Toward More Localized Local Algorithms: Removing Assumptions Concerning Global Knowledge

Amos Korman* Jean-Sébastien Sereni†
Laurent Viennot‡

*CNRS and Univ. Paris Diderot, Paris, 75013, France. E-mail: Amos.Korman@liafa.jussieu.fr. Supported in part by a France-Israel cooperation grant ("Mutli-Computing" project) from the France Ministry of Science and Israel Ministry of Science, by the ANR projects ALADDIN and PROSE, and by the INRIA project GANG.

†CNRS (LIAFA, Université Denis Diderot), Paris, France and Department of Applied Mathematics (KAM), Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic (sereni@kam.mff.cuni.cz). This author's work was partially supported by the French *Agence Nationale de la Recherche* under reference ANR 10 JCJC 0204 01.

‡INRIA and Paris Diderot University, Paris, 75013, France. E-mail: Laurent.Viennot@inria.fr. Supported by the european STREP project "EULER", and the INRIA Project-team "GANG".

Abstract

Numerous sophisticated local algorithm were suggested in the literature for various fundamental problems. Notable examples are the MIS and $(\Delta + 1)$-coloring algorithms by Barenboim and Elkin [6], by Kuhn [22], and by Panconesi and Srinivasan [34], as well as the $O(\Delta^2)$-coloring algorithm by Linial [28]. Unfortunately, most known local algorithms (including, in particular, the aforementioned algorithms) are *non-uniform*, that is, they assume that all nodes know good estimations of one or more global parameters of the network, e.g., the maximum degree Δ or the number of nodes n.

This paper provides a rather general method for transforming a non-uniform local algorithm into a *uniform* one. Furthermore, the resulting algorithm enjoys the same asymptotic running time as the original non-uniform algorithm. Our method applies to a wide family of both deterministic and randomized algorithms. Specifically, it applies to almost all of the state of the art non-uniform algorithms regarding MIS and Maximal Matching, as well as to many results concerning the coloring problem. (In particular, it applies to all aforementioned algorithms.)

To obtain our transformations we introduce a new distributed tool called *pruning algorithms*, which we believe may be of independent interest.

Keywords: distributed algorithm, global knowledge, parameters, MIS, coloring, maximal matching.

1 Introduction

1.1 Background and Motivation

Distributed computing concerns environments in which many processors, located at different sites, must collaborate in order to achieve some global task. One of the main themes in distributed network algorithms concerns the question of how to cope with *locality* constrains, that is, the lack of knowledge about the global structure of the network (cf., [35]). On the one hand, information about the global structure may not always be accessible to individual processors and the cost of computing it from scratch may overshadow the cost of the algorithm using it. On the other hand, global knowledge is not always essential, and many seemingly global tasks can be efficiently achieved by letting processors know more about their immediate neighborhoods and less about the rest of the network.

A standard model for capturing the essence of locality is the $\mathcal{LOCAL}$ model (cf., [35]). In this model, the network is modeled by a graph $G = (V, E)$, where the nodes of G represent the processors and the edges represent the communication links. To perform a task, nodes are woken up simultaneously, and computation proceeds in fault-free synchronous rounds during which every node exchanges messages with its neighbors, and performs arbitrary computations on its data. Since many tasks cannot be solved distributively in an anonymous network, symmetry breaking must be addressed. The typical way to address this issue is by assuming that a unique identity $\mathrm{Id}(v)$ is initially provided to each node v in the network, and encoded using $O(\log n)$ bits, where n is the number of nodes in the network. A *local algorithm* operating in such a setting must return an output at each node such that the collection of outputs satisfies the required task. For example, in the Maximal Independent Set (MIS) problem, the output at each node v is a bit $b(v)$ indicating whether v belongs to a selected set $S \subseteq V$ of nodes, and it is required that S forms a MIS of G. The *running time* of a local algorithm is the number of rounds needed for the algorithm to complete its operation at each node, taken in the worst case scenario. This is typically evaluated with respect to some parameters of the underlying graph. The common parameters used are the number of nodes n in the graph and the maximum degree Δ of a node in the graph.

To ease the computation, it is often assumed that some kind of knowledge about the global network is provided to each node *a priori*. A typical example of such knowledge is the number of nodes n in the network. It turns out that in some cases, this (common) assumption can give a lot of power to the distributed algorithm. This was observed by Fraigniaud *et al.* [16] in the context of local decision: they introduced the complexity class of decision problems NLD, which contains all decision problems that can be verified in constant time with the aid of a certificate. They proved that, although there exist decision problems that do not belong to NLD, every (decidable) decision problem falls in NLD if it is assumed that each node knows the value of n.

In general, the amount and type of such information may have a profound effect on the design of the distributed algorithm. Obviously, if the whole topology of the network is known to each node in advance, then the distributed algorithm can be reduced to a central one. In fact, the whole area of *computation with advice* [9, 12, 13, 14, 15, 20, 21] is dedicated to studying the amount of information known to nodes and its effect on the performances of the distributed algorithm. For instance, Fraigniaud *et al.* [15] showed that if each node is provided with only a constant number of bits then one can locally construct a BFS-tree in constant time, and can locally construct a MST in $O(\log n)$ time, while both tasks require diameter time if no knowledge is assumed. As another example, Cohen *et al.* [9] proved that $O(1)$ bits, judiciously chosen at each node, can allow a finite automata to distributively explore every graph. As a matter of fact, from a radical point of view, for many questions (e.g., MIS and Maximal Matching), additional information may push the question at hand into absurdity: even a constant number of bits of additional information per node is enough to compute a solution—simply let the additional information encode the solution!

When dealing with locality issues, it is desired that the amount of information that is known to nodes regarding the whole network is minimized. A local algorithm that assumes that each node initially knows merely its own identity is often called *uniform*. Unfortunately, there are only few local algorithms in the literature that are uniform (e.g., [11, 26, 29, 30, 37]). In contrast, most known local algorithms assume that all nodes know upper bounds on the values of some global parameters of the network. Moreover, it is often assumed that all nodes agree on their candidates for being these upper bounds. Furthermore, typically, not only the correct operation of the algorithm requires the knowledge of the upper bounds, but also its running time is actually a function of the upper bound estimations and not of the actual value of the parameters. Hence, it is desired that the known upper bounds are not significantly larger than the real values of the parameters.

Some attempts to transform a non-uniform local algorithm into a uniform one were made by examining the details of the algorithm at hand and modifying it appropriately. For example, Barenboim and Elkin [6] first gave a non-uniform MIS algorithm for the family of graphs with arboricity $a = O(\log^{1/2-\delta} n)$, for some constant $\delta \in (0, 1/2)$, running in time $O(\log n/\log\log n)$. At the cost of increasing the running time to $O(\frac{\log n}{\log\log n}\log^* n)$, the authors show how to modify their algorithm to not require the knowledge of a. (Nevertheless, their algorithm still requires nodes to agree on an upper bound on n.)

In this paper, we present a rather general method for transforming a non-uniform local algorithm into a uniform one without increasing the asymptotic running time of the original algorithm. Our method can apply to a wide family of both deterministic and randomized algorithms. In particular, our method can apply to all of the state of the art non-uniform algorithms regarding MIS and Maximal Matching, as well as to several of the best known results concerning the coloring problem.

Our transformations are obtained using a new type of local algorithms termed *pruning algorithms.* Informally, the basic property of a pruning algorithm is that it allows one to iteratively apply a sequence of local algorithms (whose output may not form a correct global solution) one after the other, in a way that "always progresses" toward a solution. In a sense, a pruning algorithm is a combination of a gluing mechanism and a *local checking* algorithm (cf., [16, 32]). A local checking algorithm for a problem Prob runs on graphs with an output value at each node (and possibly an input too), and can locally detect whether the output is "legal" with respect to Prob. That is, if the instance is not legal then at least one node detects this, and raises an alarm. (For example, a local checking algorithm for MIS is trivial: each node in the set S, which is suspected to be a MIS, checks that none of its neighbors belongs to S, and each node not in S checks that at least one of its neighbors belongs to S. If the check fails, then the node raises an alarm.) A pruning algorithm needs to satisfy an additional *gluing* property not required by local checking algorithms. Specifically, if the instance is not legal, then the pruning algorithm must carefully choose the nodes raising the alarm (and possibly modify their input too), so that a solution for the subgraph induced by those alarming nodes can be well glued to the previous output of the non-alarming nodes, in a way such that the combined output is a solution to the problem for the whole initial graph.

We believe that this new type of algorithms may be of independent interest. Indeed, as we show, pruning algorithms have several types of other applications in the theory of local computation, besides the aforementioned issue of designing uniform algorithms. Specifically, they can be used also to transform a local Monte-Carlo algorithm into a Las Vegas one, as well as to obtain an algorithm that runs in the minimum running time of a given set of uniform algorithms.

Problem	Parameters	Time	Ref.	This paper (uniform)	Theorem
Det. MIS and $(\Delta{+}1)$-coloring	n, Δ	$O(\Delta + \log^* n)$	[4, 22]	$\min\left\{O(\Delta + \log^* n), 2^{O(\sqrt{\log n})}\right\}$	Th. 1
	n	$2^{O(\sqrt{\log n})}$	[34]		Th. 2
Det. MIS (arboricity $a = o(\sqrt{\log n})$)	n, a	$o(\log n)$	[6]	$o(\log n)$	Th. 1
Det. MIS (arboricity $a = O(\log^{1/2-\delta} n)$)	n, a	$O(\log n / \log\log n)$	[6]	$O(\log n / \log\log n)$	Th. 1
Det. $\lambda(\Delta + 1)$-coloring	n, Δ	$O(\Delta/\lambda + \log^* n)$	[4, 22]	$O(\Delta/\lambda + \log^* n)$	Th. 3
Det. $O(\Delta)$-edge coloring	n, Δ	$O(\Delta^\epsilon + \log^* n)$	[7]	$O(\Delta^\epsilon + \log^* n)$	Th. 5
Det. $O(\Delta^{1+\epsilon})$-edge coloring	n, Δ	$O(\log \Delta + \log^* n)$	[7]	$O(\log \Delta + \log^* n)$	Th. 5
Det. Maximal Matching	n or Δ	$O(\log^4 n)$	[19]	$O(\log^4 n)$	Th. 6
Rand. MIS	uniform	$O(\log n)$	[30, 1]		
Rand. $(2, 2(c+1))$-ruling-set	n	$O(2^c \log^{1/c} n)$	[36]	$O(2^c \log^{1/c} n)$	Th. 7

Table 1: Comparison of $\mathcal{LOCAL}$ algorithms with respect to global parameter knowledge. "Det." stands for deterministic, and "Rand." for randomized.

1.2 Previous Work

MIS and coloring: There is a long line of research concerning the two related $(\Delta+1)$-coloring and MIS problems [3, 10, 17, 18, 23, 24, 28]. Recently, Barenboim and Elkin [4] and independently Kuhn [22] presented two elegant $(\Delta + 1)$-coloring and MIS algorithms running in $O(\Delta + \log^* n)$ time on general graphs. This is the current best bound for these problems on low degree graphs. For graphs with a large maximum degree Δ, the best bound is due to Panconesi and Srinivasan [34], who devised an algorithm running in $2^{O(\sqrt{\log n})}$ time. The aforementioned algorithms are not uniform. Specifically, all three algorithms require that all nodes know and agree on an upper bound on n and the first two also require an upper bound on Δ.

For bounded-independence graphs, Schneider and Wattenhofer [37] designed uniform deterministic MIS and $(\Delta + 1)$-coloring algorithms running in $O(\log^* n)$ time. Barenboim and Elkin devised [6] a deterministic algorithm for the MIS problem on graphs of bounded arboricity that requires time $O(\log n/\log\log n)$. More specifically, for graphs with arboricity $a = o(\sqrt{\log n})$, they show that a MIS can be computed deterministically in $o(\log n)$ time, and whenever $a = O(\log^{1/2-\delta} n)$ for some constant $\delta \in (0\,,1/2)$, the same algorithm runs in time $O(\log n/\log\log n)$. At the cost of increasing the running time by a multiplicative factor of $O(\log^* n)$, the authors show how to modify their algorithm to not require the knowledge of a. Nevertheless, all their algorithms require all nodes to know and agree on an upper bound on the value of n. Another MIS algorithm which is efficient for graphs with low arboricity was devised by Barenboim and Elkin [5]; this algorithm runs in time $O(a + a^\epsilon \log n)$ for arbitrary $\epsilon > 0$.

Concerning the problem of coloring with more than $\Delta + 1$ colors, Linial [27, 28], and subsequently Szegedy and Vishwanathan [38], described $O(\Delta^2)$-coloring algorithms with running time $\Theta(\log^* n)$. Barenboim and Elkin [4] and, independently, Kuhn [22] generalized this by presenting a tradeoff between the running time and the number of colors: they devised a $\lambda(\Delta+1)$-coloring algorithm with running time $O(\Delta/\lambda + \log^* n)$, for any $\lambda \geqslant 1$. All these algorithms require the knowledge of upper bounds on both n and Δ.

Barenboim and Elkin [5] devised a $\Delta^{1+o(1)}$ coloring algorithm running in time $O(f(\Delta)\log\Delta\log n)$, for an arbitrarily slow-growing function $f(\Delta) = \omega(1)$. They also produced an $O(\Delta^{1+\epsilon})$-coloring algorithm running in $O(\log\Delta\log n)$-time, for an arbitrarily small constant $\epsilon > 0$, and an $O(\Delta)$-coloring algorithm running in $O(\Delta^{\epsilon}\log n)$ time, for an arbitrarily small constant $\epsilon > 0$. All these coloring algorithms require the knowledge of both Δ and n. Other deterministic non-uniform coloring algorithms with number of colors and running time corresponding to the arboricity parameter were given by Barenboim and Elkin [5, 6].

Efficient deterministic algorithms for the edge-coloring problem can be obtained from [5, 7, 33]. The state of the art results are due to Barenboim and Elkin [7], who designed an $O(\Delta)$-edge coloring algorithm running in time $O(\Delta^{\epsilon}) + \log^* n$, for any $\epsilon > 0$, and an $O(\Delta^{1+\epsilon})$-edge coloring algorithm running in time $O(\log\Delta) + \log^* n$, for any $\epsilon > 0$. Both these algorithms require the knowledge of upper bounds on both n and Δ.

Randomized algorithms for MIS and $(\Delta + 1)$-coloring running in expected time $O(\log n)$ were initially given by Luby [30] and, independently, by Alon *et al.* [1].

Recently, Schneider and Wattenhofer [36] constructed the best known non-uniform $(\Delta + 1)$-coloring algorithm, which runs in time $O(\log\Delta + \sqrt{\log n})$. They also provided random algorithms for coloring using more colors. For every positive integer c, a randomized algorithm for $(2, 2(c + 1))$-ruling-set running in time $O(2^c \log^{1/c} n)$ is also presented. All these algorithms of Schneider and Wattenhofer [36] are not uniform and require the knowledge of an upper bound on n.

Maximal Matching: Schneider and Wattenhofer [37] designed a uniform deterministic maximal matching algorithm on bounded-independence graphs running in $O(\log^* n)$ time. For general graphs, however, the state of the art maximal matching algorithm is not uniform: Hanckowiak *et al.* [19] presented a non-uniform deterministic algorithm for maximal matching running in time $O(\log^4 n)$. This algorithm assumes the knowledge of an upper bound for n (for some parts of the algorithm, the nodes can ignore n provided they know Δ).

1.3 Our Results

The main conceptual contribution of the paper is the introduction of a new type of algorithms called *pruning algorithms*. Informally, the fundamental property of this type of algorithms is to allow one to iteratively run a sequence of algorithms (whose output may not necessarily be correct everywhere) so that the global output does not deteriorate, and it always progresses toward a solution.

Our main application for pruning algorithm concerns the problem of locally computing a global solution while minimizing the necessary global knowledge known to nodes. Addressing this, we provide a rather general method for transforming a non-uniform local algorithm into a uniform one without increasing the asymptotic running time of the original algorithm. Our method applies to a wide family of both deterministic and randomized algorithms; in particular, it applies to many of the best known results concerning classical problems such as MIS, Coloring, and Maximal Matching. (See table 1.1 for a summary of some of the uniform algorithms we obtain and the corresponding state of the art existing non-uniform algorithms.)

In another application, we show how to transform a Monte-Carlo local algorithm into a Las Vegas one. Finally, given several uniform algorithms for the same problem whose running times depend on different parameters—which are unknown to nodes—we show a general method for constructing a uniform algorithm solving the problem, that on every instance runs asymptotically as fast as the fastest algorithm among those given algorithms. In particular, we obtain the following theorems.

Theorem 1. *There exists a uniform deterministic algorithm solving* `MIS` *on general graphs in time*

$$\min\left\{O(\Delta + \log^* n), 2^{O(\sqrt{\log n})}, f(a)\right\},$$

where $f(a) := o(\log n)$ *for graphs of arboricity* $a = o(\sqrt{\log n})$*, and* $f(a) := O(\log n/\log\log n)$ *for arboricity* $a = O(\log^{1/2-\delta} n)$*, for some constant* $\delta \in (0\,,1/2)$*; and otherwise:* $f(a) := O(a\log a + a^\epsilon \log n \log a)$*, for arbitrary small constant* $\epsilon > 0$*.*

Theorem 2. *There exists a uniform deterministic algorithm solving the* $(\Delta+1)$*-coloring problem on general graphs in time* $\min\{O(\Delta + \log^* n), 2^{O(\sqrt{\log n})}\}$*.*

Theorem 3. *There exists a uniform deterministic algorithm solving the* $\lambda(\Delta+1)$*-coloring problem on general graphs and running in time* $O(\Delta/\lambda + \log^* n)$*, for any* $\lambda \geqslant 1$*, such that* Δ/λ *is a moderately-slow function. In particular, there exists a uniform deterministic algorithm solving the* $O(\Delta^2)$*-coloring problem in time* $O(\log^* n)$*.*

Theorem 4. *Let* $\epsilon > 0$ *be a constant. The following uniform deterministic coloring algorithms exist.*

- *A uniform* $\Delta^{1+o(1)}$*-coloring algorithm running in time* $O(f(\Delta)\log\Delta\log n\log\log n)$*, for an arbitrarily slow-growing function* $f(\Delta) = \omega(1)$*.*
- *A uniform* $O(\Delta^{1+\epsilon})$*-coloring algorithm running in* $O(\log\Delta\log n\log\log n)$ *time.*
- *A uniform* $O(\Delta)$*-coloring algorithm running in* $O(\Delta^\epsilon \log n\log\log n)$ *time.*

Theorem 5. *(1) There exists a uniform deterministic* $O(\Delta)$*-edge coloring algorithm for general graphs running in time* $O(\Delta^\epsilon + \log^* n)$*, for any* $\epsilon > 0$*. (2) There exists a uniform deterministic* $O(\Delta^{1+\epsilon})$*-edge coloring algorithm for general graphs that runs in time* $O(\log\Delta + \log^* n)$*, for any* $\epsilon > 0$*.*

Theorem 6. *There exists a uniform deterministic algorithm solving the maximal matching problem in time* $O(\log^4 n)$.

Theorem 7. *For a constant integral* $c > 0$, *there exists a uniform randomized algorithm solving the* $(2, 2(c+1))$*-ruling-set problem in time* $O(2^c \log^{1/c} n)$.

2 Preliminaries

General definitions: For two integers a and b, we let $[a\,,b] \coloneqq \{a, a+1, \cdots, b\}$. For an undirected and unweighted graph G, we let $V(G)$ and $E(G)$ be the vertex set and edge set of G, respectively. The *degree* $\deg_G(v)$ of a node $v \in V(G)$ is the number of neighbors of v in G. The *maximum degree of* G is $\Delta_G \coloneqq \max\{\deg_G(v) \mid v \in V(G)\}$. The *distance* $\mathrm{dist}_G(u, v)$ between two nodes $u, v \in G$ is the number of edges on a shortest path connecting them. Given a node u and an integer r, the *ball* of radius r around u is the subgraph $B_G(u, r)$ of G induced by the collection of nodes at distance at most r from v. The *neighborhood* $N_G(u)$ of u is the set of neighbors of u, i.e., $N_G(u) \coloneqq B_G(u, 1) \setminus \{u\}$. In what follows, we may omit the subscript G from the previous notations when there is no risk of confusion.

Functions: Fix an integer k. A function $f : \mathbf{R}^k \to \mathbf{R}$ is *non-decreasing* if $f(x_1, x_2, \cdots, x_k) \leqslant f(y_1, y_2, \cdots, y_k)$ for any two sequences $(x_1, x_2, \cdots, x_k)$ and $(y_1, y_2, \cdots, y_k)$ where $x_i \leqslant y_i$ for each $i \in [1\,, k]$. An *ascending* function is a non-decreasing and unbounded function $f : \mathbf{R} \to \mathbf{R}^+$ (by unbounded, we mean that $\lim_{x\to\infty} f(x) = \infty$). A function $f : (\mathbf{R}^+)^\ell \to \mathbf{R}^+$ is *additive* if $f(x_1, \cdots, x_\ell) = \sum_{i=1}^{\ell} f_i(x_i)$ where f_i is ascending for each $i \in [1, \ell]$.

A function $f : \mathbf{R}^+ \to \mathbf{R}^+$ is *moderately-increasing* if it is increasing and there exists a positive integer α such that $f(\alpha i) > 2f(i)$ and $\alpha f(i) > f(2i)$ for every integer $i \geqslant 2$. Note that $f(x) = x^{k_1} \log^{k_2}(x)$

is a moderately-increasing function for every non-negative constants k_1 and k_2 (such that $k_1 + k_2 > 0$).

A function $f : \mathbf{R}^+ \to \mathbf{R}^+$ is *moderately-slow* if it is non-decreasing and there exists a positive integer α such that $\alpha f(i) > f(2i)$ for every integer $i \geqslant 2$. In other words, $f(c \cdot i) = O(f(i))$ for every constant c and every integer i, where the constant hidden in the O notation depends only on c. Note that every moderately-increasing function is moderately-slow. On the other hand, there are natural functions that are moderately-slow (such as the constant functions) but are not moderately-increasing.

Consider a function $f : \mathbf{N}^\ell \to \mathbf{R}^+$. The following definition gives a certain measure for the "separation" between the variables in f. A *sequence-number* function $s_f : \mathbf{N} \to \mathbf{N}$ for f is a moderately-slow function for which there exists a constant $c > 0$, referred to as a the *bounding* constant, such that for every $i \in \mathbf{N}$, the value $s_f(i)$ upper bounds the size J of a (possibly empty) finite set of sequences $S_f(i) \coloneqq \{(x_1^j, x_2^j, \cdots, x_\ell^j)\}_{j \in [1,J]}$ satisfying the following two properties.

1. $f(x_1^j, x_2^j, \cdots, x_\ell^j) \leqslant c \cdot i$ for every $j \in [1, J]$, and
2. if $f(x_1, x_2, \cdots, x_\ell) \leqslant i$, then $\exists j \in [1, J]$ such that $x_k \leqslant x_k^j$ for every $k \in [1\,, \ell]$.

For example, consider the case where $f : \mathbf{N}^\ell \to \mathbf{R}$ is additive, i.e., $f(x_1, x_2, \cdots, x_\ell) \coloneqq \sum_{k=1}^{\ell} f_k(x_k)$, where f_k is a non-negative ascending function for each $k \in [1\,, \ell]$. Here, the constant function 1 is a sequence-number function for f. Indeed, for $i \in \mathbf{N}$, let $S_f(i) \coloneqq \{(x_1^1, x_2^1, \cdots, x_\ell^1)\}$, where x_k^1 is the largest integer y such that $f_k(y) \leqslant i$, for each $k \in [1, \ell]$ (if such an integer y exists, otherwise, $S_f(i)$ is empty). Thus Condition (1) above is satisfied with $c = \ell$, and if $(x_1, x_2, \cdots, x_\ell) \in \mathbf{N}^\ell$ such that $f(x_1, x_2, \cdots, x_\ell) \leqslant i$ then, since each function f_k is non-negative, we deduce that $f_k(x_k) \leqslant i$ for each $k \in [1\,, \ell]$. Hence $x_k \leqslant x_k^1$, as required by Condition (2).

As another example, consider the case where $f : \mathbf{N}^2 \to \mathbf{R}$ is given by $f(x_1, x_2) \coloneqq f_1(x_1) \cdot f_2(x_2)$, where f_ℓ is an ascending function with $f_\ell \geqslant 1$ for each $\ell \in [1\,,2]$. Then, the function $s_f(i) := \lceil \log i \rceil + 1$ is a sequence-number for f. Indeed, for $i \in \mathbf{N}$ let $S_f(i) := \{(x_1^j, x_2^j)\}_{j\in[0,\lceil \log i\rceil]}$ where x_1^j is the largest integer y_1 such that $f_1(y_1) \leqslant 2^j$ and x_2^j is the largest integer y_2 such that $f_2(y_2) \leqslant 2^{\log i - j+1}$ for each $j \in [0\,, \lceil \log i \rceil]$ (if such integers y_1 and y_2 exist, otherwise we do not define the pair (x_1^j, x_2^j)). Again, a straightforward check ensures that both Conditions (1) and (2) hold (with bounding constant $c = 2$).

Observation 2.1

- *The constant function* 1 *is a sequence-number function for any additive function.*
- *Let* $f : \mathbf{N}^2 \to \mathbf{R}$ *be a function given by* $f(x_1, x_2) := f_1(x_1) \cdot f_2(x_2)$, *where* $f_1 \geqslant 1$ *and* $f_2 \geqslant 1$ *are ascending functions. Then, the function* $s_f(i) := \lceil \log i \rceil + 1$ *is a sequence-number function for* f.

Finally, note that not all functions have a bounded sequence-number function, as one can see by considering the min function over $\mathbf{N}^2$.

Problems and instances: Given a set V of nodes, a *vector* for V is an assignment $\mathbf{x}$ of a bit string $\mathbf{x}(v)$ to each $v \in V$, i.e., $\mathbf{x}$ is a function $\mathbf{x} : V \to \{0,1\}^*$. A *problem* is defined by a collection of triplets: Prob $= \{(G, \mathbf{x}, \mathbf{y})\}$, where $G = (V, E)$ is a (not necessarily connected) graph, and $\mathbf{x}$ and $\mathbf{y}$ are *input* and *output* vectors for V, respectively. We consider only problems that are closed under disjoint union, i.e., if G_1 and G_2 are two vertex disjoint graphs and $(G_1, \mathbf{x}_1, \mathbf{y}_1), (G_2, \mathbf{x}_2, \mathbf{y}_2) \in$ Prob then $(G, \mathbf{x}, \mathbf{y}) \in$ Prob, where $G := G_1 \cup G_2$, $\mathbf{x} := \mathbf{x}_1 \cup \mathbf{x}_2$ and $\mathbf{y} := \mathbf{y}_1 \cup \mathbf{y}_2$. An *instance*, with respect to a given a problem Prob, is a pair $(G, \mathbf{x})$ for which there exists an output vector $\mathbf{y}$ satisfying $(G, \mathbf{x}, \mathbf{y}) \in$ Prob. In what follows, whenever we consider some collection $\mathcal{F}$ of instances, we always assume that $\mathcal{F}$ is closed under inclusion. That is, if $(G, \mathbf{x}) \in \mathcal{F}$ and $(G', \mathbf{x}') \subseteq (G, \mathbf{x})$ (i.e., G' is a subgraph of G and $\mathbf{x}'$ is the input vector $\mathbf{x}$ restricted to $V(G')$) then $(G', \mathbf{x}') \in \mathcal{F}$. Informally, given a problem Prob and a collection of instances $\mathcal{F}$, the goal is to design an efficient distributed algorithm that takes an instance $(G, \mathbf{x}) \in \mathcal{F}$ as input, and produces an output vector $\mathbf{y}$ satisfying $(G, \mathbf{x}, \mathbf{y}) \in$ Prob. The reason why we require Prob to be closed under disjoint union is that a distributed algorithm operating on an instance $(G, \mathbf{x})$ behaves separately and independently on each connected component of G. Let $\mathcal{G}$ be a family of graphs closed under inclusion. We define $\mathcal{F}(\mathcal{G})$ to be the set of pairs $\{(G, \mathbf{x}) \mid G \in \mathcal{G},\ \mathbf{x} \text{ is arbitrary}\}$.

We assume that each node $v \in V$ is provided with a unique integer referred to as the *identity* of v, and denoted $\mathrm{Id}(v)$, encoded using $O(\log |V|)$ bits; by unique identities, we mean that $\mathrm{Id}(u) \neq \mathrm{Id}(v)$ for every two distinct nodes u and v. (In some works on coloring, the assumption that the identities are unique is relaxed and the collection of assigned identities is only required to be a coloring, i.e., every two neighboring nodes have different identities.) For ease of exposition, we consider the identity of a node to be part of its input.

We consider classical problems such as coloring, (α, β)-ruling set (and in particular MIS, which is $(2, 1)$-ruling set) and maximal matching. Informally, viewing the output of a node as a *color*, the requirement of *coloring* is that the colors of two neighboring nodes must be different. In *MIS* (for Maximal Independent Set), the output at each node is Boolean, and indicates whether the node belongs to a set S that must form a MIS, that is, S must be independent: no two neighboring nodes are in S, and must be maximal: if all neighbors of a node v are not in S then $v \in S$. In (α, β)-ruling set, the set S of selected nodes must satisfy: (1) two nodes that belong to S must be at distance at least α from each other, and (2) if a node does not belong to S, then there is a node in the set at distance at most β from it. (Observe that `MIS` is precisely $(2, 1)$-ruling set.) Finally, given a triplet $(G, \mathbf{x}, \mathbf{y})$, two nodes u and v are said to be *matched* if $(u, v) \in E$, $\mathbf{y}(u) = \mathbf{y}(v)$ and $\mathbf{y}(w) \neq \mathbf{y}(u)$ for every $w \in (N_G(u) \cup N_G(v)) \setminus \{u, v\}$. In `MM` (for Maximal Matching) it is required that each node u is either matched to one of its neighbors or that every neighbor v of u is matched to one of v's neighbors.

Parameters: Fix a problem Prob and let $\mathcal{F}$ be a collection of instances for Prob. A *parameter* $\mathtt{p}$ is a non-decreasing positive valued function $\mathtt{p} : \mathcal{F} \to \mathbf{N}$. By non-decreasing, we mean that if $(G', \mathbf{x}') \in \mathcal{F}$ and $(G', \mathbf{x}') \subseteq (G, \mathbf{x})$ then $\mathtt{p}(G', \mathbf{x}') \leqslant \mathtt{p}(G, \mathbf{x})$.

Let $\mathcal{F}$ be a collection of instances. A parameter $\mathtt{p}$ for $\mathcal{F}$ is called a *graph-parameter* if $\mathtt{p}$ is oblivious of the input, that is, if $\mathtt{p}(G, \mathbf{x}) = \mathtt{p}(G, \mathbf{x}')$ for every two instances $(G, \mathbf{x}), (G, \mathbf{x}') \in \mathcal{F}$ such that the input assignments $\mathbf{x}$ and $\mathbf{x}'$ preserve the identities, i.e., the inputs $\mathbf{x}(v)$ and $\mathbf{x}'(v)$ contain the same identity $\mathrm{Id}(v)$ for every $v \in V(G)$. For example, in what follows, we will focus on the following graph-parameters: the number n of nodes of the graph G, i.e., $|V(G)|$, the maximum degree $\Delta = \Delta(G)$ of G, i.e., $\max\{\deg_G(u) \mid u \in V(G)\}$, and the arboricity $a = a(G)$ of G, i.e., the least number of edge-disjoint forests whose union is G.

Local algorithms: Consider a problem Prob and a collection of instances $\mathcal{F}$ for Prob. An algorithm for Prob and $\mathcal{F}$ takes as input an instance $(G, \mathbf{x}) \in \mathcal{F}$ and must terminate with an output vector $\mathbf{y}$ such that $(G, \mathbf{x}, \mathbf{y}) \in$ Prob. We consider the $\mathcal{LOCAL}$ model (cf., [35]). During the execution of a *local* algorithm $\mathcal{A}$, all processors are woken up simultaneously and computation proceeds in fault-free synchronous rounds, i.e., it occurs in discrete rounds. In each round, every node may send messages of unrestricted size to its neighbors and may perform arbitrary computations on its data. A message that is sent in a round r, arrives to its destination before the next round $r + 1$ starts. It must be guaranteed that after a finite number of rounds, each node v terminates with some output value $\mathbf{y}(v)$. (It is required that a node knows that its output is indeed its final output.) The algorithm $\mathcal{A}$ is *correct* if for every instance $(G, \mathbf{x}) \in \mathcal{F}$, the resulted output vector $\mathbf{y}$ satisfies $(G, \mathbf{x}, \mathbf{y}) \in \text{Prob}$.

Let $\mathcal{A}$ be a local deterministic algorithm for Prob and $\mathcal{F}$. The *running time* of $\mathcal{A}$ over a particular instance $(G, \mathbf{x}) \in \mathcal{F}$, denoted $T_{\mathcal{A}}(G, \mathbf{x})$, is the number of rounds from the beginning of the execution of $\mathcal{A}$ until all nodes terminate. The running time of $\mathcal{A}$ is typically evaluated with respect to a collection of parameters $\Lambda = \{\mathsf{q}_1, \mathsf{q}_2, \cdots, \mathsf{q}_\ell\}$. Specifically, it is compared to a function $f : \mathbf{N}^\ell \to \mathbf{R}^+$; we say that f is an upper bound for the running time of $\mathcal{A}$ with respect to Λ if $T_{\mathcal{A}}(G, \mathbf{x}) \leqslant f(\mathsf{q}_1^*, \mathsf{q}_2^*, \cdots, \mathsf{q}_\ell^*)$ for every instance $(G, \mathbf{x}) \in F$ with parameters $\mathsf{q}_i^* := \mathsf{q}_i(G, \mathbf{x})$ for $i \in [1, \ell]$.

For an integer i, the algorithm $\mathcal{A}$ *restricted to i rounds* is the local algorithm $\mathcal{B}$ that consists in running $\mathcal{A}$ for precisely i rounds. The output $\mathbf{y}(u)$ of $\mathcal{B}$ at a vertex u is defined as follows: if, during the i rounds, $\mathcal{A}$ outputted a value y at u then $\mathbf{y}(u) := y$; otherwise we let $\mathbf{y}(u)$ be an arbitrary value, e.g., “0”.

A *randomized* local algorithm is a local algorithm that allows each node to use random bits in its local computation—the random bits used by different nodes being independent. A randomized algorithm $\mathcal{A}$ is *Las Vegas* if its correctness is guaranteed with probability 1. The *running time* of a Las Vegas algorithm $\mathcal{A}_{LV}$ over a particular configuration $(G, \mathbf{x}) \in \mathcal{F}$, denoted $T_{\mathcal{A}_{LV}}(G, \mathbf{x})$, is a random variable, which may not be bounded. However, the expected value of $T_{\mathcal{A}_{LV}}(G, \mathbf{x})$ is bounded. A *Monte-Carlo* algorithm $\mathcal{A}_{MC}$ with guarantee $\rho \in (0, 1]$ is a randomized algorithm that takes a configuration $(G, \mathbf{x}) \in \mathcal{F}$ as input and terminates before a predetermined time $T_{\mathcal{A}_{MC}}(G, \mathbf{x})$ (called the *running time* of $\mathcal{A}_{MC}$). It is guaranteed that the output vector produced by Algorithm $\mathcal{A}_{MC}$ is a solution to Prob with probability at least ρ. Finally, a *weak Monte-Carlo* algorithm $\mathcal{A}_{WMC}$ with guarantee $\rho \in (0, 1]$ guarantees that with probability at least ρ, the algorithm outputs a correct solution by its running time $T_{\mathcal{A}_{WMC}}(G, \mathbf{x})$. (Observe that it is not guaranteed that any execution of the weak Monte-Carlo algorithm will terminate by the prescribed time $T_{\mathcal{A}_{WMC}}(G, \mathbf{x})$, or even terminate at all.) Note that a Monte-Carlo algorithm is in particular a weak Monte-Carlo algorithm, with the same running time and guarantee. Moreover, for any constant $c \in (0,1]$, a Las Vegas algorithm running in expected time T is a weak Monte-Carlo algorithm running in time $O(T)$ with guarantee c.

A remark about running an algorithm after another: Many $\mathcal{LOCAL}$ algorithms happen to have different termination times at different nodes. On the other hand, most of the algorithms rely on a simultaneous wake up of all nodes. This becomes a problem when one wants to run an algorithm $\mathcal{A}_1$ and subsequently an algorithm $\mathcal{A}_2$ taking the output of $\mathcal{A}_1$ as input. Indeed, this problem amounts to running $\mathcal{A}_2$ with non-simultaneous wake up: a node u starts $\mathcal{A}_2$ when it terminates $\mathcal{A}_1$.

As observed (e.g., by Kuhn [22]), one can use a synchronizer [2] to run a synchronous local algorithm in an asynchronous system, with the same asymptotic time complexity. Hence, the synchronicity assumption can actually be removed. Although the standard asynchronous model introduced still assumes simultaneous wake up, it can be easily verified that the technique still applies with non-simultaneous wake up times if a node can buffer messages received before it wakes up, which is the case when running an algorithm after another. However, we have to adapt the notion of running time. We define it as the number of time units elapsed between the last wake up time of a node and the last termination time of a node. We let $\mathcal{A}_1;\mathcal{A}_2$ be the process of running $\mathcal{A}_2$ after $\mathcal{A}_1$. Note that the running time of $\mathcal{A}_1;\mathcal{A}_2$ is bounded by the sum of the running times of $\mathcal{A}_1$ and $\mathcal{A}_2$. In the sequel we implicitly assume that the simple α synchronizer is used when running a sequence $\mathcal{A}_1;\mathcal{A}_2;\cdots;\mathcal{A}_k$ of algorithms.

Local algorithms requiring parameters: Fix a problem Prob and let $\mathcal{F}$ be a collection of instances for Prob.
Let $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_r\}$ be a collection of parameters and let $\mathcal{A}$ be a local algorithm. We say that $\mathcal{A}$ *requires* Γ if, in order to execute

$\mathcal{A}$ on an instance $(G, \mathbf{x}) \in \mathcal{F}$, all nodes must agree on a value $\tilde{\mathtt{p}}$ for each parameter $\mathtt{p} \in \Gamma$. The value $\tilde{\mathtt{p}}$ is a *guess* for $\mathtt{p}$. A collection of guesses for the parameters in Γ is denoted by $\tilde{\Gamma}$ and an algorithm $\mathcal{A}$ that requires Γ is denoted by $\mathcal{A}^{\Gamma}$. An algorithm that does not require any parameter is *uniform*.

Consider an instance $(G, \mathbf{x}) \in \mathcal{F}$, a collection Γ of parameters and a parameter $\mathtt{p} \in \Gamma$. A guess $\tilde{\mathtt{p}}$ for $\mathtt{p}$ is termed *good* if $\tilde{\mathtt{p}} \geqslant \mathtt{p}(G, \mathbf{x})$, and the guess $\tilde{\mathtt{p}}$ is called *correct* if $\tilde{\mathtt{p}} = \mathtt{p}(G, \mathbf{x})$. We typically write correct guesses and collection of correct guesses with a star exponent, that is $\mathtt{p}^*$ and $\Gamma^*(G, \mathbf{x})$, respectively. When $(G, \mathbf{x})$ is clear from the context, we may use the notation Γ^* instead of $\Gamma^*(G, \mathbf{x})$.

An algorithm $\mathcal{A}^{\Gamma}$ *depends* on Γ if for every instance $(G, \mathbf{x}) \in \mathcal{F}$, the correctness of $\mathcal{A}^{\Gamma}$ over $(G, \mathbf{x})$ is guaranteed when $\mathcal{A}^{\Gamma}$ uses a collection $\tilde{\Gamma}$ of good guesses.

Consider an algorithm $\mathcal{A}^{\Gamma}$ that depends on a collection of parameters $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_r\}$ and fix an instance $(G, \mathbf{x})$. Observe that the running time of $\mathcal{A}^{\Gamma}$ over $(G, \mathbf{x})$ may be different for different collections of guesses $\tilde{\Gamma}$, in other words, the running time over $(G, \mathbf{x})$ is a function of $\tilde{\Gamma}$. Recall that when we consider an algorithm that does not require parameters, we still typically evaluate its running time with respect to a collection of parameters Λ. We generalize this to the case where the algorithm depends on Γ as follows.

Consider two collections of parameters $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_r\}$ and $\Lambda = \{\mathtt{q}_1, \mathtt{q}_2, \cdots, \mathtt{q}_\ell\}$. Some parameters may belong to both Γ and Λ. Without loss of generality, we shall always assume that $\{\mathtt{p}_{r'+1}, \mathtt{p}_{r'+2}, \cdots, \mathtt{p}_r\} \cap \{\mathtt{q}_{r'+1}, \mathtt{q}_{r'+2}, \cdots, \mathtt{q}_\ell\} = \emptyset$ for some $r' \in [0, \min\{r, \ell\}]$ and $\mathtt{p}_i = \mathtt{q}_i$ for every $i \in [1, r']$. Notice that $\Gamma \setminus (\Gamma \cap \Lambda) = \{\mathtt{p}_{r'+1}, \mathtt{p}_{r'+2}, \cdots, \mathtt{p}_r\}$. A function $f : (\mathbf{R}^+)^\ell \to \mathbf{R}^+$ *upper bounds* the running time of $\mathcal{A}^{\Gamma}$ with respect to Γ and Λ if the running time $T_{\mathcal{A}^{\Gamma}}(G, \mathbf{x})$ of $\mathcal{A}^{\Gamma}$ for $(G, \mathbf{x}) \in \mathcal{F}$ using a collection of good guesses $\tilde{\Gamma} = \{\tilde{\mathtt{p}}_1, \tilde{\mathtt{p}}_2, \cdots, \tilde{\mathtt{p}}_r\}$ is at most $f(\tilde{\mathtt{p}}_1, \tilde{\mathtt{p}}_2, \cdots, \tilde{\mathtt{p}}_{r'}, \mathtt{q}^*_{r'+1}, \cdots, \mathtt{q}^*_\ell)$, where $\mathtt{q}^*_i = \mathtt{q}_i(G, \mathbf{x})$ for $i \in [r', \ell]$. Note that we do not put any restriction on the running time of $\mathcal{A}^{\Gamma}$ over $(G, \mathbf{x})$ if some of the guesses in $\tilde{\Gamma}$ are not good. In fact, in such a case, the algorithm may not even terminate.

For simplicity of notation, when Γ and Λ are clear from the context, we say that f upper bounds the running time of $\mathcal{A}^\Gamma$, without writing that it is with respect to Γ and Λ.

The set Γ is *weakly-dominated* by Λ if for each $j \in [r'+1\,, r]$, there exists an index $i_j \in [1\,, \ell]$ and an ascending function g_j such that $g_j(\mathtt{p}_j(G, \mathbf{x})) \leqslant \mathtt{q}_{i_j}(G, \mathbf{x})$ for every instance $(G, \mathbf{x}) \in \mathcal{F}$. (For example, $\Gamma = \{\Delta\}$ is weakly-dominated by $\{\Lambda\} = n$, since $\Delta(G, \mathbf{x}) \leqslant n(G, \mathbf{x})$ for any $(G, \mathbf{x})$.)

3 Pruning Algorithms

3.1 Overview

Consider a problem Prob in the centralized setting and an efficient randomized Monte-Carlo algorithm $\mathcal{A}$ for Prob. A known method for transforming $\mathcal{A}$ into a Las Vegas algorithm is based on repeatedly doing the following. Execute $\mathcal{A}$ and, subsequently, execute an algorithm that checks the validity of the output. If the checking fails then continue, and otherwise, terminate, i.e., break the loop. This transformation can yield a Las Vegas algorithm whose expected running time is similar to the running time of the Monte-Carlo algorithm provided that the checking mechanism used is efficient.

If we wish to come up with a similar transformation in the context of locality, a first idea would be to consider a local algorithm that checks the validity of a tentative output vector. This concept has been studied from various perspectives (cf., e.g., [16, 21, 32]). However, such fast local checking procedures can only guarantee that faults are detected by at least one node, whereas to restart the Monte-Carlo algorithm, all nodes should be aware of a fault. This notification can take diameter time and will thus violate the locality constraint.

Instead of using local checking procedures, we introduce the notion of *pruning algorithms*. Informally, this is a mechanism that identifies "valid areas" where the tentative output vector $\hat{\mathbf{y}}$ is valid and *prunes* these areas, i.e., takes them out of further consideration. A pruning algorithm $\mathcal{P}$ must satisfy two properties, specifically, (1) *gluing*: $\mathcal{P}$ must make sure that the current solution on these "pruned areas" can be extended to a valid solution for the remainder of the graph, and (2) *solution detection*: if $\hat{\mathbf{y}}$ was a valid global solution to begin with then $\mathcal{P}$ should prune all nodes. Observe that since the empty output vector is a solution for the empty input graph then (1) implies the converse of (2), that is, if $\mathcal{P}$ prunes all nodes, then $\hat{\mathbf{y}}$ was a valid global solution.

Now, given a Monte-Carlo algorithm $\mathcal{A}$ and a pruning algorithm $\mathcal{P}$ for the problem, we can transform $\mathcal{A}$ into a Las Vegas algorithm by executing the pair of algorithms $(\mathcal{A};\mathcal{P})$ in iterations, where each iteration i is executed on the graph G_i induced by the set of nodes that were not pruned in previous iterations (G_1 is the initial graph G). If, in some iteration i, Algorithm $\mathcal{A}$ solves the problem on the graph G_i, then the solution detection property guarantees that the subsequent pruning algorithm will prune all nodes in G_i and hence at that time all nodes are pruned and the execution terminates. Furthermore, using induction, it can be shown that the gluing property guarantees that the correct solution to G_i combined with the output of the previously pruned nodes forms a solution for G.

3.2 Pruning algorithms: definition and examples.

We now formally define pruning algorithms. Fix a problem Prob and a family of instances $\mathcal{F}$ for Prob. A *pruning* algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$ is a uniform algorithm that takes as input a triplet $(G, \mathbf{x}, \hat{\mathbf{y}})$, where $(G, \mathbf{x}) \in \mathcal{F}$ and $\hat{\mathbf{y}}$ is some tentative output vector, and returns a bit $b(v)$ and an updated input $\mathbf{x}'(v)$ at each node v. The bit $b(v)$ indicates whether v belongs to some selected subset $W \subseteq V(G)$ of nodes to be pruned. (Recall that the idea is to assume that nodes in W have a satisfying tentative output value and that they can be excluded from further computations.) Let G' be the subgraph of G induced by the nodes in $V(G) \setminus W$. The pruning algorithm does not change the input of the nodes in W, i.e., $\mathbf{x}'(v) = \mathbf{x}(v)$ whenever $b(v) = 1$ (that is, $v \in W$). On the other hand, the pruning algorithm may change the input for the rest of the nodes, i.e., those in G'. (Informally, this is because after $\mathcal{P}$ pruned the set W, it may need to adjust the remaining nodes for further use.) Thus, Algorithm $\mathcal{P}$ takes a triplet $(G, \mathbf{x}, \hat{\mathbf{y}})$ as input, where $(G, \mathbf{x}) \in \mathcal{F}$, and returns a pair $(G', \mathbf{x}')$. The pruning algorithm must guarantee that $(G', \mathbf{x}') \in \mathcal{F}$.

Consider now an output vector $\mathbf{y}'$ for the nodes in $V(G')$. The *combined* output vector $\mathbf{y}$ of the vectors $\hat{\mathbf{y}}$ and $\mathbf{y}'$ is the output vector that is a combination of $\hat{\mathbf{y}}$ restricted to the nodes in W and $\mathbf{y}'$ restricted to the nodes in G', i.e., $\mathbf{y}(v) := \hat{\mathbf{y}}(v)$ if $v \in W$ and $\mathbf{y}(v) := \mathbf{y}'(v)$ otherwise. A pruning algorithm $\mathcal{P}$ for a problem Prob must satisfy the following properties.

- **Solution detection:** $(G, \mathbf{x}, \hat{\mathbf{y}}) \in \text{Prob} \implies \mathcal{P}$ outputs $W = V(G)$.
- **Gluing:** if $\mathcal{P}(G, \mathbf{x}, \hat{\mathbf{y}}) = (G', \mathbf{x}')$ and $\mathbf{y}'$ is a solution for $(G', \mathbf{x}')$, i.e., $(G', \mathbf{x}', \mathbf{y}') \in \text{Prob}$, then the combined output vector $\mathbf{y}$ is a solution for $(G, \mathbf{x})$, i.e, $(G, \mathbf{x}, \mathbf{y}) \in \text{Prob}$.

As mentioned earlier, it follows from the gluing property that if the pruning algorithm $\mathcal{P}$ outputs $W = V(G)$ (i.e., all nodes are pruned) then $(G, \mathbf{x}, \hat{\mathbf{y}}) \in \text{Prob}$.

The pruning algorithm $\mathcal{P}$ is *monotone with respect to a parameter* $\mathtt{p}$ if $\mathtt{p}(G, \mathbf{x}) \geqslant \mathtt{p}(G', \mathbf{x}')$, for every $(G, \mathbf{x}) \in \mathcal{F}$, where $(G', \mathbf{x}') = \mathcal{P}(G, \mathbf{x}, \hat{\mathbf{y}})$, for some $\hat{\mathbf{y}}$. The pruning algorithm $\mathcal{P}$ is *monotone with respect to a collection of parameters* Γ if $\mathcal{P}$ is monotone with respect to every parameter $\mathtt{p} \in \Gamma$. In such a case, we may also say that $\mathcal{P}$ is Γ-*monotone.* The following assertions follow from the definition of a parameter.

Observation 3.1 *Let $\mathcal{P}$ be a pruning algorithm. Then (1) Algorithm $\mathcal{P}$ is monotone with respect to any graph-parameter, and (2) If $\mathcal{P}$ does not update the inputs of the unpruned nodes, i.e., $\mathbf{x}'(v) = \mathbf{x}(v)$ for every $v \in V \setminus W$, then $\mathcal{P}$ is monotone with respect to any parameter.*

For simplicity, we restrict the running time of a pruning algorithm $\mathcal{P}$ to be constant. We stress however, that our results can be extended to general pruning algorithms while paying an extra additive cost to the running time that corresponds to the running time of the given pruning algorithm; we shall elaborate more on this later.

We now give examples of (constant time) pruning algorithms for several problems, namely, $(2, \beta)$-Ruling set for a constant integer β (recall that MIS is precisely $(2, 1)$-Ruling set), and maximal matching. These pruning algorithms do not change the input at nodes outside W (in fact, the input is ignored in these problems, and can be assumed to be empty). Thus, by Observation 3.1, they are monotone with respect to any parameter.

The $(2, \beta)$-ruling set pruning algorithm: Let β be a (constant) integer. We define a pruning algorithm $\mathcal{P}_{(2,\beta)}$ for the $(2, \beta)$-ruling set problem as follows. Given a triplet $(G, \mathbf{x}, \hat{\mathbf{y}})$, let W be the set of nodes u satisfying one of the following two conditions.

- $\hat{\mathbf{y}}(u) = 1$ and $\hat{\mathbf{y}}(v) = 0$ for all $v \in N(u)$, or
- $\hat{\mathbf{y}}(u) = 0$ and $\exists v \in B_G(u, \beta)$ such that $\hat{\mathbf{y}}(v) = 1$ and $\hat{\mathbf{y}}(w) = 0$ for all $w \in N(v)$.

The question of whether a node u belongs to W can be determined by inspecting $B_G(u, 1 + \beta)$, the ball of radius $1 + \beta$ around u. Hence, we obtain the following.

Observation 3.2 *Algorithm $\mathcal{P}_{(2,\beta)}$ is a pruning algorithm for the $(2, \beta)$-ruling set problem, running in time $1+\beta$. (In particular, $\mathcal{P}_{(2,1)}$ is a pruning algorithm for the MIS problem running in time 2.) Furthermore, $\mathcal{P}_{(2,\beta)}$ is monotone with respect to any parameter.*

The interested reader can check that this pruning algorithm is not implementable through simple combinations of the classical local check procedure for $(2, \beta)$-ruling set even though it has a similar flavor.

The maximal matching problem: We define a pruning algorithm $\mathcal{P}_{\mathtt{MM}}$ as follows. Given a tentative output vector $\hat{\mathbf{y}}$, recall that u and v are matched when u and v are neighbors, $\hat{\mathbf{y}}(u) = \hat{\mathbf{y}}(v)$ and $\hat{\mathbf{y}}(w) \neq \hat{\mathbf{y}}(u)$ for every $w \in (N_G(u) \cup N_G(v)) \setminus \{u, v\}$. Set W to be the set of nodes u satisfying one of the following conditions.

- $\exists v \in N(u)$ such that u and v are matched, or
- $\forall v \in N(u)$, $\exists w \neq u$ such that v and w are matched.

Observation 3.3 *Algorithm $\mathcal{P}_{\mathtt{MM}}$ is a pruning algorithm for* `MM` *whose running time is 3. Furthermore, $\mathcal{P}_{\mathtt{MM}}$ is monotone with respect to any parameter.*

We exhibit several applications of pruning algorithms. The main application appears in the next section, where we show how pruning algorithms can be used to transform non-uniform algorithms into uniform ones. Before we continue, we need the following concept.

3.3 Alternating Algorithms

A pruning algorithm can be used in conjunction with a sequence of algorithms as follows. Let $\mathcal{F}$ be a collection of instances for some problem Prob. For each $i \in \mathbf{N}$, let $\mathcal{A}_i$ be an algorithm defined on $\mathcal{F}$. Algorithm $\mathcal{A}_i$ does not necessarily solve Prob, it is only assumed to produce some output.

Let $\mathcal{P}$ be a pruning algorithm for Prob and $\mathcal{F}$, and for $i \in \mathbf{N}$, let $\mathcal{B}_i := (\mathcal{A}_i; \mathcal{P})$, that is, given an instance $(G, \mathbf{x})$, Algorithm $\mathcal{B}_i$ first executes $\mathcal{A}_i$, which returns an output vector $\mathbf{y}$ for the nodes of G and, subsequently, Algorithm $\mathcal{P}$ is executed over the triplet $(G, \mathbf{x}, \mathbf{y})$. We define the *alternating* algorithm π for $(\mathcal{A}_i)_{i \in \mathbf{N}}$ and $\mathcal{P}$ as follows. The alternating algorithm $\pi = \pi((\mathcal{A}_i)_{i \in \mathbf{N}}, \mathcal{P})$ executes the algorithms $\mathcal{B}_i$ for $i = 1, 2, 3, \cdots$ one after the other: let $(G_1, \mathbf{x}_1) = (G, \mathbf{x})$ be the initial instance given to π; for $i \in \mathbf{N}$, Algorithm $\mathcal{A}_i$ is executed on the instance $(G_i, \mathbf{x}_i)$ and produces the output vector $\mathbf{y}_i$. The subsequent pruning algorithm $\mathcal{P}$ takes the triplet $(G_i, \mathbf{x}_i, \mathbf{y}_i)$ as input and produces the instance $(G_{i+1}, \mathbf{x}_{i+1})$. See Figure 1 for a schematic view of an alternating algorithm. The definition extends to a finite sequence $(\mathcal{A})_{i=1}^k$ of algorithms in a natural way; the alternating algorithm for $(\mathcal{A})_{i=1}^k$ and $\mathcal{P}$ being $A_1; \mathcal{P}; A_2; \mathcal{P}; \cdots; A_k; \mathcal{P}$.

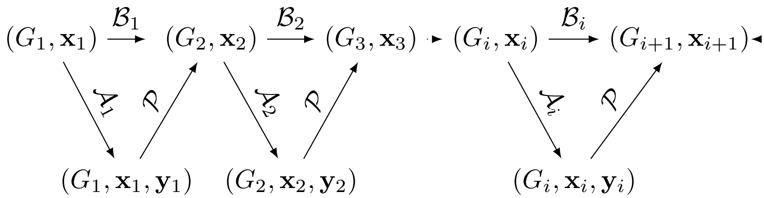

Figure 1: Schematic view of an alternating algorithm for $(\mathcal{A}_i)_{i \in \mathbf{N}}$ and $\mathcal{P}$.

The alternating algorithm π *terminates* on an instance $(G, \mathbf{x}) \in \mathcal{F}$ if there exists k such that $V(G_k) = \emptyset$. Observe that in such a case, the tail $\mathcal{B}_k; \mathcal{B}_{k+1}; \cdots$ of π is trivial. The output vector $\mathbf{y}$ of a terminating alternating algorithm π is defined as the combination of the output vectors $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \cdots$. Specifically, for $s \in [1, k-1]$, let $W_s \coloneqq V(G_s) \setminus V(G_{s+1})$. (Observe that W_s is precisely the set of nodes pruned by the execution of the pruning algorithm $\mathcal{P}$ in $\mathcal{B}_s$.) Then, the collection $\{W_s \mid 1 \leqslant s \leqslant k-1\}$ forms a partition of $V(G)$, i.e., $W_s \cap W_{s'} = \emptyset$ if $s \neq s'$, and $\cup_{s=1}^{k-1} W_s = V(G)$. Observe that the final output $\mathbf{y}$ of π is defined by $\mathbf{y}(u) = \mathbf{y}_s(u)$ for every $u \in W_s$ and every $s \in [1, k-1]$. That is, the output of π restricted to the nodes in W_s is precisely the corresponding output of Algorithm $\mathcal{A}_s$.

Lemma 1. *Consider a problem Prob, a collection of instances $\mathcal{F}$, a sequence of algorithms $(\mathcal{A}_i)_{i \in \mathbf{N}}$ defined on $\mathcal{F}$, and a pruning algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$. Consider the alternating algorithm $\pi =$*

$\pi((\mathcal{A}_i)_{i\in\mathbf{N}},\mathcal{P})$ for $(\mathcal{A}_i)_{i\in\mathbf{N}}$ and $\mathcal{P}$. If π terminates on an instance $(G,\mathbf{x})\in\mathcal{F}$ then it produces a correct output $\mathbf{y}$, that is, $(G,\mathbf{x},\mathbf{y})\in$ Prob.

Proof. Let $(G,\mathbf{x})\in\mathcal{F}$ be a configuration, and consider the execution of Algorithm π over $(G,\mathbf{x})$. Let k be the smallest integer such that $V(G_k)=\emptyset$, and consider the partition $(W_s)_{1\leqslant s\leqslant k-1}$ of $V(G)$ defined above. Recall that the final output vector $\mathbf{y}$ is given by $\mathbf{y}(u)=\mathbf{y}_s(u)$ for every $u\in W_s$ and every $s\in[1\,,k-1]$.

For $a\in[1\,,k-1]$, let $\mathbf{z}_a$ be the restriction of $\mathbf{y}$ to the nodes in $V(G_a)$, i.e., $\mathbf{z}_a(u)\coloneqq\mathbf{y}_s(u)$ for every $u\in W_s$ with $s\in[a\,,k-1]$. Note that $\mathbf{z}_1=\mathbf{y}$ and $\mathbf{z}_{k-1}=\mathbf{y}_{k-1}$. We prove by a descending induction on $a\in[1\,,k-1]$ that $(G_a,\mathbf{x}_a,\mathbf{z}_a)\in$ Prob. For the basis of the induction, i.e., the case $a=k-1$, observe that Algorithm $\mathcal{P}$ applied to $(G_{k-1},\mathbf{x}_{k-1},\mathbf{y}_{k-1})$ outputs $W_{k-1}=V(G_{k-1})$ by the definition of k. Consequently, the gluing property of $\mathcal{P}$ implies that $(G_{k-1},\mathbf{x}_{k-1},\mathbf{z}_{k-1})=(G_{k-1},\mathbf{x}_{k-1},\mathbf{y}_{k-1})\in$ Prob. Now, assume that $(G_{a+1},\mathbf{x}_{a+1},\mathbf{z}_{a+1})\in$ Prob for some integer $a\in[1\,,k-2]$. Consider the operation of $\mathcal{B}_a$. The pruning algorithm of that algorithm operates on $(G_a,\mathbf{x}_a,\mathbf{y}_a)$ and outputs the configuration $(G_{a+1},\mathbf{x}_{a+1})$. Notice that $\mathbf{z}_a$ is the combined output vector for this operation of the pruning algorithm. Therefore, the gluing property implies that $(G_a,\mathbf{x}_a,\mathbf{z}_a)\in$ Prob. This concludes the induction proof. The desired fact that $(G,\mathbf{x},\mathbf{y})\in$ Prob now follows from the fact that $(G,\mathbf{x},\mathbf{y})=(G_1,\mathbf{x}_1,\mathbf{z}_1)$. ∎

In what follows, we often produce a sequence of algorithms $(\mathcal{A}_i)_{i\in\mathbf{N}}$ from an algorithm $\mathcal{A}^\Gamma$ requiring a collection Γ of parameters. The general idea is to design a sequence of guesses $\tilde{\Gamma}_i$ and let $\mathcal{A}_i$ be algorithm $\mathcal{A}^\Gamma$ provided with guesses $\tilde{\Gamma}_i$. Given a pruning algorithm $\mathcal{P}$, we then obtain a uniform alternating algorithm $\pi=\pi((\mathcal{A}_i)_{i\in\mathbf{N}},\mathcal{P})$. The sequence of guesses is designed such that for any configuration $(G,\mathbf{x})\in\mathcal{F}$, there exists some i for which $\tilde{\Gamma}_i$ is a collection of good guesses for $(G,\mathbf{x})$. The crux is to obtain an execution time for $\mathcal{A}_1;\mathcal{P};\mathcal{A}_2;\mathcal{P};\cdots;\mathcal{A}_i;\mathcal{P}$ of the same order as the execution time of $\mathcal{A}^\Gamma$ provided with the collection $\Gamma^*(G,\mathbf{x})$ of correct guesses.

4 A General Method

We now turn to the main application of pruning algorithms discussed in this paper, that is, the construction of a transformer taking a non-uniform algorithm A^{Γ} as a black box and producing a uniform one that enjoys the same (asymptotic) time complexity as the original non-uniform algorithm. In Subsection 4.1 we consider the deterministic setting. We begin with a few illustrative examples and conclude the subsection with a somewhat restrictive, yet useful, transformer (Theorem 8). This transformer considers a single set of parameters $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_\ell\}$, and assumes that (1) the given non-uniform algorithm $\mathcal{A}^{\Gamma}$ depends on Γ and (2) the running time of $\mathcal{A}^{\Gamma}$ is evaluated with respect to the parameters in Γ. Such a situation is customary, and occurs for instance for the best currently known MIS Algorithms [4, 22, 34] as well as for the maximal matching algorithm of Hanckowiak *et al.* [19]. As a result, the transformer given by Theorem 8 can be used to transform each of these algorithms into a uniform one with asymptotically the same time complexity.

This transformer is then extended in Subsection 4.2 to the randomized setting (Theorem 9). In Subsection 4.3, we establish Theorem 10, which generalizes both Theorem 8 and Theorem 9. Finally, we conclude the section with Theorem 11 in Subsection 4.4, which shows how to manipulate several uniform algorithms that run in unknown times to obtain a uniform algorithm that runs as fast as the fastest algorithm among those given algorithms.

4.1 The Deterministic Case

The basic idea is very simple. Consider a problem for which we have a pruning algorithm $\mathcal{P}$, and a non uniform algorithm $\mathcal{A}$ that requires the knowledge of upper bounds on some parameters. To obtain a uniform algorithm, we execute the pair of algorithms $(\mathcal{A};\mathcal{P})$ in iterations, where each iteration executes $\mathcal{A}$ using a specific set of guesses for the parameters. Typically, as iterations proceed, the guesses for the parameters grow larger and larger until we reach an iteration i where all the guesses are larger than the actual value of the corresponding parameters. In this iteration, the operation of $\mathcal{A}$ on G_i using such guesses guarantees a correct solution on G_i. The solution detection property of the pruning algorithm guarantees that the execution terminates in this iteration and hence, Lemma 1 guarantees that the output of all nodes combines to a global solution on G. To bound the running time, we shall make sure that the total running time is dominated by the running time of the last iteration, and that this last iteration is relatively fast.

There are various delicate points when using this general strategy. For example, in iterations where incorrect guesses are used, we have no control over the behavior of the non-uniform algorithm $\mathcal{A}$ and, in particular, it may run for too many rounds, perhaps even indefinitely. To overcome this obstacle, we allocate a prescribed number of rounds for each iteration; if Algorithm $\mathcal{A}$ reaches this time bound without outputting at some node u, then we force it to terminate with an arbitrary output. Subsequently, we run the pruning algorithm and proceed to the next iteration.

Obviously, this simple approach of running in iterations and increasing the guesses from iteration to iteration is hardly new. It was used, for example, in the context of wireless networks to compute estimates of parameters (cf., e.g., [8, 31]), or to estimate the number of faults [25]. One of the main contributions of the current paper is the formalization and generalization of this technique, allowing it to be used for a wide varieties of problems and applications. Interestingly, note that we are only concerned with getting rid of the knowledge of some parameters, and not with obtaining estimates for them (in particular, when our algorithms terminate, the vertices have no way of knowing whether they have upper bounds on these parameters).

To illustrate the method, let us consider the non-uniform MIS algorithm of Panconesi and Srinivasan [34]. This algorithm $\mathcal{A}$ assumes the knowledge of an upper bound $\tilde{n}$ on the number of nodes n, and runs in time at most $f(\tilde{n}) = 2^{O(\sqrt{\log \tilde{n}})}$. Consider a pruning algorithm $\mathcal{P}_{\texttt{MIS}}$ for MIS (such an algorithm is given by Observation

3.2). The following sketches our technique for obtaining a uniform MIS algorithm. For each integer i, set $n_i \coloneqq \max\left\{a \in \mathbf{N} \,\middle|\, f(a) \leqslant 2^i\right\}$. In Iteration i, for $i = 1, 2, \cdots$, we first execute Algorithm $\mathcal{A}$ using the guess n_i (as an input serving as an upper bound for the number of nodes) for precisely 2^i rounds. Subsequently, we run the pruning algorithm $\mathcal{P}_{\texttt{MIS}}$. When the pruning algorithm terminates, we execute the next iteration on the non-pruned nodes. Let s be the integer such that $2^{s-1} < f(n) \leqslant 2^s$, where n is the number of nodes of the input graph. By the definition, $n \leqslant n_s$. Therefore, the application of $\mathcal{A}$ in Iteration s uses a guess n_s that is indeed good, i.e., larger than the number of nodes. Moreover, this execution of A is completed before the prescribed deadline of 2^s rounds expires because its running time is at most $f(n_s) \leqslant 2^s$. Hence, we are guaranteed to have a correct solution by the end of Iteration s. The running time is thus at most $\sum_{i=1}^{s} 2^i = O(f(n))$.

This method can sometimes be extended to simultaneously remove prior knowledge concerning several parameters. For example, consider the MIS algorithm of Barenboim and Elkin [4] (or that of Kuhn [22]), which requires the knowledge of upper bounds $\tilde{n}$ and $\tilde{\Delta}$ on n and Δ, respectively, and runs in time $f(\tilde{n}, \tilde{\Delta}) = f_1(\tilde{n}) + f_2(\tilde{\Delta})$, where $f_1(\tilde{\Delta}) = O(\tilde{\Delta})$ and $f_2(\tilde{n}) = O(\log^* \tilde{n})$. The following sketches our method for obtaining a corresponding uniform MIS algorithm that runs in time $O(f(n, \Delta))$. For each integer i, set $n_i \coloneqq \max\left\{a \in \mathbf{N} \,\middle|\, f_1(a) \leqslant 2^i\right\}$ and $\Delta_i \coloneqq \max\left\{a \in \mathbf{N} \,\middle|\, f_2(a) \leqslant 2^i\right\}$. In Iteration i, for $i = 1, 2, \cdots$, we first execute Algorithm $\mathcal{A}$ using the guesses n_i and Δ_i, but this time the execution lasts for precisely $2 \cdot 2^i$ rounds. (The factor 2 in the running time of an iteration follows from the fact that we consider two parameters here, namely n and Δ.) Subsequently, we run the pruning algorithm $\mathcal{P}_{\texttt{MIS}}$, and as before, when the pruning algorithm terminates, we execute the next iteration on the non-pruned nodes. Now, let s be the integer such that $2^{s-1} < f(n, \Delta) \leqslant 2^s$. By the definition, $n \leqslant n_s$ and $\Delta \leqslant \Delta_s$. Hence, the application of $\mathcal{A}$ in Iteration s uses guesses that are indeed good. This execution of A is completed before the prescribed deadline of 2^{s+1} rounds expires because its running time is at most

$f_1(n_s) + f_2(\Delta_s) \leqslant 2^{s+1}$. Thus, the algorithm consists of at most s iterations. Since the running time of the whole execution is dominated by the running time of the last iteration, the total running time is $O(2^{s+1}) = O(f(n, \Delta))$.

The following theorem formalizes the above discussion. It considers a single set of parameters $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_\ell\}$, and assumes that (1) the given non-uniform algorithm $\mathcal{A}^\Gamma$ depends on Γ and (2) the running time of $\mathcal{A}^\Gamma$ is evaluated according to the parameters in Γ. Recall that in such a case, we say that a function $f : \mathbf{N}^\ell \to \mathbf{R}^+$ *upper bounds* the running time of $\mathcal{A}^\Gamma$ with respect to Γ if the running time $T_{\mathcal{A}^\Gamma}(G, \mathbf{x})$ of $\mathcal{A}^\Gamma$ for every $(G, \mathbf{x}) \in \mathcal{F}$ using a collection of good guesses $\tilde{\Gamma} = \{\tilde{\mathtt{p}}_1, \tilde{\mathtt{p}}_2, \cdots, \tilde{\mathtt{p}}_\ell\}$ for $(G, \mathbf{x})$ is at most $f(\tilde{\mathtt{p}}_1, \tilde{\mathtt{p}}_2, \cdots, \tilde{\mathtt{p}}_\ell)$.

Theorem 8. *Consider a problem Prob and a family of instances $\mathcal{F}$. Let $\mathcal{A}^\Gamma$ be a deterministic algorithm for Prob and $\mathcal{F}$ depending on $\Gamma := \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_\ell\}$. Suppose that the running time of $\mathcal{A}^\Gamma$ is upper bounded by some function $f : \mathbf{N}^\ell \to \mathbf{R}^+$. Assume there exists a sequence-number function s_f for f, and a Γ-monotone pruning algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$. Then there exists a* uniform *deterministic algorithm for Prob and $\mathcal{F}$ whose running time is $O(f^* \cdot s_f(f^*))$, where $f^* = f(\Gamma^*)$.*

Proof. Let c be the bounding constant as described in the definition of the sequence-number function s_f (see the definition in Section 2). Fix a positive integer i. We describe a uniform algorithm $\mathcal{B}_i$ defined for the configurations in $\mathcal{F}$ and running in time $O(s_f(2^i) \cdot 2^i)$. Let $S_i = S_f(2^i) = \{\underline{x}^1, \underline{x}^2, \cdots, \underline{x}^{J_i}\}$ be a set of size $J_i \leqslant s_f(2^i)$ as described in the definition of the sequence-number function s_f. For every $j \in [1\,, J_i]$, consider the uniform algorithm $\mathcal{A}_{j,i}$ that consists of running $\mathcal{A}^\Gamma$ with the collection of guesses $\underline{x}^j$ of S_i. More precisely, the sequence $\underline{x}^j = \{x^j_1, x^j_2, \cdots, x^j_\ell\} \in S_i$ can be seen as a collection $\tilde{\Gamma}_{j,i} = \{\mathtt{p}_1(j,i), \mathtt{p}_2(j,i), \cdots, \mathtt{p}_\ell(j,i)\}$, where $\mathtt{p}_r(j,i) := x^j_r$, for every $r \in [1\,, \ell]$. Now, we define $\mathcal{A}'_{j,i}$ to be the algorithm $\mathcal{A}_{j,i}$ restricted to $c \cdot 2^i$ rounds where c is the bounding constant in the definition of s_f.

Next, we let $\mathcal{B}_i$ be the uniform alternating algorithm for the sequence of uniform algorithms $\{\mathcal{A}'_{j,i}\}_{j \in [1, J_i]}$ and the pruning algorithm $\mathcal{P}$. That is, $\mathcal{B}_i = \hat{\mathcal{A}}_{1,i}; \hat{\mathcal{A}}_{2,i}; \cdots; \hat{\mathcal{A}}_{J_i,i}$ where $\hat{\mathcal{A}}_{j,i} := \mathcal{A}'_{j,i}; \mathcal{P}$ for each $j \in [1, J_i]$. Finally, the desired uniform algorithm π is simply $\mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3; \cdots$. Thus, π runs in iterations and, for each $i \in \mathbf{N}$, Iteration i of π consists in running $\mathcal{B}_i$ over the configuration $(G_i, \mathbf{x}_i)$. Now, $\mathcal{B}_i$ is an alternating algorithm itself, and thus also runs in iterations, called *sub-iterations*: each such sub-iteration consists in running $\hat{\mathcal{A}}_{j,i}$, that is, an algorithm of the form $\mathcal{A}^\Gamma$ and subsequently the pruning algorithm $\mathcal{P}$. Let $(G_{j,s}, \mathbf{x}_{j,s})$ denote the configuration over which π operates during Sub-iteration j of Iteration s, for $j \in [1, J_s]$. See Figure 2 for a schematic view of an iteration of π.

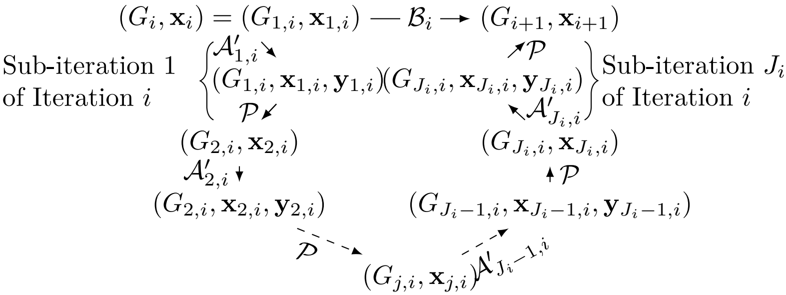

Figure 2: Schematic view of an iteration of π: the iteration i is composed of J_i sub-iterations.

Let us prove that Algorithm π is correct and that its running time over any configuration $(G, \mathbf{x})$ is $O(s_f(f^*) \cdot f^*)$. Fix a configuration $(G, \mathbf{x})$ and consider the operation of Algorithm π on $(G, \mathbf{x})$. Let $\mathtt{p}_r^* = \mathtt{p}_r(G, \mathbf{x})$, for $r \in [1, \ell]$. We know that $f^* = f(\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_\ell^*)$ is an upper bound on the running time of $\mathcal{A}^\Gamma$ over $(G, \mathbf{x})$, assuming $\mathcal{A}^\Gamma$ uses the collection of correct guesses $\Gamma^* := (\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_\ell^*)$. Consider the smallest integer s such that $f^* \leqslant 2^s$. By the definition, there exists $j^* \in [1, J_s]$, such that $\underline{x}^{j^*} = (x_1^{j^*}, x_2^{j^*}, \cdots, x_\ell^{j^*}) \in S_s$ and $\mathtt{p}_r^* \leqslant x_r^{j^*}$, for every $r \in [1, \ell]$. (Recall that $J_s = |S_s| \leqslant s_f(2^s)$.)

Observe that the input of a node stays the same until the node is pruned, since a sub-iteration consists of running an algorithm of the form $\mathcal{A}^\Gamma$ and subsequently the pruning algorithm $\mathcal{P}$. Consequently, for each $i \in \mathbf{N}$, the vectors $\mathbf{x}_{j+1,i}$ and $\mathbf{x}_{j,i}$ coincide on $V(G_{j+1,i})$ for $j \in [1\,, J_i - 1]$, and hence $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$ coincide on $V(G_{i+1})$. Therefore, the monotonicity property of $\mathcal{P}$ implies that $\mathtt{p}_r(G_{j-1,i}, \mathbf{x}_{j-1,i}) \geqslant \mathtt{p}_r(G_{j,i}, \mathbf{x}_{j,i})$ for every $r \in [1\,, \ell]$. Thus, we infer by induction on k that $\mathtt{p}_r^* = \mathtt{p}_r(G, \mathbf{x}) \geqslant \mathtt{p}_r(G_{j,i}, \mathbf{x}_{j,i})$ for every $i \in \mathbf{N}$, $j \in [1\,, J_i]$ and $r \in [1\,, \ell]$.

Now, let us consider Iteration s of π (which is composed of executing Algorithm $\mathcal{B}_s$). Assume that some nodes are still active during Iteration s of π, that is, G_s is not empty. Iteration s of π is composed of $J_s \leqslant s_f(2^s)$ sub-iterations. During Sub-iteration j, Algorithm $\hat{\mathcal{A}}_{j,s} = \mathcal{A}'_{j,s}; \mathcal{P}$ is executed over $(G_s^j, \mathbf{x}_s^j)$. We know that $\mathtt{p}_r^* \geqslant \mathtt{p}_r(G_{j,s}, \mathbf{x}_{j,s})$ for every $j \in [1, J_s]$, and every $r \in [1\,, \ell]$. So, in Sub-iteration j^* of Iteration s, we have $x_{j^*,r} \geqslant \mathtt{p}_r^* \geqslant \mathtt{p}_r(G_{j^*,s}, \mathbf{x}_{j^*,s})$ for every $r \in [1\,, \ell]$.

Now, Sub-iteration j^* consists of first running Algorithm $\mathcal{A}'_{j^*,s}$, which amounts to running $\mathcal{A}^\Gamma$ for $c \cdot 2^s$ rounds using the collection of guesses $\underline{x}^{j^*} = (x_1^{j^*}, x_2^{j^*}, \cdots, x_\ell^{j^*})$. By definition of $S_f(2^s)$, it follows that $f(x_1^{j^*}, x_2^{j^*}, \cdots, x_\ell^{j^*}) \leqslant c \cdot 2^s$, hence, this execution of Algorithm $\mathcal{A}^\Gamma$ is actually completed by time $c \cdot 2^s$. Furthermore, since $x_r^{j^*} \geqslant \mathtt{p}_r(G_{j^*,s}, \mathbf{x}_{j^*,s})$ for every $r \in [1\,, \ell]$, the collection of guesses used by Algorithm $\mathcal{A}^\Gamma$ is good, and hence the algorithm outputs a vector $\mathbf{y}_s^{j^*}$ satisfying $(G_{j^*,s}, \mathbf{x}_{j^*,s}, \mathbf{y}_{j^*,s}) \in \text{Prob}$. By the solution detection property, the subsequent pruning algorithm (still in Sub-iteration j^* of Iteration s) selects $W_{j^*,s} = V(G_{j^*,s})$. By Lemma 1, it follows that π is correct.

Finally, we bound the running time of π. Let T_0 be the (constant) running time of $\mathcal{P}$. Observe that the running time of $\mathcal{B}_i$ is at most $J_i(c \cdot 2^i + T_0) = O(s_f(2^i) \cdot 2^i)$. In other words, Iteration i of π also takes $O(s_f(2^i) \cdot 2^i)$ rounds. Since π consists of at most s iterations, the running time of π is bounded by $\sum_{i=1}^{s} s_f(2^i) \cdot 2^i = O(s_f(2^s) \cdot 2^s)$

because s_f is non-decreasing. Moreover,

$$O(s_f(2^s) \cdot 2^s) = O(s_f(2 \cdot f^*) \cdot 2^s) = O(s_f(f^*) \cdot f^*)$$

since $2^{s-1} < f^*$ and s_f is non-decreasing and moderately-slow. Therefore, the running time of π is bounded by $O(s_f(f^*) \cdot f^*)$. $\square$

Recall from Observation 2.1 that the constant function $s_f = 1$ is a sequence number function for any additive function f. Hence, Theorem 6 follows directly by applying Theorem 8 to the maximal matching algorithm of Hanckowiak *et al.* [19], and using Observation 3.3.

In addition, using Observation 3.2, Theorem 8 allows us to transform each of the MIS algorithms in [4, 22, 34] into a uniform one with asymptotically the same time complexity. That is, we obtain the following.

Corollary 1. *Consider the family $\mathcal{F}$ of general graphs.*

- *There exists a uniform deterministic `MIS` algorithm for $\mathcal{F}$ running in time $O(\Delta + \log^* n)$.*
- *There exists a uniform deterministic `MIS` algorithm for $\mathcal{F}$ running in time $2^{O(\sqrt{\log n})}$.*

4.2 The Randomized Case

We now show how to extend Theorem 8 to the randomized setting. More specifically, we replace the given non-uniform deterministic Algorithm $\mathcal{A}^\Gamma$ in Theorem 8 by a non-uniform weak Monte-Carlo algorithm A^Γ and produce a uniform Las Vegas one running in the same asymptotic running time as A^Γ. This transformer is more sophisticated than the one given in Theorem 8, and requires the use of sub-iterations for bounding the expected running time and probability of success of the resulting Las-Vegas algorithm.

Theorem 9. *Consider a problem Prob and a family of instances $\mathcal{F}$. Let $\mathcal{A}^\Gamma$ be a weak Monte-Carlo algorithm for Prob and $\mathcal{F}$ depending on $\Gamma := \{p_1, p_2, \cdots, p_\ell\}$. Suppose that the running time of $\mathcal{A}^\Gamma$ is upper bounded by some function $f : \mathbf{N}^\ell \to \mathbf{R}^+$. Assume there exists a sequence-number function s_f for f, and a Γ-monotone pruning algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$. Then there exists a* uniform *Las Vegas algorithm for Prob and $\mathcal{F}$ whose expected running time is $O(f^* \cdot s_f(f^*))$, where $f^* = f(\Gamma^*)$.*

Proof. Let T_0 be the (constant) running time of the pruning algorithm $\mathcal{P}$, and let $\mathcal{A}^\Gamma$ be the given weak Monte-Carlo algorithm. To simplify the notations, we assume that the success guarantee ρ of $\mathcal{A}^\Gamma$ is $1/2$.

For each $i \in \mathbf{N}$, let $\mathcal{B}_i$ be the uniform alternating algorithm as defined in the proof of Theorem 8, and let $\mathcal{C}_i := \mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3; \cdots; \mathcal{B}_i$. Finally, let $\pi := \mathcal{C}_1; \mathcal{C}_2; \cdots$.

Let us note some simple facts to ease the analysis of Algorithm $\pi = \mathcal{C}_1; \mathcal{C}_2; \cdots$. First, π can be seen as running in iterations: the iteration i consists in executing $\mathcal{C}_i$. Further, Algorithm $\mathcal{C}_i$ itself also runs in iterations, which are called *sub-iterations of* π: each sub-iteration of π consists in running the alternating algorithm of the form $\mathcal{B}_j$. Recall (from the proof of Theorem 8) that for each positive integer i, the number of rounds used by $\mathcal{B}_i$ is $O(s_f(2^i) \cdot 2^i)$.

Let β_i be the number of rounds used in Iteration i of π (that is, during the execution of algorithm $\mathcal{C}_i$). We have $\beta_i = O(s_f(2^i) \cdot 2^i)$. Consequently, the total number of rounds used during the first i'th iterations of π (that is, running $\mathcal{C}_1; \mathcal{C}_2; \cdots; \mathcal{C}_i$) is

$$\alpha_i := \sum_{k=1}^{i} \beta_k = O(s_f(2^i) \cdot 2^i).$$

For integers $j \leqslant i$, let $(G_{i,j}, \mathbf{x}_{i,j})$ be the configuration given as input to algorithm $\mathcal{B}_j$ in Iteration i. In particular, $(G_i, \mathbf{x}_i) := (G_{i,1}, \mathbf{x}_{i,1})$ is the configuration given as input to algorithm $\mathcal{C}_i$.

First, it follows using similar arguments as the ones given in the proof of Theorem 8, that if π outputs, then the outputted vector $\mathbf{y}$ is a solution, i.e. $(G, \mathbf{x}, \mathbf{y}) \in \text{Prob}$.

It remains to bound the running time of π. We consider the random variable $\mathrm{T}_\pi(G, x)$ that stands for "the running time of π on $(G, \mathbf{x})$". For every integer i, let ρ_i be the probability that $G_i \neq \emptyset$ and $G_{i+1} = \emptyset$, that is, ρ_i is the probability that the last active node becomes inactive precisely during Iteration i of π. In other words,

$$\rho_i \coloneqq \mathbf{Pr}\left(\mathrm{T}_\pi(G, x) \in [\alpha_{i-1} + 1\,, \alpha_i]\right).$$

We know that $f^* = f(\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_\ell^*)$ is an upper bound on the running time of $\mathcal{A}^\Gamma$ over $(G, \mathbf{x})$, assuming $\mathcal{A}^\Gamma$ uses the collection of correct guesses $\Gamma^* \coloneqq (\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_\ell^*)$. Consider the smallest integer s such that $f^* \leqslant 2^s$.

Note that $\alpha_{i+1} \leqslant 2 \cdot \alpha_i$ for every positive integer i. In particular, $\alpha_{s+i} \leqslant 2^i \cdot \alpha_s$, and hence:

$$\begin{aligned}
\mathbf{E}(\mathrm{T}_\pi(G, x)) &\leqslant \alpha_s \cdot \mathbf{Pr}\left(\mathrm{T}_\pi(G, \mathbf{x}) \leqslant \alpha_s\right) + \sum_{i=1}^{\infty} \alpha_{s+i} \cdot \rho_{s+i} \\
&\leqslant \alpha_s + \alpha_s \sum_{i=1}^{\infty} 2^i \cdot \rho_{s+i}.
\end{aligned}$$

Our next goal is to bound ρ_{s+i} from above. For a positive integer r, let χ_r denote the event that $V(G_{r+1}) \neq \emptyset$, that is, none of $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_r$ did output the empty configuration and thus, there is still an active node at the beginning Iteration $r+1$ of π. Thus, $\rho_{s+i} \leqslant \mathbf{Pr}(\chi_{s+i-1})$.

Recall that we assume that the success guarantee of $\mathcal{A}^\Gamma$ is $1/2$. Therefore, using similar analysis as in the proof of Theorem 8, it follows that for every integer $1 \leqslant k$, the probability that an application of $\mathcal{B}_{s+k-1}$ (in particular, during iteration $s+i-1$) does not output the empty configuration is at most $1/2$. As a result, we have

$$\rho_{s+i} \leqslant \mathbf{Pr}(\chi_{s+i-1}) \leqslant \prod_{j=1}^{i} 2^{-j} = 2^{-(i^2+i)/2}.$$

Therefore,

$$\mathbf{E}(\mathrm{T}_\pi(G,x)) \leqslant \alpha_s \left(1 + \sum_{i=1}^{\infty} 2^{i-(i^2+i)/2}\right) = O(\alpha_s) = O(f^* \cdot s_f(f^*)).$$

$\square$

Theorem 7 follows by applying Theorem 9 to the ruling-set algorithm of Schneider and Wattenhofer [36], and using the pruning algorithm given by Observation 3.2.

4.3 The General Theorem

Some complications arise when the correctness of the given non-uniform problem depends on the knowledge of one set of parameters Γ, while its running time is evaluated by another set of parameters Λ. For example, it may be the case that an upper bound on a parameter $\mathtt{p}$ is required for the correct operation of an algorithm, yet the running time of the algorithm does not depend on $\mathtt{p}$. In this case, it may not be clear how to choose the guesses for $\mathtt{p}$. (This occurs, for example, in the MIS algorithms of Barenboim and Elkin [6], where the knowledge of n and the arboricity a are required, yet the running time f is a function of n only.) Such complications can be solved when there is some relation between the parameters in Γ and those in Λ; specifically, when Γ is weakly-dominated by Λ. (Recall the definition of weakly-dominated in Section 2.) This issue is handled in the following theorem, which extends both Theorem 8 and Theorem 9.

Theorem 10. *Consider a problem Prob, a family of instances $\mathcal{F}$ and two sets of parameters Γ and $\Lambda = \{\mathtt{q}_1, \mathtt{q}_2, \cdots, \mathtt{q}_\ell\}$, where Γ is weakly-dominated by Λ. Let $\mathcal{A}^\Gamma$ be a deterministic (respectively, weak Monte-Carlo) algorithm depending on Γ whose running time is upper bounded by some function $f : \mathbf{N}^\ell \to \mathbf{R}^+$. Assume there exists a sequence-number function s_f for f, and a $\Lambda \cup \Gamma$-monotone pruning algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$. Then there exists a* uniform *deterministic (resp., Las Vegas) algorithm for Prob and $\mathcal{F}$ whose running time on every configuration $(G, \mathbf{x}) \in \mathcal{F}$ is is $O(f^* \cdot s_f(f^*))$, where $f^* = f(\Lambda^*(G, \mathbf{x}))$.*

Proof. We first show that we can reduce the problem to the (seemingly) more restrictive case in which we have also $\Gamma \subseteq \Lambda$. Assume that the theorem holds for this restrictive case, and consider the more general case, stated in the theorem. Let $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_r\}$ be weakly-dominated by $\Lambda = \{\mathtt{q}_1, \mathtt{q}_2, \cdots, \mathtt{q}_\ell\}$. Recall that $r' \in [0\,, \min\{r, \ell\}]$ is such that $\{\mathtt{p}_{r'+1}, \mathtt{p}_{r'+2}, \cdots, \mathtt{p}_r\} \cap \{\mathtt{q}_{r'+1}, \mathtt{q}_{r'+2}, \cdots, \mathtt{q}_\ell\} = \emptyset$ and $\mathtt{p}_i = \mathtt{q}_i$ for every $i \in [1\,, r']$. In particular, $\Gamma \setminus (\Gamma \cap \Lambda) = \{\mathtt{p}_{r'+1}, \mathtt{p}_{r'+2}, \cdots, \mathtt{p}_r\}$. Set $t \coloneqq r - r'$. By the definition, the fact that Γ is weakly-dominated by Λ implies that there exists a function $h : [1\,, t] \to [1\,, \ell]$ and, for each $j \in [1\,, t]$, an ascending function g_j such that $g_j(\mathtt{p}_{r'+j}(G, \mathbf{x})) \leqslant \mathtt{q}_{h(j)}(G, \mathbf{x})$ for every configuration $(G, \mathbf{x}) \in \mathcal{F}$.

Let $\Lambda' \coloneqq \Lambda \cup \Gamma = \{\mathtt{q}_1, \mathtt{q}_2, \cdots, \mathtt{q}_\ell, \mathtt{p}_{r'+1}, \mathtt{p}_{r'+2}, \cdots, \mathtt{p}_r\}$, and recall that $f : \mathbf{N}^\ell \to \mathbf{R}^+$ is the function bounding the running time of $\mathcal{A}^\Gamma$. We define a new function $f' : \mathbf{N}^{\ell+t} \to \mathbf{R}$ by

$$f'(x_1, x_2, \cdots, x_\ell, y_1, y_2, \cdots, y_t) \coloneqq f(z_1, z_2, \cdots, z_\ell)$$

where for each $i \in [1\,, \ell]$,

$$z_i \coloneqq \max\left(\{x_i\} \cup \left\{g_k(y_k) \,\middle|\, k \in h^{-1}(\{i\})\right\}\right).$$

Let s_f be a sequence-number function for f and let c be a bounding constant as promised by the definition of s_f. We assert that s_f is also a sequence-number function of f' (with the same bounding constant c). Indeed, let $i \in \mathbf{N}$ and let $S_f(i)$ be a set of sequences promised in the definition of s_f. We construct a set of sequences $S_{f'}(i)$ satisfying $|S_{f'}(i)| = |S_f(i)|$ as follows. For each $(x_1^j, x_2^j, \cdots, x_\ell^j) \in S_f(i)$ we let $S_{f'}(i)$ contain $(x_1^j, x_2^j, \cdots, x_\ell^j, w_1, w_2, \cdots, w_t)$ where $w_k \in g_k^{-1}(\{x_{h(k)}^j\})$ for $k \in [1\,, t]$.

We now check that the set $S_{f'}(i)$ satisfies the two conditions required by the definition of a sequence-number function. Let $\underline{x}' =$

$(x_1, \cdots, x_\ell, y_1, \cdots, y_t) \in \mathbf{N}^{\ell+t}$. First, if $\underline{x}' \in S_{f'}(i)$, then $\underline{x} := (x_1, \cdots, x_\ell) \in S_f(i)$, so $f(\underline{x}) \leqslant c \cdot i$. Furthermore, the definition of f' together with the fact that $\underline{x}' \in S_{f'}(i)$ imply that $f'(\underline{x}') = f(\underline{x})$, hence $f'(\underline{x}') \leqslant c \cdot i$. Second, assume that $f'(\underline{x}') \leqslant i$. Since $f'(\underline{x}') = f(z_1, \cdots, z_\ell)$ with z_j being defined as above for $j \in [1, \ell]$, there exists $\underline{z} := (\tilde{z}_1, \cdots, \tilde{z}_\ell) \in S_f(i)$ such that $z_j \leqslant \tilde{z}_j$ for each $j \in [1, \ell]$. Moreover,

$$\underline{z}' := (\tilde{z}_1, \cdots, \tilde{z}_\ell, g_1^{-1}(\tilde{z}_{h(1)}), \cdots, g_t^{-1}(\tilde{z}_{h(t)})) \in S_{f'}(i).$$

The definition of $\underline{z}$ ensures that if $j \in [1, \ell]$ then $(\underline{z}')_j = \tilde{z}_j \geqslant x_j$, and if $j \in [1, t]$ then $(\underline{z}')_{\ell+j} = g_j^{-1}(\tilde{z}_{h(j)}) \geqslant g_j^{-1}(z_{h(j)}) \geqslant y_j$, as desired. This proves that s_f is also a sequence-number function of f'.

Since $\Gamma \subseteq \Lambda'$, we conclude (by our assumption) that there exists a uniform local deterministic (respectively, randomized Las Vegas) algorithm $\mathcal{A}$ for Prob and $\mathcal{F}$ such that the (respectively, expected) running time of $\mathcal{A}$ over any configuration $(G, \mathbf{x}) \in \mathcal{F}$ is $O(f'^* \cdot s_{f'}(f'^*)) = O(f'^* \cdot s_f(f'^*))$, where $f'^* = f(\mathtt{q}_1^*, \mathtt{q}_2^* \cdots, \mathtt{q}_\ell^*, \mathtt{p}_{r'+1}^*, \mathtt{p}_{r'+2}^*, \cdots, \mathtt{p}_r^*\})$. The fact that f' is non-decreasing implies that

$$f'^* \leqslant f'(\mathtt{q}_1^*, \mathtt{q}_2^* \cdots, \mathtt{q}_\ell^*, g_1^{-1}(\mathtt{q}_{h(1)}^*), g_2^{-1}(\mathtt{q}_{h(2)}^*), \cdots, g_t^{-1}(\mathtt{q}_{h(t)}^*\})) = f^*.$$

The fact that s_f is non-decreasing implies that the (respectively, expected) running time of $\mathcal{A}$ is in fact bounded by $O(f^* \cdot s_f(f^*))$, as desired.

The above discussion shows that it is sufficient to prove the theorem assuming that $\Gamma \subseteq \Lambda$. Consider therefore the case where $\Gamma \subseteq \Lambda$. For convenience, let $\Gamma = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_r\}$ and $\Lambda = \{\mathtt{p}_1, \mathtt{p}_2, \cdots, \mathtt{p}_\ell\}$, where $\ell \geqslant r$. We now observe that without loss of generality, we can actually assume that $\Gamma = \Lambda$. Indeed, if $\ell > r$, we simply impose that A^Γ also requires estimates for the parameters $\mathtt{p}_{r+1}, \mathtt{p}_{r+2}, \cdots, \mathtt{p}_\ell$, that is, the operation of A^Γ requires such estimates but actually ignores them after obtaining them. This way, we obtain an algorithm A^Λ depending on Λ. Since f is non-decreasing,
$f(\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_r^*, \mathtt{p}_{r+1}^*, \mathtt{p}_{r+2}^*, \cdots, \mathtt{p}_\ell^*)$
$\leqslant f(\mathtt{p}_1^*, \mathtt{p}_2^*, \cdots, \mathtt{p}_r^*, \tilde{\mathtt{p}}_{r+1}, \tilde{\mathtt{p}}_{r+2}, \cdots, \tilde{\mathtt{p}}_\ell)$, where $\tilde{\mathtt{p}}_i$ is a good guess for

every $i \in [r+1\,, \ell]$. Hence, the running time of Algorithm A^{Λ} is also bounded by f. $\square$

The following corollary follows by applying the theorem above to the work of Barenboim and Elkin [6], by setting $\Gamma = \{a, n\}$ and $\Lambda = \{n\}$ since $a \leqslant n$.

Corollary 2. *There exists a uniform deterministic algorithm solving* `MIS` *on general graphs in time $O(f(a))$, where $f(a) = o(\log n)$ for graphs with arboricity $a = o(\sqrt{\log n})$, and $f(a) = O(\log n / \log \log n)$ for graphs with arboricity $a = O(\log^{1/2-\delta} n)$, for some constant $\delta \in (0\,, 1/2)$ (otherwise, $f(a) = n$).*

4.4 Running as Fast as the Fastest Algorithm

To illustrate the topic of the next theorem, consider the best known non-uniform algorithms for MIS, namely the algorithms of Barenboim and Elkin [4] and that of Kuhn [22], which run in time $O(\Delta + \log^* n)$ and require the knowledge of n and Δ, and the algorithm of Panconesi and Srinivasan [34], which runs in time $2^{O(\sqrt{\log n})}$ and requires the knowledge of n. Furthermore, consider the MIS algorithm of Barenboim and Elkin [6], which is very efficient for graphs with bounded arboricity a (let $f'(a)$ be the running time of their algorithm). If n, Δ and a are known to all nodes, then one can compare the running times of these algorithms and use the fastest one. That is, there exists a non-uniform algorithm $\mathcal{A}^{\{n,\Delta,a\}}$ that runs in time $T'(n, \Delta, a) \coloneqq \min\{2^{O(\sqrt{\log n})}, O(\Delta + \log^* n),\ f'(a)\}$.

Unfortunately, Theorem 10 does not allow us to transform $\mathcal{A}^{\{n,\Delta,a\}}$ into a uniform one—the reason being that the function $T(n, \Delta, a)$ bounding running time does not have a sequence number as specified in Theorem 10. On the other hand, as mentioned in Corollaries 1 and 2, Theorem 8 does allow us to transform each of the aforementioned algorithms into a uniform MIS algorithm, with time complexity $2^{O(\sqrt{\log n})}$, $O(\Delta + \log^* n)$, and $O(f(a))$, respectively. Nevertheless, since n, Δ and a are unknown to the nodes, it is not clear how to obtain from these transformed algorithms a uniform algorithm running in time $T(n, \Delta, a) := \min\{2^{O(\sqrt{\log n})}, O(\Delta + \log^* n), O(f(a))\}$. The following theorem solves this problem.

Theorem 11. *Consider a problem Prob, a family of instances $\mathcal{F}$. Let Λ_1 and Λ_2 be two sets of parameters. Suppose that there exists a $\Lambda_1 \cup \Lambda_2$-monotone pruning algorithm $\mathcal{P}$ for Prob and $\mathcal{F}$. Consider two uniform algorithms $\mathcal{U}_1$ and $\mathcal{U}_2$ whose running times are bounded by non-decreasing functions $f_1(\Lambda_1^*)$ and $f_2(\Lambda_2^*)$, respectively. Then there is a uniform algorithm with running time $O(f_{\min})$, where $f_{\min} := \min\{f_1(\Lambda_1^*), f_2(\Lambda_2^*)\}$.*

Proof. The basic idea behind the proof of theorem above is to run in iterations, such that each iteration i consists of running the quadruple $(\mathcal{U}_1; \mathcal{P}; \mathcal{U}_2; \mathcal{P})$, where $\mathcal{U}_1$ and $\mathcal{U}_2$ are executed for precisely 2^i rounds each. Hence, a correct solution will be produced in Iteration $s := \lceil \log f_{\min} \rceil$ or before. Since each iteration i lasts for roughly 2^{i+1} rounds (recall that the running time of $\mathcal{P}$ is constant), the running time is $O(f_{\min})$.

Formally, we define a sequence of uniform algorithms $(\mathcal{A}_i)_{i \in \mathbf{N}}$ as follows. For $i \in \mathbf{N}$, set $\mathcal{A}_{2i+1} := \hat{\mathcal{U}}_1$ and $\mathcal{A}_{2i+2} := \hat{\mathcal{U}}_2$, where $\hat{\mathcal{U}}_j$ is $\mathcal{U}_j$ restricted to 2^i rounds for $j \in \{1, 2\}$. Let π be the uniform alternating algorithm with respect to $(\mathcal{A}_i)_{i \in \mathbf{N}}$ and $\mathcal{P}$, that is $\pi := \mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3; \cdots$ where $\mathcal{B}_{2i+j} := \hat{\mathcal{U}}_j; \mathcal{P}$ for every $i \in \mathbf{N}$ and every $j \in \{1, 2\}$. Letting T_0 be the (constant) running time of $\mathcal{P}$, the running time of $\mathcal{B}_i$ is at most $2^i + T_0$, for every $i \in \mathbf{N}$.

Consider an instance $(G, \mathbf{x}) \in \mathcal{F}$. For each $(\mathtt{p}, \mathtt{q}) \in \Lambda_1 \times \Lambda_2$, let $\mathtt{p}^* := \mathtt{p}(G, \mathbf{x})$ and $\mathtt{q}^* := \mathtt{q}(G, \mathbf{x})$. Algorithm $\mathcal{B}_i$ operates on the configuration $(G_i, \mathbf{x}_i)$. Let $\mathtt{p} \in \Lambda_1 \cup \Lambda_2$. Because $\mathcal{P}$ is monotone with respect to $\Lambda_1 \cup \Lambda_2$, it follows by induction on i that $\mathtt{p}^* \geqslant \mathtt{p}(G_i, \mathbf{x}_i)$. Hence, the running time of $\mathcal{U}_j$ over $(G_i, \mathbf{x}_i)$ is upper bounded by $f_j(\Lambda_j^*)$ for every $i \in \mathbf{N}$ and each $j \in \{1, 2\}$. Thus, it follows that $V(G_{2k+2}) = \emptyset$ for the smallest k such that $2^k \geqslant f_{\min}$. Consequently, by Lemma 1, Algorithm π correctly solves Prob on $\mathcal{F}$. Hence, the tail of the alternating algorithm π is trivial, that is $\pi = \mathcal{B}_1; \mathcal{B}_2; \cdots; \mathcal{B}_{2k+1}$.

Since Algorithm $\mathcal{B}_i$ runs in at most $2^{\lceil i/2 \rceil} + T_0$ rounds, it follows that the running time of π is $O(2^k) = O(f_{\min})$, as asserted. $\square$

Theorem 1 follows as a direct corollary of Theorem 11, using Corollaries 1 and 2.

5 Uniform Coloring Algorithms

In general, we could not find a way to directly apply our transformers (e.g., the one given by Theorem 10) for the coloring problem. The main reason is that we could not find an efficient pruning algorithm for the coloring problem. Indeed, consider for example the $O(\Delta)$-coloring problem. The checking property of a pruning algorithm requires that, in particular, the nodes can locally decide whether they belong to a legal configuration. Locally checking that neighboring nodes have distinct colors is easy, however, to know whether a color is in the required range, namely, in $[1, O(\Delta)]$, seems difficult as the nodes do not know Δ. Moreover, the gluing property seems difficult to tackle also: after pruning a node with color c, none of its unpruned neighbors can be colored in color c. In other words, a correct solution on the non-pruned subgraph may not glue well with the pruned subgraph.

Nevertheless, we show in this section that several relatively general transformers can be used to obtain uniform coloring algorithms from non-uniform one. We focus on standard coloring problems in which the required number of colors is given as a function of Δ.

5.1 Uniform $(\Delta + 1)$-coloring algorithms

A standard trick (cf., [28, 30]) allows us to transform an efficient (with respect to n and Δ) MIS algorithm for general graphs into one for $(\Delta + 1)$-coloring. This is based on the observation that $(\Delta + 1)$-colorings of G and maximal independent sets of $G' := G \times K_{\Delta+1}$ are in one-to-one correspondence. This known transformation, however, uses the knowledge of Δ. Nevertheless, it is straightforward to check that a similar correspondence holds when G' is defined as follows. For each node $u \in V(G)$, take a clique of size $\deg_G(u)+1$ with vertices $u_1, u_2, \cdots, u_{\deg_G(u)+1}$. Now, for each $(u, v) \in E(G)$ and each $i \in [1, 1+\min\{\deg_G(u), \deg_G(v)\}]$, let $(u_i, v_i) \in E(G')$. The graph G' can be constructed locally without any global knowledge, hence we obtain Theorem 2 as a corollary of Theorem 1.

5.2 Uniform coloring with more than $\Delta+1$ colors

We now aim to provide a general transformer taking as input an efficient non-uniform coloring algorithm that uses $g(\Delta)$ colors (where $g(\Delta) > \Delta$) and produces an efficient uniform coloring algorithm that uses $O(g(\Delta))$ colors. We begin with the following definitions.

An *instance* for the coloring problem is a pair $(G, \mathbf{x})$ where G is a graph and $\mathbf{x}(v)$ contains a color $c(v)$ such that the collection $\{c(v) \mid v \in V(G)\}$ forms a coloring of G. (The color $c(v)$ can be the identity $\mathrm{Id}(v)$.) For a given family of graphs $\mathcal{G}$, we define $\mathcal{F}(\mathcal{G})$ to be the collection of instances $(G, \mathbf{x})$ for the coloring problem, where $G \in \mathcal{G}$.

Recall that many coloring algorithms consider the identities as colors, and relax the assumption that the identities are unique by replacing it with the weaker requirement that the set of initial colors forms a coloring. Given an instance $(G, \mathbf{x})$, let $m = m(G, \mathbf{x})$ be the largest integer i such that all identities (initial colors) are taken from $[1, i]$, in other words, m is the maximal identity. Note that m is a graph-parameter.

Recall the $\lambda(\tilde{\Delta}+1)$-coloring algorithms designed by Barenboim and Elkin [4] and Kuhn [22] (which generalize the $O(\tilde{\Delta}^2)$-coloring algorithm of Linial [28]). We would like to point out that, in fact, everything works similarly in these algorithms if one replaces n with m. That is, these $\lambda(\tilde{\Delta}+1)$-coloring algorithms can be viewed as requiring m and Δ and running in time $O(\tilde{\Delta}/\lambda + \log^* \tilde{m})$. The same is true for the edge-coloring algorithms of Barenboim and Elkin [7].

The following theorem implies that these algorithms can be transformed into uniform ones. In the theorem, we consider two sets of graph-parameters Γ and Λ such that (1) Γ is weakly-dominated by Λ, and (2) $\Gamma \subseteq \{\Delta, m\}$. Such a pair of sets of parameters is said to be *related*. We also need the following definition, which will be used for the function governing the number of colors used by the coloring algorithms. A function g is *moderately-fast* if and only if (1) g is moderately-increasing, and (2) there exists a polynomial P such that $x < g(x) < P(x)$ for every $x \in \mathbf{R}^+$.

Theorem 12. *Let Γ and Λ be two related collections of graph parameters, and let $\mathcal{A}^\Gamma$ be a $g(\tilde{\Delta})$-coloring algorithm with running time bounded by some function $f(\Lambda)$. If*

1. *there exists a sequence-number function s_f for f;*
2. *g is moderately-fast;*
3. *the dependence of f on m is bounded by a polylog; and*
4. *the dependence of f on Δ is moderately-slow;*

then there exists a uniform $O(g(\Delta))$-coloring algorithm running in time $O(f(\Lambda^*) \cdot s_f(f(\Lambda^*)))$.

Proof. Our first goal is to obtain a coloring algorithm that does not require m (and thus requires only Δ). For this purpose we first define the following problem.
The *strong list-coloring* (SLC) problem: a configuration for the SLC problem is a pair $(G, \mathbf{x}) \in \mathcal{F}(\mathcal{G})$ such that

1. there exists an integer $\hat{\Delta} \geqslant \Delta$ that is contained in $\cap_{v \in V(G)} \mathbf{x}(v)$; and
2. the input $\mathbf{x}(v)$ of every vertex $v \in V(G)$ contains a list $L(v) \subseteq [1\,, g(\hat{\Delta})] \times [1\,, \hat{\Delta}+1]$ of colors such that

$$\forall k \in [1\,, g(\hat{\Delta})], \quad |\{j \mid (k,j) \in L(v)\}| \geqslant \deg_G(v) + 1.$$

Given a configuration $(G, \mathbf{x}) \in \mathcal{F}(\mathcal{G})$, an output vector $\mathbf{y}$ is a *solution* for SLC if it forms a coloring and if $\mathbf{y}(v) \in L(v)$ for every node $v \in V(G)$. Observe that Condition (1) above implies that an instance to SLC already contains an upper bound $\tilde{\Delta}$ on Δ at the input of each node (this upper bound being the same at all nodes), therefore, in a sense, an algorithm for SLC can be designed assuming that each node knows $\tilde{\Delta}$. Condition (2) above informally implies that the list of colors $L(v)$ available at each node v contains $\deg_G(v) + 1$ copies of each color in the range $[1\,, g(\hat{\Delta})]$.

We now design a pruning algorithm for SLC. Consider a triplet $(G, \mathbf{x}, \hat{\mathbf{y}})$, where $(G, \mathbf{x})$ is a configuration for SLC and $\hat{\mathbf{y}}$ is some tentative assignment of colors. The set W of nodes to be pruned is the set of nodes u satisfying $\hat{\mathbf{y}}(u) \in L(u)$ and $\hat{\mathbf{y}}(u) \neq \hat{\mathbf{y}}(v)$ for all $v \in N_G(u)$. Algorithm $\mathcal{P}$ modifies the input for the nodes outside W as follows: for $u \in V \setminus W$, set

$$\mathbf{x}'(u) \coloneqq (\mathbf{x}(u) \setminus L(u)) \cup L'(u) \quad \text{where}$$
$$L'(u) \coloneqq L(u) \setminus \{\hat{\mathbf{y}}(v) \mid v \in N_G(u) \cap W\}\,.$$

In other words, the input at a node $u \in V \setminus W$ is changed so that the new list of available colors $L'(u)$ contains precisely $L(u)$ minus the colors assigned to those neighbors of u in G that belong to W.

Observe that if we start with a configuration $(G, \mathbf{x})$ for SLC then the output $(G', \mathbf{x}')$ of the pruning algorithm $\mathcal{P}$ is also a configuration for SLC. This is because, for every node v and every k, at most $\deg_W(v)$ pairs (k, j) were removed from the list $L(v)$ of v, where $\deg_W(v)$ is the number of neighbors of v that belong to W. On the other hand, the degree of v in G' is reduced by $\deg_W(v)$.

From $\mathcal{A}^\Gamma$, it is straightforward to design a local algorithm $\mathcal{B}^{\Gamma'}$ for *SLC* that depends on $\Gamma' \coloneqq \Gamma \setminus \{\Delta\}$. Specifically, let $B^{\Gamma'}$ consist of executing A^Γ using the good guess $\tilde{\Delta} \coloneqq \hat{\Delta}$ for the parameter Δ. Furthermore, if A^Γ outputs at v a color c, then $B^{\Gamma'}$ outputs the color (c, j) where $j \coloneqq \min\{s \mid (c, s) \in L(v)\}$.

Given an instance for SLC, we view $\hat{\Delta}$ as a (graph) parameter, and convert Λ to a new set of parameters Λ' by replacing Δ with

Δ'. Formally, if $\Delta \in \Lambda$ then set $\Lambda' \coloneqq \Lambda \setminus \Delta \cup \hat{\Delta}$, and otherwise, set $\Lambda' \coloneqq \Lambda$. Since Γ and Λ contain only graph parameters—and since $\hat{\Delta}$ is contained in all the inputs—we deduce that the pruning algorithm $\mathcal{P}$ is $\Gamma' \cup \Lambda'$-monotone.

Now, having the sets of parameters Γ' and Λ' in mind, Algorithm $\mathcal{B}^{\Gamma'}$, and the aforementioned pruning algorithm $\mathcal{P}$ for SLC, we apply Theorem 10 and obtain a uniform algorithm $\mathcal{B}$ for SLC and $\mathcal{F}(\mathcal{G})$ whose running time is $O(f(\Lambda'^{*}) \cdot s_f(f(\Lambda'^{*})))$.

We are now ready to specify the desired uniform $O(g(\Delta))$-coloring algorithm. We inductively define integers D_i for $i \in \mathbf{N}$ by setting

$$D_1 \coloneqq 1 \quad \text{and} \quad D_{i+1} \coloneqq \min\{\ell \mid g(\ell) \geqslant 2g(D_i)\}, \quad \text{for } i \geqslant 1.$$

Given an initial configuration $(G, \mathbf{x})$, we partition it according to the node degrees. For $i \in \mathbf{N}$, let G_i be the subgraph of G induced by the set of nodes $v \in G$ with $\deg_G(v) \in [D_i\,, D_{i+1}-1]$. Let $\mathbf{x}_i$ be the input $\mathbf{x}$ restricted to the nodes in G_i. The configuration $(G_i, \mathbf{x}_i) \in \mathcal{F}(\mathcal{G})$ is referred to as *layer* i. Note that nodes can figure out locally which layer they belong to. Observe also that $D_{i+1} - 1$ is an upper bound on node degrees in layer i.

The algorithm proceeds in two phases. In the first phase, each node in layer i is assigned the list of colors $L''_i \coloneqq [1\,, g(D_{i+1})] \times [1\,, D_{i+1} + 1]$, and the degree estimation $\hat{\Delta}_i \coloneqq D_{i+1}$. Each layer is now an instance of SLC and we execute Algorithm $\mathcal{B}$ in parallel on all layers. If Algorithm $\mathcal{B}$ assigns a color (c, j) to a node v in layer i then we change this color to $(g(D_{i+1})+c, j)$. Hence, each layer i is colored with colors taken from $L'_i \coloneqq [g(D_{i+1}) + 1\,, 2g(D_{i+1})] \times [1\,, D_{i+1} + 1]$.

Note that nodes in different layers have disjoint colors, and hence we obtain a coloring of the whole graph G. The number of colors in L'_i is at most $\lambda D_{i+1} g(D_{i+1})$, for some constant integer λ. From the definition of the integers D_i and the fact that g is moderately-increasing, it follows that the total number of colors used in this phase is $O(\Delta g(\Delta))$. Furthermore, the running time of this first phase of the algorithm is dominated by the running time of the algorithm on layer $i_{\max}$, where $i_{\max}$ is the maximal i such that layer i is non-empty. That is, the running time is at most $O(f(\Lambda'^*) \cdot s_f(f(\Lambda'^*)))$, where Λ'^* is the collection of correct parameters in Λ' for layer $i_{\max}$. Since $D_{i_{\max}+1} = O(\Delta)$ and since the dependence of f on Δ is moderately-slow, we infer that $f(\Lambda'^*) = O(f(\Lambda^*))$. Recalling that s_f is moderately-slow (by the definition), we deduce that the running time is $O(f(\Lambda^*) \cdot s_f(f(\Lambda^*)))$.

The second phase consists of running a second algorithm to change the set of possible colors of nodes in layer i from L'_i to $L_i := [g(D_{i+1})+1\,, 2g(D_{i+1})]$. Specifically, on layer i, we execute $\mathcal{A}^\Gamma$ using the guess $\tilde{\Delta} = D_{i+1}$ for the parameter Δ and the guess $\tilde{m} = \lambda D_{i+1} g(D_{i+1})$ for the parameter m (recall that $\Gamma \subseteq \{\Delta, m\}$). This procedure colors each layer with colors taken from the range $[1\,, g(D_{i+1})]$. Let v be in layer i and let $c(v)$ be the color assigned to v by $\mathcal{A}^\Gamma$. The final color of v given by our desired algorithm $\mathcal{A}$ is $g(D_{i+1})+c(v)$. Thus, the colors assigned to the nodes in layer i belong to $[g(D_{i+1}) + 1\,, 2g(D_{i+1})]$, and are therefore disjoint on different layers. The algorithm is executed on each layer independently, all in parallel, and hence, we obtain a coloring. Moreover, since g is moderately-increasing, the total number of colors used is $O(g(\Delta))$.

Recall that $D_{i+1} = O(\Delta)$ and $g(D_{i+1}) = O(g(\Delta))$ for all i such that G_i is not empty. Hence, we deduce that the running time of the second phase of the algorithm is bounded from above by the running time of $\mathcal{A}^\Gamma$ on $(G, \mathbf{x})$ using the guesses $\tilde{\Delta} = O(\Delta)$ and $\tilde{m} = O(\Delta g(\Delta))$. Moreover, the fact that $g(x)$ is bounded by a polynomial in x implies that $\tilde{m}$ is at most polynomial in Δ, and hence in m.

Now, because the dependence of f on Δ is moderately-slow and the dependence of f on m is polylogarithmic, the running time of the second phase of $\mathcal{A}$ is $O(f(\Lambda))$. Combining this with the running time of the first phase concludes the proof. $\square$

Recall from Observation 2.1 that the constant function $s_f = 1$ is a sequence-function for every additive function f. Hence, Theorem 3 now follows as a direct corollary of the Theorem 12. Regarding edge-coloring, observe that Barenboim and Elkin [7] obtain their edge-coloring algorithm on general graphs by running a vertex-coloring algorithm on the line-graph of the given graph. This vertex-coloring algorithm assumes the knowledge of m and Δ and uses the same number of color and time complexity as the resulted edge-coloring algorithm. Using Theorem 12, one can transform that vertex-coloring algorithm [7] designed for the family of line graphs into a uniform one, having the same asymptotic number of colors and running time. Hence, Theorem 5 follows.

Let $f : \mathbf{N}^2 \to \mathbf{R}$ be a function given by $f(x_1, x_2) := f_1(x_1) \cdot f_2(x_2)$, where f_1 and f_2 are non-negative ascending functions. Recall from Observation 2.1 that the function $s_f(i) := \lceil \log i \rceil + 1$ is a sequence-number function for f. Therefore, Theorem 4 now follows by applying Theorem 12 to the coloring algorithms of Barenboim and Elkin [5].

6 Conclusion and further research

6.1 Pruning algorithms

This paper focuses on removing assumptions concerning global knowledge in the context of local algorithms. We provide rather general transformers taking a non-uniform local algorithm as a black box and producing a uniform algorithm running in asymptotically the same number of rounds. This is established via the notion of pruning algorithms. We believe that this novel notion is of independent interest and can be used for other purposes too, e.g., in the context of fault tolerance or dynamic settings.

We would like to remind the reader that we restrict the running time of a pruning algorithm to be constant. This is because in all our applications we use constant time pruning algorithms. In fact, our results extend to the case where the given *uniform* pruning algorithm $\mathcal{P}$ runs in time bounded by $h(\mathcal{S})$, where h is non-decreasing, $\mathcal{S}$ is a set of parameters, and $\mathcal{P}$ is $\mathcal{S}$-monotone. We note, however, that this generalization may incur an additive overhead in the running time of our transformations, as these repeatedly use $\mathcal{P}$. Specifically, the overhead will be $h(\mathcal{S}^*)$ times the number of iterations used by the transformer (which is typically logarithmic in the running time of the non-uniform algorithm). It would be interesting to have an example of a problem that admits a fast non-trivial uniform pruning algorithm but does not admit a constant time one.

6.2 Bounded message size

This paper focuses on the $\mathcal{LOCAL}$ model which does not restrict the number of bits used in messages. Ideally, messages should be short, i.e., using $O(\log n)$ bits. We found it difficult to obtain a general transformer that takes an arbitrary non-uniform algorithm which uses short messages and produces a uniform one having the same asymptotic running time and message size. The reason is that using similar techniques to the ones we use, one would need to use guesses that fit to both the function bounding the running time as well as to the function bounding the message size. Nevertheless, we note that maintaining the same message size may still be possible given particular non-uniform algorithms that use messages whose content does not depend on the guessed upper bounds, for example, algorithms that encode in the messages only identifiers, colors, or degrees.

6.3 Coloring

Recall that one of the difficulties in obtaining a pruning algorithm for coloring problems lies in the fact that a pruned node v with color c may have a non-pruned neighbor u which is also colored c in some correct coloring of the non-pruned subgraph, that is, the gluing property may not hold. In the context of running in iterations, in which one invokes a pruning algorithm and subsequently, an algorithm $\mathcal{A}$ on the non-pruned subgraph (similarly to Theorem 10), the aforementioned undesired phenomena could be prevented if the algorithm $\mathcal{A}$ would avoid coloring node u with color c. With this respect, we believe that it would be interesting to investigate connections between

g-coloring problems and *strong* g-coloring problems, in which each node v is given as an input a list of (forbidden) colors $F(v)$. In a correct solution, each node v must color itself with a color not in $L(v)$ such that the final configuration is a coloring and such that the total number of colors used does not exceed g.

Finally, recall that our transformer for coloring applies to deterministic algorithms only. It would be interesting to design a general transformer that takes non-uniform randomized coloring algorithms (e.g., the ones by Schneider and Wattenhofer [36]) and transforms them to uniform ones with asymptotically the same running time.

References

[1] N. Alon, L. Babai, and A. Itai, *A fast and simple randomized parallel algorithm for the maximal independent set problem.*, J. Algorithms, 7 (1986), pp. 567–583.

[2] B. Awerbuch, *Complexity of network synchronization*, J. ACM, 32 (1985), pp. 804–823.

[3] B. Awerbuch, M. Luby, A. V. Goldberg, and S. A. Plotkin, *Network decomposition and locality in distributed computation*, in FOCS, 1989, pp. 364–369.

[4] L. Barenboim and M. Elkin, *Distributed* $(\Delta+1)$*-coloring in linear (in* Δ*) time*, in STOC, 2009, pp. 111–120.

[5] ——, *Deterministic distributed vertex coloring in polylogarithmic time*, in PODC, 2010, pp. 410–419.

[6] ——, *Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition*, Distrib. Comput., 22 (2010), pp. 363–379.

[7] ——, *Distributed deterministic edge coloring using bounded neighborhood independence*, in PODC, 2011. Available at http://arxiv.org/abs/1010.2454.

[8] J. L. Bentley and A. C.-C. Yao, *An almost optimal algorithm for unbounded searching*, Inf. Process. Lett., 5 (1976), pp. 82–87.

[9] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg, *Label-guided graph exploration by a finite automaton*, ACM Trans. Algorithms, 4 (2008), pp. 42:1–42:18.

[10] R. Cole and U. Vishkin, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, in STOC, 1986, pp. 206–219.

[11] B. Derbel, C. Gavoille, D. Peleg, and L. Viennot, *On the locality of distributed sparse spanner construction*, in PODC, 2008, pp. 273–282.

[12] D. Dereniowski and A. Pelc, *Drawing maps with advice*, in DISC, Springer-Verlag, 2010, pp. 328–342.

[13] P. Fraigniaud, C. Gavoille, D. Ilcinkas, and A. Pelc, *Distributed computing with advice: information sensitivity of graph coloring*, Distrib. Comput., 21 (2009), pp. 395–403.

[14] P. Fraigniaud, D. Ilcinkas, and A. Pelc, *Communication algorithms with advice*, J. Comput. Syst. Sci., 76 (2010), pp. 222–232.

[15] P. Fraigniaud, A. Korman, and E. Lebhar, *Local mst computation with short advice*, in SPAA, 2007, pp. 154–160.

[16] P. Fraigniaud, A. Korman, and D. Peleg, *Local distributed decision.* Submitted for Publication.

[17] A. V. Goldberg and S. A. Plotkin, *Efficient parallel algorithms for $(\Delta + 1)$-coloring and maximal independent set problem*, in STOC, 1987, pp. 315–324.

[18] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon, *Parallel symmetry-breaking in sparse graphs*, SIAM J. Discrete Math., 1 (1988), pp. 434–446.

[19] M. Hańćkowiak, M. Karoński, and A. Panconesi, *On the distributed complexity of computing maximal matchings*, SIAM J. Discrete Math., 15 (2001/02), pp. 41–57.

[20] A. Korman and S. Kutten, *Distributed verification of minimum spanning trees*, Distrib. Comput., 20 (2007), pp. 253–266.

[21] A. Korman, S. Kutten, and D. Peleg, *Proof labeling schemes*, Distrib. Comput., 22 (2010), pp. 215–233.

[22] F. Kuhn, *Weak graph colorings: distributed algorithms and applications*, in SPAA, 2009, pp. 138–144.

[23] F. Kuhn, T. Moscibroda, and R. Wattenhofer, *What cannot be computed locally!*, in PODC, 2004, pp. 300–309.

[24] F. Kuhn and R. Wattenhofer, *On the complexity of distributed graph coloring*, in PODC, 2006, pp. 7–15.

[25] S. Kutten and D. Peleg, *Tight Fault Locality*, SIAM J. Comput. 30(1): 247-268 (2000).

[26] C. Lenzen, Y. Oswald, and R. Wattenhofer, *What can be approximated locally?: case study: dominating sets in planar graphs*, in SPAA, 2008, pp. 46–54.

[27] N. Linial, *Distributive graph algorithms global solutions from local data*, in FOCS, 1987, pp. 331–335.

[28] ———, *Locality in distributed graph algorithms*, SIAM J. Comput., 21 (1992), p. 193.

[29] Z. Lotker, B. Patt-Shamir, and A. Rosén, *Distributed approximate matching*, SIAM J. Comput., 39 (2009), pp. 445–460.

[30] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.

[31] K. Nakano and S. Olariu, *Uniform leader election protocols for radio networks*, IEEE Trans. Parallel Distrib. Syst., 13 (2002), pp. 516–526.

[32] M. Naor and L. Stockmeyer, *What can be computed locally?*, SIAM J. Comput., 24 (1995), pp. 1259–1277.

[33] A. Panconesi and R. Rizzi, *Some simple distributed algorithms for sparse networks*, Distrib. Comput., 14 (2001), pp. 97–100.

[34] A. Panconesi and A. Srinivasan, *On the complexity of distributed network decomposition*, J. Algorithms, 20 (1996), pp. 356–374.

[35] D. Peleg, *Distributed computing. A locality-sensitive approach.*, SIAM Monographs on Discrete Mathematics and Applications, SIAM, 343 p. , 2000.

[36] J. Schneider and R. Wattenhofer, *A new technique for distributed symmetry breaking*, in PODC, 2010, pp. 257–266.

[37] ——, *An optimal maximal independent set algorithm for bounded-independence graphs*, Distrib. Comput., 22 (2010), pp. 1–13.

[38] M. Szegedy and S. Vishwanathan, *Locality based graph coloring*, in STOC, 1993, pp. 201–207.