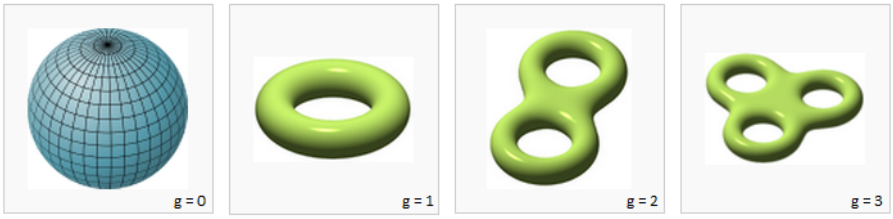

Figure 4.1: Surfaces of bounded genus

This result can be used in the same framework used in the proof of Theorem 2.2. In particular, using $m_L = 1$ will lead to obtain $d = \left\lceil \frac{\log_2 n}{\log_2 \frac{2}{3}} \right\rceil \in O(\log_2 n)$. The space complexity of a tree DAG $\overrightarrow{G^T}$ can be estimated as:

$$S_{free}\left(\overrightarrow{G^T}\right) \leq \sum_{i=1}^{c \log_2 n} 1 + 1 = c \log_2 n + 1 \in O(\log_2 n),$$

which corresponds to other results in the literature.

4.3 Applications to DAGs of known genus

We will now to now propose a similar result for non-planar graphs based on the concept of genus of a DAG.

Definition 4.5 (Genus of a graph)**.** The genus of a graph G_V is the minimal integer $g(G_V)$ such that the graph can be drawn without crossing itself on a sphere with $g(G_V)$ handles, or equivalently, $g(G_V)$ holes, (i.e. an oriented surface of genus $g(G_V)$).

Planar graph have genus 0 since a graph that can be drawn on the plane can be drawn on the sphere without self-crossing as well. Every graph has a genus, in fact a graph of size m can be surely embedded on a surface of genus m, therefore all graphs can be partitioned in classes whose elements share the same value of genus.

There are certain classes of graphs of particular interest for which it is possible to easily estimate the genus value of their elements:

- The genus of the complete graph is given by $g(K_n) = \left\lceil \frac{(n-3)(n-4)}{12} \right\rceil$, for $n \geq 3$,
- The genus of the complete bipartite graph is given by $g(K_{r,s}) = \frac{(r-2)(s-2)}{4}$, for $r, s \geq 2$,

- The genus of the n-cube is given by $g(Q_n) = (n-4)2^{n-3}+1$, for $n \geq 2$.

Among these, it is interesting to note that the graphs K_5 and $K_{3,3}$, used in Wagner's theorem as forbidden minors for the class of planar graphs [20], have genus 1 and thus can be drawn without edge-crossings on a toroidal surface.

The same definition can be applied for DAGs, and obviously a directed DAG $\overrightarrow{G_V}$ will have the same genus of its corresponding undirected intrinsic graph.

A result of particular relevance for our analysis has been proven by Gilbert, Hutchinson and Tarjan in 1984 [10], by using a similar approach to that used in the demonstration of the Planar Separator Theorem by Lipton and Tarjan.

Theorem 4.4 (Vertex separator for bounded genus graph)**.**
A graph with genus g with n vertices has a set of at most $6\sqrt{gn} + 2\sqrt{2n} + 1$ vertices whose removal leaves no component with more than $2n/3$ vertices.

For the purposes of our analysis, this result allows us to assume that for a generic graph G_V of genus g it is possible to find a $\frac{2}{3}$-balanced DAG-vertex separator X whose size will be $c\sqrt{gn} \in O\left(\sqrt{gn}\right)$ which generates the partition (L, X, R) of the vertices of V. Furthermore, the sub-graphs $G_{L\cup X}$ and $G_{X\cup R}$, defined in the same way already described for the planar case, will have at most $2n/3$ vertices each, and will still have at most genus g, and therefore they will maintain the same separator properties of G_V. The authors provide also an algorithm to identify such separator in time $O(gn)$.

As seen for the planar case, is possible to transfer these considerations on graphs to DAGs, by relying on the relation between a DAG $\overrightarrow{G_V}$ and its associated undirected intrinsic graph G_V. A partition (L, X, R) defined in G_V by means of a vertex separator X, corresponds to a partition (L, X, R) in $\overrightarrow{G_V}$ generated by a DAG-vertex separator X, constituted by the same vertices, which achieves the same balance and the sub-graphs $G_{L\cup X}$ and $G_{L\cup X}$ defined above correspond to the sub-DAGs $\overrightarrow{G_{L\cup X}}$ and $\overrightarrow{G_{L\cup X}}$ which will have the same number of vertices of the corresponding sub-graphs, with at most $Deg^-(\overrightarrow{G})\,|X|$ additional vertices.

Following the same line of reasoning described for the case of planar graphs we arrive at the following theorem:

Theorem 4.5 (Upper bound on space complexity for DAGs of bounded genus)**.**

Let $\overrightarrow{G_V}$ be any n-vertex planar graph of genus $g(\overrightarrow{G_V}) = g \geq 1$, $S_{free}(\overrightarrow{G_V}) \in O(\sqrt{gn})$.

Proof. The demonstration procedure will be very similar to the proof of Theorem 4.3. Theorem 3.3 leads to:

$$S_{free}(\overrightarrow{G_V}) \leq c\sqrt{gn} + \max\left(S_{free}(\overrightarrow{G_{L\cup X}}), S_{free}(\overrightarrow{G_{X\cup R}})\right).$$

The previous result can be further developed by applying the method based on the separator extraction to the sub-DAG with maximum space complexity between $\overrightarrow{G_{L\cup X}}$ and $\overrightarrow{G_{X\cup R}}$, whose size will be at most $\frac{2n}{3}$. Subsequent applications of this method will lead to obtain a tree separators decomposition of $\overrightarrow{G_V}$ where the application of Theorem 5.3 guarantees that the sizes of the sub-DAGs go down by a constant factor at each level, and there will therefore be a corresponding lowering of the upper bound on the number of nodes required to identify a the DAG-vertex separator. In particular, at the i-th depth level there will be at most 2^i sub-DAGs composed by at most $O\left(\left(\frac{2}{3}\right)^i n\right)$vertices and for whom a $\frac{2}{3}$-balanced separator of size $O\left(\left(\frac{2}{3}\right)^{-\frac{i}{2}}\sqrt{gn}\right)$may be found. The decomposition will cease when the size of the sub-DAGs extracted by a separator will be at most $m_L \in O\left(\sqrt{gn}\right)$, and thus each of the leaf components will surely be evaluable using $O\left(\sqrt{gn}\right)$ memory space.

The maximum number of levels d of the tree decomposition can be estimated as the smallest integer value for which $\left(\frac{2}{3}\right)^{-d}\sqrt{gn} \geq n$:

$$d = \left\lceil \frac{\log_2 \frac{n}{m_L}}{\log_2 \frac{3}{2}} \right\rceil \leq \left\lceil \frac{\log_2 \frac{n}{\sqrt{gn}}}{0.6} \right\rceil \in O\left(\log_2 \sqrt{n}\right).$$

The size of a path on the tree separator decomposition can thus be estimated as:

$$|P_{l^i}| \leq \sum_{i=0}^{log_2\sqrt{n}} c\sqrt{\left(\frac{2}{3}\right)^i gn} \leq c\sqrt{gn}\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^{i/2} \leq c3\left(1+\sqrt{\frac{2}{3}}\right)\sqrt{gn} \in O\left(\sqrt{gn}\right).$$

Resorting to Proposition 3.5, we can thus conclude that:

$$S_{free}(\overrightarrow{G_V}) \leq Deg^-(\overrightarrow{G_V})O\left(\sqrt{gn}\right) + O\left(\sqrt{gn}\right) \in O\left(\sqrt{gn}\right).$$

□

Observations

Theorem 4.5 allows to obtain a significant quantitative bound to the space complexity of any non-planar DAG whose genus is known.

Moreover, this result together with Theorem 4.3 highlights an interesting connection between space complexity and genus of a DAG, providing a criterion to approximate an upper bound of space complexity of any DAG as a function of just its genus.

Graphs of bounded genus constitutes an example of a family of graphs closed under the operation of taking minors, where an undirected graph H is called a minor of the graph G if H is isomorphic to a graph that can be obtained by zero or more edge contractions on a sub-graph of G. Among the separator theorems applying to arbitrary minor-closed graph families that were presented in literature, for our analysis it is of particular interest the result according to which if a graph family has a forbidden minor with h vertices, then it has a separator with $O\left(h\sqrt{n}\right)$ vertices [2].

Following the same approach used in Theorem 4.3 and 4.5, it will thus be possible to conclude that a DAG belonging to a family which has a forbidden minor with h vertices will surely be evaluable using at most $O\left(h\sqrt{n}\right)$ memory space.

4.4 Separators and sub-DAGs

In Chapter 2 we have seen how, within the marking rule approach, accurate lower bounds for the space complexity can be achieved by detecting space demanding sub-DAGs. We would like to resort to a similar approach while using the separators method to achieve tighter upper bounds.

The separator makes it possible to go to split the DAG in several parts for each of which it will then be possible to estimate the space complexity. Therefore, the greater the accuracy of the estimate of the space complexity of the sub-DAG, the higher the overall quality of the bound identified.

Thus, it may prove useful to select expressly vertex separators that make it possible to identify significant sub-DAGs whose space complexity is known.

Again, without affecting the generality of the method, it is evident how the more is known of the peculiar characteristics of a DAG, and in particular of its space demanding components, the more one can obtain an accurate estimate of

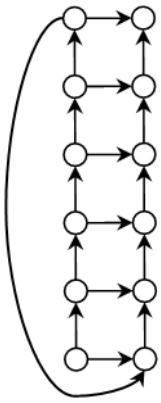

Figure 4.2: DAG B1

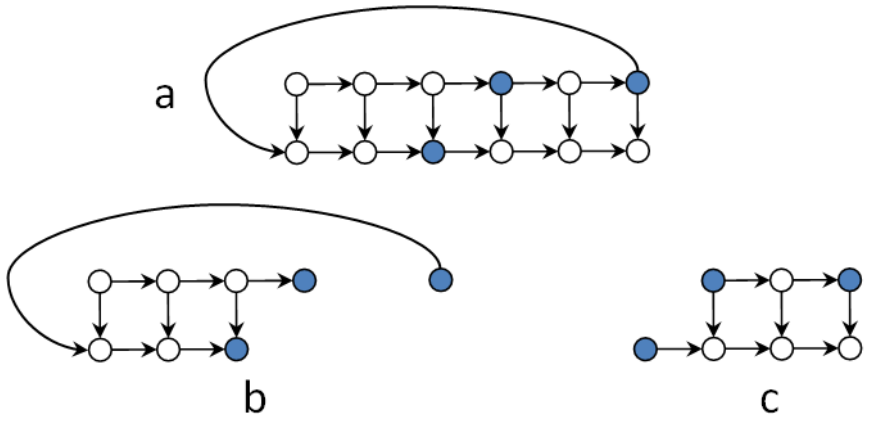

Figure 4.3: Decomposition of DAG B1 using vertex separator

the space complexity.

We can show an example of applying this approach by considering the B1 DAG (see Figure 4.2).

We can show an example of applying this approach by considering the n-vertex B1 DAG $\overrightarrow{G}$ which has been obtained by taking an $\frac{n}{2} \times 2$ directed mesh and adding a directed edge from the upper-left corner to the lower-right corner. This modifications makes the DAG connected, and thus it is possible to evaluate the minimum space needed for computations strictly without recalculations through the analysis of the total topological ordering of the vertices, as $S^{wr}\left(\overrightarrow{G^{B1}}\right) \geq \frac{n}{2}$.

Since the DAG is planar, Theorem 4.2 allows us to conclude that, by using recalculations, a computational strategy may be devised which will require at most $\sqrt{n}$ memory space, thus with a reduction of at least of a factor $O(\sqrt{n})$, compared to strictly non re-pebbling computations.

However an even better result can be achieved by using the separator approach to analyze sub-DAGS. In Figure 4.3, we propose a decomposition of the B1 DAG by finding a vertex-separator (highlighted in blue in Figure 4.3a) and identifying the sub-DAGs $\overrightarrow{G_{L\cup X}}$ in Figure 4.3b and $\overrightarrow{G_{X\cup R}}$ in Figure 4.3c.

It is easy to see that both the sub-DAGs are slight variations of the $\frac{n}{4} \times 2$ directed mesh which, as discussed in Chapter 2 with reference to the pebble game, will be evaluable using at most 2 pebbles. Thus Theorem, 3.4 leads to $S(\overrightarrow{G} \leq 3 + 2$. Beyond the quantitative value, the previous result highlights the fact that the memory space required for the evaluation of B1 is not related to the size of the DAG, and thus it is possible to conclude $S(\overrightarrow{G} \in O(1)$.

Chapter 5

Conclusions and points of interest for future developments

In this thesis we have studied some issues related to the space complexity of Directed Acyclic Graphs computations. In particular, the main objective, as set out in Chapter 1, was to find a relationship between the properties of a DAG and its space complexity, through which it is possible to get indications on the possibility of obtaining benefits, in terms of reducing the amount of memory necessary to the evaluation of the DAG, using computations with multiple assessments of the same vertex rather than computations strictly without recalculations.

In Chapter 2 we have seen, through the marking rule approach, how the necessity of taking into account the possible execution of recalculations greatly complicates the analysis of space complexity, especially if it is focused on the analysis of the possible computations.

The main result of this thesis was obtained in Chapter 3. Given a DAG $\overrightarrow{G}$, it is shown that a relationship exists between the space complexity $S(\overrightarrow{G})$ and the size of the vertex separators, defined with regard to its undirected intrinsic graph G. This is even more interesting in light of the analogous relationship between the minimum memory space required using computations strictly without recalculations $S^{wr}(\overrightarrow{G})$ and the size and topological separators which are defined, instead, with respect to the directed graph.

This result may be used in the divide and conquer paradigm to obtain a tree decomposition of the starting DAG, based on the subsequent identification of separators at each level. The upper bound thus obtained can be confronted with the value $S^{wr}(\overrightarrow{G})$, providing a sufficient but not necessary condition to determine

whether there actually are benefits associated with the use of computations with recalculations.

These results constitute a novelty compared to other upper bounds for space complexity known in the literature, for the use of the concept of separator and for the focus being put on the undirected intrinsic structure of the DAG.

Through the obtained results we managed to achieve most of the original Goals of the thesis while leaving the door open for further developments and future refinements. Among these, of particular interest it would be the possibility of formulating an algorithm that, given a DAG $\overrightarrow{G}$, finds, in polynomial (or polylogarithmic) time, a tree separator decomposition by using which it is possible to estimate the space complexity of $\overrightarrow{G}$ within a certain range of precision. This argument seems very promising, as it is encouraged by a similar result in which using topological separators it is possible to obtain, in polynomial time, a computation without recalculation using memory space at most $O(\log^2 n)$ times the optimum memory space achievable using only computations without recalculations [1].

An important goal going further would be the formulation of a sufficient condition that determines, in relation to the properties of the separator, if a DAG obtains benefits from the use of computations with recalculations.

Another direction worth exploring concerns the analysis of weighted DAGs which can be employed in order to study the trade-off between the balance achieved by a separator and the number of vertices which constitute it, or to represents programs for which the memory space required to store the values produced by the operations corresponding to the vertices is not the same.

The results seen in this thesis point out that there is a relationship between space complexity and a measure of the bandwidth of DAGs. It seems appropriate to further investigate this notion, in particular by exploiting the wealth of known results in graph theory.

In this sense, the concepts that express decomposition properties similar to those discussed for vertex separators are to be considered of interest, among them we cite edge separators (also called separations), tree decomposition, path decomposition and branch decomposition.

However, the issue of greatest interest that emerges from the work presented in this thesis concerns the relation between the space complexity of a DAG, its undirected intrinsic graph structure and the actual orientation of the edges of the DAGs. The results presented in this paper seem to suggest that the space com-

plexity is strongly linked to the characteristics of decomposition of the undirected intrinsic graph, while it substantially ignores the orientation of the edges. The observations presented here can not be considered conclusive, and further work should be dedicated to the study of this property, which, if it actually occurred, would constitute a major achievement in the study of DAGs.

Bibliography

[1] Ajit Agrawal, Philip Klein, and R. Ravi. Ordering problems approximated: Register sufficiency, single-processor scheduling and interval graph. Technical report, Providence, RI, USA, 1991.

[2] N. Alon, P. Seymour, and R. Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 293–299, New York, NY, USA, 1990. ACM.

[3] G. Bilardi and F. P. Preparata. Area-time lower-bound techniques with applications to sorting. *Algorithmica*, 1(1):65–91, January 1986.

[4] G. Bilardi and F. P. Preparata. Processor time tradeoffs under bounded-speed message propagation: Part i, upper bounds. *Theory of Computing Systems*, 30:523–546, 1997. 10.1007/s002240000066.

[5] G. Bilardi and F. P. Preparata. Processor time tradeoffs under bounded-speed message propagation: Part ii, lower bounds. *Theory of Computing Systems*, 32:531–559, 1999. 10.1007/s002240000131.

[6] Gianfranco Bilardi, Andrea Pietracaprina, and Paolo D'Alberto. On the space and access complexity of computation dags. In Ulrik Brandes and Dorothea Wagner, editors, *Graph-Theoretic Concepts in Computer Science*, volume 1928 of *Lecture Notes in Computer Science*, pages 81–92. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40064-8-6.

[7] Stephen A. Cook. An observation on time-storage trade off. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 29–33, New York, NY, USA, 1973. ACM.

[8] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[9] Hristo Nicolov Djidjev. On the problem of partitioning planar graphs. *SIAM Journal on Algebraic and Discrete Methods*, 3(2):229–240, 1982.

[10] John R. Gilbert, Joan P. Hutchinson, and Robert Endre Tarjan. A separator theorem for graphs of bounded genus. *J. Algorithms*, 5(3):391–407, September 1984.

[11] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *J. ACM*, 24(2):332–337, April 1977.

[12] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[13] C. Jordan. Sur les assemblages de lignes. *J. Reine Angew Math*, 70:185–190, 1869.

[14] Thomas Lengauer and Robert E. Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *J. ACM*, 29(4):1087–1130, October 1982.

[15] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[16] Michael S. Paterson and Carl E. Hewitt. Record of the project mac conference on concurrent systems and parallel computation. chapter Comparative schematology, pages 119–127. ACM, New York, NY, USA, 1970.

[17] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, STOC '76, pages 149–160, New York, NY, USA, 1976. ACM.

[18] Desh Ranjan, John Savage, and Mohammad Zubair. Upper and lower i/o bounds for pebbling r-pyramids. In *Proceedings of the 21st international conference on Combinatorial algorithms*, IWOCA'10, pages 107–120, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[20] K. Wagner. Ãber eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114:570–590, 1937. 10.1007/BF01594196.