Diss. ETH No. 21663

Complexity of Optimization Problems: Advice and Approximation

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by

Lucia Keller

Dipl. Math. ETH, ETH Zurich
born on July 24, 1982
citizen of Wettingen and Villigen (Aargau)

accepted on the recommendation of

Prof. Dr. Juraj Hromkovič, examiner
Prof. Dr. Peter Widmayer, co-examiner
Priv.-Doz. Dr. Walter Unger, co-examiner

2014

Abstract

Many optimization problems are computationally hard in the sense that no efficient algorithms are known for computing an optimal solution. Since many so-called $\mathcal{NP}$-hard problems are practically relevant, many sophisticated techniques have been developed to cope with these problems in order to get a satisfactory solution in reasonable time. Moreover, one often has to deal with incomplete or erroneous information which often makes even those problems hard that are easily solvable in the presence of full information.

One standard scenario of missing information are the so-called online optimization problems. Here, the input arrives piecewise step by step and a part of the output has to be irrevocably produced already for each arriving slice of the input. An online optimization problem can be seen as a game in which an adversary tries to create the hardest possible instances and feeds them piecewise to the online player who tries to compute the best possible piecewise output.

We consider two classical optimization problems in this online setting, graph coloring and finding a maximum matching in a graph. Both problems are efficiently solvable in the classical (offline) setting when restricted to bipartite graphs, but become hard in the online case.

Online coloring is one of the most studied online optimization problems. The instance arrives vertex by vertex and the task is to assign colors to the vertices such that no neighboring vertices receive the same color, in the way that as few colors as possible are used. For offline bipartite graphs, this task is easy. We show that, in the online scenario, the adversary is able to create an online bipartite graph on n vertices such that every online algorithm has to use at least $\lfloor 1.13746 \cdot \log_2(n) - 0.49887 \rfloor$ colors, improving the previously known lower bound of $\lfloor \log_2(n) \rfloor + 1$ colors.

The main drawback of online algorithms is that they do not know what happens in the future. To measure the impact of this disadvantage, one can allow an oracle which knows the instance and the algorithm and computes, based on this knowledge, an advice string which can be accessed by the algorithm during its computation.

We applied this new tool to online coloring, showing an almost matching bound for the number of advice bits needed if the algorithm is expected to output an optimal solution. Then, we also study some tradeoffs allowing the algorithm not to be optimal but using less advice bits instead, and consider also subclasses such as paths, cycles, and spider graphs.
Another important online optimization problem is the search for a maximum matching in an online graph. Online matching has practical applications everywhere where one wants to associate something with something else. For bipartite graphs, there are two models that have been investigated in the literature. In the one-sided model, one shore is fixed and only the vertices of the opposite shore arrive in consecutive time steps. We study the other model, the so-called fully online matching problem, in which the vertices from both shores are revealed piecewise. In the case of general graphs, a greedy online algorithm can already reach the best possible competitive ratio of 2, even without advice, whereas on paths, only a competitive ratio of approximately $\frac{3}{2}$ can be achieved. There is an optimal online algorithm with advice for paths accessing only approximately $\frac{n}{3}$ advice bits. For paths, special trees and other special bipartite graphs, we show often almost tight upper and lower bounds on the number of advice bits necessary to be optimal, and we investigate what competitive ratios can be reached with less advice bits.
Another obstacle we have to overcome when modelling some practical situation, is that we encounter erroneous information. Many real-world situations can be modeled by Boolean formulas. Here, the variables represent some parameters of the problem at hand and the clauses of the formula represent various constraints. If the constructed formula for an input instance is satisfiable, every satisfying assignment gives us a solution, i. e., a feasible setting of the parameters. If the formula is not satisfiable, one can try to satisfy as many constraints as possible, this leads to the well-studied maximum satisfiability problem. But, besides conflicting constraints, there is another possible source of errors in modeling, namely mapping two different parameters of the real-world problem to one variable. To tackle this problem, we try to split a minimum number of variables into two (one for the positive occurrences and one for the negative ones) such as to make the given formula satisfiable. This is known to be a very hard optimization problem, thus, we restrict our attention to the special case of 2-CNF Horn formulas, i. e., 2-CNF formulas without positive 2-clauses. We prove that this restricted problem is still hard to approximate. We also analyze subcases of 2-CNF Horn formulas, where additional clause types are forbidden. While excluding negative 2-clauses admits a polynomial-time algorithm based on network flows, the splitting problem stays APX-hard, even for formulas consisting of positive 1-clauses and negative 2-clauses only.

Zusammenfassung

Viele Optimierungsprobleme sind algorithmisch schwer in dem Sinne, dass kein effizienter Algorithmus für die Berechnung von optimalen Lösungen existiert. Weil viele dieser sogenannten $\mathcal{NP}$-schweren Probleme praxisrelevant sind, sind viele ausgeklügelte Techniken entwickelt worden, um diese Probleme so zu behandeln, dass eine zufriedenstellende Lösung in vernünftiger Zeit gefunden wird. Ausserdem bekommt man oft unvollständige oder fehlerhafte Informationen, die sogar solche Probleme schwer machen, die einfach zu lösen wären, wenn die ganze Information vorhanden wäre.

Sogenannte Online-Optimierungsprobleme beschreiben eine übliche Art von fehlenden Informationen. Hier kommt die Eingabe stückweise, Schritt für Schritt, und ein Teil der Ausgabe soll jeweils für jedes angekommene Stück der Ausgabe sofort und unwiderruflich berechnet werden. Ein Online-Optimierungsproblem kann betrachtet werden als ein Spiel, in welchem Gegenspieler versucht, eine möglichst schwierige Eingabe zu finden, und diese dem Online-Spieler stückweise übermittelt, welcher seinerseits versucht, eine möglichst gute Ausgabe stückweise zu generieren.

Wir betrachten zwei klassische Optimierungsprobleme in diesem online Szenario, Färbung von Graphen und die Suche nach einem Maximum-Matching in einem Graphen. Beide Probleme sind in der klassischen (Offline-) Formulierung und eingeschränkt auf bipartite Graphen effizient lösbar, aber sie werden im Online-Fall schwer.

Eines der am besten untersuchten Online-Optimierungsprobleme ist die Online-Färbung von Graphen. Die Instanzen erscheinen stückweise, Knoten für Knoten, und die Aufgabe ist es, den Knoten Farben zuzuordnen, so dass keine benachbarten Knoten die gleiche Farbe bekommen, und dass dabei so wenige Farben wie nur möglich gebraucht werden. Für bipartite Offline-Graphen ist diese Aufgabe einfach. Wir zeigen, dass im Online-Fall der Gegenspieler in der Lage ist, einen bipartiten Online-Graphen mit n Knoten zu konstruieren, so dass jeder Online-Algorithmus mindestens $\lfloor 1.13746 \cdot \log_2(n) - 0.49887 \rfloor$ Farben brauchen muss, was die schon bekannte untere Schranke von $\lfloor \log_2(n) \rfloor + 1$ Farben verbessert.

Der grosse Nachteil von Online-Algorithmen ist, dass sie nicht wissen, was in der Zukunft passieren wird. Um die Auswirkung dieser Einschränkung zu messen, kann man ein Orakel erlauben, welches die Instanz und den Algorithmus kennt und basierend auf diesem Wissen einen Advice-String berechnet, auf den der Algorithmus während der Berechnung zugreifen darf. Wir haben dieses neue Hilfsmittel auf die Online-Färbung angewendet und fast zusammenfallende untere und obere Schranken für die Anzahl der Advice-Bits gezeigt, falls vom Algorithmus erwartet wird, dass er eine optimale Lösung ausgibt. Dann haben wir auch ein paar Kompromisse studiert, die dem Algorithmus erlauben, nicht optimal zu sein, er aber dafür weniger Advice-Bits lesen kann, und wir haben auch Unterklassen wie Pfade, Kreise und Spinnengraphen angeschaut.
Ein weiteres wichtiges Online-Optimierungsproblem ist die Suche nach einem Maximum-Matching in einem Online-Graphen. Online-Matching hat überall dort praktische Anwendung, wo jemand etwas zu etwas anderem zuordnen möchte. Für bipartite Graphen gibt es zwei Modelle, die in der Literatur untersucht worden sind. Im One-Sided-Modell ist eine Seite fixiert und nur die Knoten der gegenüberliegenden Seite tauchen in nacheinanderfolgenden Schritten auf. Wir studieren das andere Modell, das sogenannte Fully-Online-Matching Problem, in welchem die Knoten auf beiden Seiten nacheinander aufgedeckt werden. Im Falle von allgemeinen Graphen erreicht sogar schon ein Greedy-Algorithmus die bestmögliche kompetitive Güte von 2 sogar ohne Advice, wohingegen auf Pfaden nur eine kompetitive Güte von $\frac{3}{2}$ erreicht werden kann. Es gibt einen optimalen Online-Algorithmus mit Advice für Pfade, der nur ungefähr $\frac{n}{3}$ Advice-Bits liest. Für Pfade, spezielle Bäume und andere spezielle bipartite Graphen sind wir oft in der Lage, fast zusammenfallende untere und obere Schranken für die Anzahl von nötigen Advice-Bits zu zeigen, die gebraucht werden, um optimal zu sein, und welche kompetitive Güte mit weniger Advice-Bits erreicht werden kann.
Ein anderes Hindernis, das wir bei der Modellierung von praktischen Aufgabenstellungen überwinden müssen, ist, dass wir fehlerhafte Informationen antreffen. Viele alltägliche Situationen können durch Boolesche Formeln modelliert werden. Hier repräsentieren die Variablen gewisse Parameter des Problems und die Klauseln der Formel repräsentieren verschiedene Einschränkungen. Wenn die konstruierte Formel für eine Eingabeinstanz erfüllbar ist, liefert uns jede erfüllbare Belegung eine Lösung, also eine zulässige Festlegung der Parameter. Wenn die Formel nicht erfüllbar ist, kann man versuchen, so viele Einschränkungen wie nur möglich zu erfüllen, was zum gut untersuchten Maximum-Satisfiability-Problem führt. Aber neben diesen sich widersprechenden Einschränkungen gibt es noch eine andere Fehlerquelle in der Modellierung, dass nämlich zwei verschiedene Parameter der Anwendung auf eine Variable abgebildet wurden. Um dieses Problem zu bewältigen, versuchen wir eine minimale Anzahl von Variablen jeweils in zwei Variablen aufzuspalten (eine für die positiven Vorkommen und eine für die negativen Vorkommen), so dass die Formel dadurch erfüllbar wird. Dieses Problem ist als ein sehr schweres Optimierungsproblem bekannt, deshalb schränken wir uns auf den Spezialfall von 2-CNF-Horn-Formeln ein, also auf Formeln in 2-CNF ohne positive 2-Klauseln. Wir

zeigen, dass man dieses eingeschränkte Modell immer noch schwer approximieren kann. Wir analysieren auch Unterklassen von 2-CNF-Horn-Formeln, in denen noch zusätzliche Klauseltypen verboten sind. Während das Ausschliessen von negativen 2-Klauseln einen Polynomzeit-Algorithmus zulässt, der auf Netzwerk-Flüssen basiert, bleibt das Problem sogar schon für Formeln, die nur aus positiven 1-Klauseln und negativen 2-Klauseln bestehen, APX-schwer.

Acknowledgments

At first, I would like to express my gratitude to my supervisor Juraj Hromkovič who introduced me to the exciting world of theoretical computer science. From the first minute on, when I first visited one of his lectures, I was fascinated of this topic and also of the way he conveyed this to the students. I am so grateful that, besides the research, he taught me how to become a good teacher, and I am very thankful that our common work will continue after finishing this thesis.
Then, I want to thank to Hans-Joachim Böckenhauer for his unmeasurable help. I enjoyed the innumerable, fruitful discussions, and I am deeply grateful that he always encouraged me and supported me whenever I needed help. For sure, he is also the best proofreader a PhD-student can imagine.
I would also like to give thanks to all members of the Chair of Information Technology and Education, not only for the recreative breaks, but also for the numerous stimulating discussions, and for always supplying me with coffee and cakes. Especially, I want to thank Björn Steffen, Dennis Komm and Giovanni Serafini for always helping me patiently when I needed help, giving me inspiration and motivation.
I am thankful to Peter Widmayer and Walter Unger that they agreed to review my thesis, and for the enumerous productive discussions and for giving me inspiring ideas for my thesis.
Most importantly, I want to thank to my family and especially to my parents for their overwhelming support and care since 31 years and for always believing in me. Last but not least, I am in deepest dept to Ivan Di Caro who always supported and strengthened me and without whom I would never come so far.

"The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity."

Albert Einstein (1955)

Contents

1 Introduction and Preliminaries **1**
1.1 Mathematical Foundations . . . 4
Notations . . . 4
Sequences and Combinatorial Tools . . . 5
Graphs . . . 6
Some Other Graph Classes and Their Properties . . . 8
Boolean Formulas . . . 8
1.2 Optimization Problems and Approximation . . . 9
Decision Problems . . . 10
Optimization Problems . . . 11
Approximation of Optimization Problems . . . 12
1.3 Online Algorithms and Competitive Analysis . . . 14
1.4 Online Algorithms With Advice . . . 20
Analyzing Online Algorithms: A Game Between an Online Player and an Adversary . . . 22
Self-Delimiting Codes . . . 24

2 Online Coloring of Graphs **27**
k-Colorable Graphs . . . 27
Bipartite Graphs . . . 28
2.1 Preliminaries . . . 28
2.2 Online Coloring Without Advice . . . 30
2.3 Proof of Lemma 2.8 . . . 34
Base Cases ($k \leq 3$) . . . 35
Inductive Step ($k \geq 4$) . . . 36
2.4 Advice Complexity in Bipartite Graphs . . . 41
2.5 Advice Complexity in Paths, Cycles, and Spider Graphs . . . 48
Paths . . . 48
Cycles . . . 49

Spider Graphs . . . 51
2.6 Conclusion . . . 55

3 Online Matching in Bipartite Graphs **57**
3.1 Paths and Trees Without Advice . . . 60
Competitive Ratio for Online Algorithms Without Advice on Paths 60
Competitive Ratio for Online Algorithms Without Advice on Trees 67
3.2 Optimality in Paths and Cycles . . . 69
3.3 Lower Bound for Optimality in Trees . . . 79
3.4 Upper Bounds for Optimality in Trees . . . 82
Trees With Restricted Vertex Degrees . . . 82
A Special Maximum Matching . . . 85
One First Algorithm for Finding a Maximum Matching in {1,3}-Trees 85
An Improved Algorithm for Finding a Maximum Matching in {1,3}-Trees . . . 92
3.5 Optimality in General and Bipartite Graphs . . . 101
Upper Bound for General Graphs . . . 101
Lower Bound for Bipartite Graphs . . . 103
P_k-free Bipartite Graphs . . . 104
Graphs With Small Diameter . . . 107
3.6 Tradeoffs in Paths . . . 107
A Lower Bound for One Advice Bit . . . 108
Not Enough Advice Bits to Be Optimal . . . 113
3.7 Tradeoffs in Paths and Trees via the String Guessing Problem . . . 120
An Upper Bound for the Online Matching Problem on Paths . . . 121
A Lower Bound on the Online Matching Problem in Trees . . . 125
3.8 Conclusion . . . 128

4 Approximability of Splitting-SAT in 2-CNF Horn Formulas **131**
4.1 Basic Definitions . . . 133
4.2 Splitting in 2-CNF Horn Formulas . . . 136
4.3 Maximum Assignment in 2-CNF Horn Formulas . . . 140
4.4 Maximum Assignment in Exact-2-CNF Formulas . . . 142
4.5 Conclusion . . . 143

Bibliography **145**

Introduction and Preliminaries

In 1936, Alan Turing [63] and Alonzo Church [16], among others, found two different formalizations of the intuitive concept of computability. Following Turing, every algorithmically solvable problem can be solved by a Turing machine. This precise notion of an algorithm was necessary to show, based on the Incompleteness Theorem of Gödel [32], that there are problems which are not algorithmically solvable. This insight concluded the search for a proof of the Entscheidungsproblem of David Hilbert formulated in 1928 [34], who – motivated by the first mechanical calculating machine of Gottfried Wilhelm von Leibniz in 1785 – was convinced that every mathematical decision problem is algorithmically solvable [36].

In the early 1960s – after researchers were able to classify the problems into automatically solvable and unsolvable – the technology started to get better and better, and therefore the question arose which of the automatically solvable problems are also practically solvable in the sense that they can be solved by computers within a reasonable time. The main goal of complexity theory is to classify the computationally solvable problems into easy and hard problems with respect to their computational complexity, i. e., the amount of work a computer has take in order to solve the problem [36, 59].

Although more than fifty years of intensive research in complexity did not succeed in unconditionally proving the computational intractability of practical relevant (computational) problems, there exists a class of problems that are widely believed to be intractable, i. e., not solvable within polynomial time. These problems are referred to as $\mathcal{NP}$-hard problems, the target of this thesis.

Because many $\mathcal{NP}$-hard problems have practical applications, theoretical computer scientists are motivated trying to tackle these problems with sophisticated techniques in order to get a satisfying solution in reasonable time. There are many approaches for this, such as approximation algorithms (allowing suboptimal

solutions), randomized algorithms (allowing to toss coins during the computation), heuristics (experience-based techniques) and many more.
Complexity theory can also be extended to problems where, instead of taking a binary decision, a solution with optimized costs has to be found. We call these problems optimization problems. The goal is either to maximize (maximization problems) or to minimize (minimization problems) the costs. The definition of $\mathcal{NP}$-hardness can be transferred from decision problems to optimization problems by considering decision problems asking if there is a solution of at least a fixed size for maximization problems or at least of this size for minimization problems. If this corresponding decision problem is $\mathcal{NP}$-hard, then the original optimization problem is hard as well.
The probably most crucial decision problem is to decide whether a Boolean formula is satisfiable, i. e., whether we can assign truth values to the variables such that the formula evaluates to true. In this thesis, we take care especially of the formulas which are not satisfiable and try to make them satisfiable using a new approach. The idea behind this is that Boolean formulas are often used to model practical applications and the variables of the formula represent some parameters of the problem. If the computed formula is not satisfiable, this might arise from the fact that different parameters have been modeled by only one variable. Thus, replacing the positive and the negative literals by two new variables can help to make the formula satisfiable. The goal is to find a smallest subset of such variables such that the formula is satisfiable after applying this operation. This new approach complements the classical maximum satisfiability approach where one tries to satisfy as many constraints (modeled by clauses) as possible of the problem instance at hand [3].
A variation of optimization problems we consider in this thesis is even nearer to many real-life situations, because an instance often is not presented at once but comes in step by step. A job agency, for example, receives applications for an employment day by day and they have to try to assign the applicants a suitable job within a short period of time. They cannot wait infinitely long for the best candidate for a particular job. Thus, they have to make decisions every day without knowing the future.
In online optimization problems, the instance is presented in several rounds, and the algorithm immediately has to decide on the solution for the already revealed part. This answer cannot be changed in later time steps. Clearly, online algorithms encounter a harder situation than offline algorithms. Nevertheless, the quality of an online algorithm is usually compared – using the so-called competitive ratio – with the best solution, which is usually only reachable in offline scenarios. This competitive analysis was first introduced in [60].
Since this comparison seems to be a little unfair, and since we would like to have a more fine-grained analysis of online optimization problems with respect to the knowledge about future requests, we use a new model which was first introduced in [12, 21, 38]. The idea is to allow the algorithm to make use of an oracle which knows the whole instance the algorithm will receive. Based on this knowledge,

the oracle computes an advice string which is a part of an infinite advice tape. Accessing this advice tape, the algorithm gets additional information which can help him to solve the instance optimally or at least better than without reading any advice. The number of advice bits read help us to classify online algorithms into different classes with respect to the amount of advice they need to be optimal. Some online optimization problems need only little advice, but some need almost the whole knowledge about future requests. Especially for the second type of problems, it also makes sense to look at a tradeoff of advice bits and the competitive ratio which can be reached with this fixed amount of available advice bits.

In this thesis, two online optimization problems will be of special interest, online coloring and online matching. Both problems are defined on graphs in which the vertices are revealed one by one with all the vertices that are adjacent to already present vertices.

Graph coloring is one of the most prominent optimization problems. Already the offline version of the problem is hard even for many restricted graph classes, but online graph coloring belongs to the hardest online optimization problems. Even for bipartite, i. e., two-colorable graphs, no deterministic online algorithm can guarantee to use less than a logarithmic number of colors. In this thesis, we consider the problem of coloring bipartite graphs online with and without advice.

Also the matching problem belongs to the most studied classical optimization problems with many applications, e. g., in economics. While being efficiently solvable in the offline case, its online version is hard even for bipartite graphs. Most attention has been paid to the so-called one-sided matching, where the vertices from one shore of the bipartite graph are given beforehand and the vertices from the other shore appear in an online fashion [41]. In this thesis, we consider the fully online version, where the vertices from both shores appear online.

This thesis is organized as follows: In the remaining part of this chapter, we fix our notation and recall some mathematical background we use in this thesis. Then, we define optimization problems and the concept of approximating those problems. In the last part, we introduce online algorithms formally, give the idea of competitive analysis, and present the concept of using advice in online algorithms.

In Chapter 2, we deal with online coloring. The first part is devoted to general bipartite graphs, where we first improve the existing lower bound on solving the online coloring problem deterministically without advice on bipartite graphs from $\lfloor \log(n) \rfloor + 1$ to $\lfloor 1.13746 \cdot \log(n) - 0.49887 \rfloor$. This is followed by a discussion on the advice complexity of bipartite graphs in general, and of paths, cycles and spider graphs.

Chapter 3 deals with the online matching problem. We summarize what is known on deterministic algorithms without advice on paths and trees, complement these results, and examine afterwards how much advice is needed to be optimal by giving almost tight lower and upper bounds in paths and cycles. After giving a lower bound for optimality in special trees with restricted vertex degrees, we show two alternative algorithms for upper bounds. Then, we investigate some more general graph classes and give an upper bound for general graphs, a lower bound

for bipartite graphs and make a short detour to P_k-free bipartite graphs and to bipartite graphs with a small diameter. This chapter is concluded with a section on tradeoffs between the number of advice bits an algorithm needs to read and the competitive ratio it can reach with this amount of advice.
Last, we introduce in Chapter 4 a new approach for making CNF-formulas satisfiable by splitting some of the variables to their positive and their negative literals. We try to minimize the number of split variables, which is very hard to approximate in general. Thus, we focus on 2-SAT Horn formulas here, where we are able to give an almost full picture of the approximability of variable splitting.

1.1 Mathematical Foundations

First, we want to fix some mathematical notation and introduce the mathematical background we need in this thesis.

Notations

By $\mathbb{N}$ we denote the *set of natural numbers* including zero, $\mathbb{N} = \{0, 1, 2, \ldots\}$. If we address only the positive natural numbers, we write $\mathbb{N}^+$ or $\mathbb{N}^{\geq 1}$. We take the same notation for integer numbers ($\mathbb{Z}$), rational numbers ($\mathbb{Q}$) and real numbers ($\mathbb{R}$). Other sets are denoted by ordinary capital letters, i. e., S, and for classes we write calligraphical symbols, e. g., $\mathcal{G}$. By $|S|$ we denote the cardinality of the set S (the number of elements in the set).
A *partition* of a set S is a set of subsets $S_1, S_2, \ldots, S_n$, $S_i \subseteq S$ for all $i \in \{1, 2, \ldots, n\}$, such that the union of these sets covers S, i. e., $\bigcup_{i=1}^{n} S_i$, and each two of these sets are pairwise disjoint, $S_i \cap S_j = \varnothing$ for $i \neq j$.
We denote the integer part of a real value x by $\lfloor x \rfloor$, defined as

$$\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leq x\}\,,$$

and the smallest integer number greater than x by

$$\lceil x \rceil = \min\{m \in \mathbb{Z} \mid m \geq x\}\,.$$

The logarithm function is denoted by $\log_b(x)$ for an arbitrary base b. Since we use mainly the logarithm function for base 2, we write $\log(x)$ instead of $\log_2(x)$ for the *binary logarithm*. The *natural logarithm* is denoted by $\ln(x)$ for the base e, indicating the Eulerian number $\mathrm{e} = \lim_{n\to\infty}(1 + \frac{1}{n})^n = 2.7182\ldots$. For some $x, y, p \in \mathbb{R}$, we use the following rules to analyze terms containing a logarithm function: $\log_b(x \cdot y) = \log_b(x) + \log_b(y)$ (product), $\log_b(x/y) = \log_b(x) - \log_b(y)$ (quotient), $\log_b(x^p) = p\log_b(x)$ (power), $\log_b(x) = (\log_a(x))\,/\,(\log_a(b))$ (change of base).

To investigate the asymptotic behaviour of two positive functions $f(n)$ and $g(n)$ for some variable n, we use the Landau symbols (also referred to as big-O notation) that are described, e. g., in [37]. They are defined as

$$\begin{aligned}
f(n) \in \mathcal{O}(g(n)) &\iff \exists n_0, c > 0 \text{ such that } \forall n > n_0 : f(n) \leq c \cdot g(n) \\
f(n) \in \Omega(g(n)) &\iff \exists n_0, c > 0 \text{ such that } \forall n > n_0 : c \cdot g(n) \leq f(n) \\
f(n) \in \Theta(g(n)) &\iff f(n) \in \mathcal{O}(g(n)) \text{ and } f(n) \in \Omega(g(n))
\end{aligned}$$

Sequences and Combinatorial Tools

In Chapter 2, we use in a proof a generalization of the *Fibonacci numbers* $F(n)$, recursively defined as

$$\begin{aligned}
&F(0) = 0\,, \\
&F(1) = 1\,, \\
&F(n) = F(n-1) + F(n-2)\,, \text{ for any integer number } n > 1.
\end{aligned}$$

Definition 1.1 (Tribonacci Numbers). For *Tribonacci numbers*, denoted by $T(n)$ (see [27,61]), the sequence starts with three predetermined numbers and each new number is the sum of the three preceding numbers,

$$\begin{aligned}
&T(0) = 0\,, \\
&T(1) = 0\,, \\
&T(2) = 1\,, \\
&T(n) = T(n-1) + T(n-2) + T(n-3)\,, \text{ for any } n > 2.
\end{aligned}$$

This leads to the sequence $0, 0, 1, 1, 2, 4, 7, 13, 24, \ldots$ (see [61]). The number $T(n)$ can be computed as follows:

$$T(n) = 3b \cdot \frac{\left(\frac{1}{3}(a_+ + a_- + 1)\right)^n}{b^2 - 2b + 4} \leq 0.336229 \cdot 1.83929^n, \tag{1.1}$$

where $a_+ = (19 + 3\sqrt{33})^{\frac{1}{3}}$, $a_- = (19 - 3\sqrt{33})^{\frac{1}{3}}$, and $b = (586 + 102\sqrt{33})^{\frac{1}{3}}$.
A useful tool to get lower and upper bounds on the *factorial* of n, i. e., $n! = 1 \cdot 2 \cdot \ldots \cdot n$ is the approximation of James Stirling, called *Stirling's formula* [22], given by

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{11n}\right). \tag{1.2}$$

This formula can be especially used to give bounds on the *binomial coefficient*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

indicating the number of subsets of size k in a set of size n. To get the number of all subsets of an arbitrary size, we can sum up all binomial coefficients for an arbitrary k, and get

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n .$$

Suppose that $n = 2k$ is even for an $k \in \mathbb{N}$. Using Stirling's inequality, we get an upper and a lower bound on the central binomial coefficient,

$$\frac{4^k}{2\sqrt{\pi k}} < \binom{2k}{k} < \frac{4^k}{\sqrt{2\pi k}} . \tag{1.3}$$

A *permutation* of a set $\{a_1, a_2, \ldots, a_n\}$ is a bijection

$$\pi \colon \{a_1, a_2, \ldots, a_n\} \to \{a_1, a_2, \ldots, a_n\}$$

such that $(\pi(a_1), \pi(a_2), \ldots, \pi(a_n))$ is an ordered set and $\pi(a_i)$ indicates the position of a_i in the this ordered set. The number of permutations of a set containing n elements is $n!$.

Graphs

Many practical problems, such as trying to color a map with only four colors such that no two neighboring countries receive the same color, can be modeled by a graph. Graphs are a powerful tool for visualizing many problems in order to find a good solution. A detailed introduction to graph theory can be found in [66].

Definition 1.2 (Graph). An (undirected) *graph* $G = (V(G), E(G))$ consists of a finite set of vertices $V(G) = \{v_1, v_2, \ldots, v_n\}$ and a set of edges

$$E \subseteq \{\{v_i, v_j\} \mid v_i, v_j \in V \text{ for } i \neq j\}$$

connecting the vertices.
In a directed graph, *digraph* for short, the edges have additionally a direction, and we write (v_i, v_j) for an edge e pointing from v_i to v_j. We call the directed edges *arcs* to distinguish them from ordinary edges.

We say that an edge $e = \{v_i, v_j\}$ is *incident* to its two end vertices v_i and v_j, and that a vertex v_i is *incident* to the edge e. Two vertices v_i and v_j are *adjacent* if they are connected by an edge $e = \{v_i, v_j\}$. Similarly we define that two edges $e_1 = \{v_i, v_j\}$ and $e_2 = \{v_i, v_k\}$ are adjacent, if they have a common vertex v_i. Adjacent vertices are called *neighbors* or *neighboring vertices*. The *degree* of a vertex v is the number of incident edges.
We call a graph $G = (V, E)$ *complete*, if all possible edges appear in the graph, i. e., $E = \{\{v_i, v_j\} \mid v_i, v_j \in E \text{ for every } i \neq j\}$.

We only consider so-called *simple* graphs in which there is at most one edge connecting two vertices, and no loops are present, i. e., there is no edge $e = \{v, v\}$, for a vertex v. In this thesis, we will mostly focus on the following graph class and subclasses of this class.

Definition 1.3 (Bipartite Graph). In a *bipartite graph* $G = (V(G), E(G))$, the vertex set $V(G)$ can be partitioned into two subsets, called *shores* and denoted by $S_1(G)$ and $S_2(G)$, with the property that the edges in $E(G)$ connect only vertices from different shores.

If the graph is clear from the context, we write V, E, S_1, S_2 instead of $V(G)$, $E(G)$, $S_1(G)$ and $S_2(G)$. If, in a bipartite graph $G = (V, E)$, all edges between the two shores $S_1(G)$ and $S_2(G)$ are present, we call this special bipartite graph a *complete bipartite graph.*
A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. We will focus on special subgraphs, namely those for which every edge connecting vertices of V' in G appears in the subgraph G'.

Definition 1.4 (Induced Subgraph). A graph $G' = (V', E')$ is an *induced subgraph* of $G = (V, E)$, if $V' \subseteq V$, $E' \subseteq E$ and, for all $v_i, v_j \in V'$,

$$\{v_i, v_j\} \in E(G) \iff \{v_i, v_j\} \in E(G')\,.$$

We denote the subgraph of $G = (V, E)$ induced by a vertex subset $V' \subseteq V$ by $G[V']$, i. e., $G[V'] = (V', E')$, where $E' = \{\{v_i, v_j\} \in E \mid v_i, v_j \in V'\}$.

One important subclass of bipartite graphs are paths. They are also used to define some structures in general graphs.

Definition 1.5 (Path). A *path* is a graph containing only vertices of degree 2, except of two vertices that have degree 1. The vertices of degree one are called the *end vertices.*
We call a path a *u,v-path* if the end vertices are u and v.

The *length* of a path is its number of edges. We denote by P_n a path on n vertices, i. e., a path of length $n - 1$. Clearly, every path is a bipartite graph since one shore consists of the vertices on odd positions and the other shore of the vertices at even positions.

Definition 1.6 (Connected, Isolated). A graph G is *connected* if it has a u,v-path for all $u, v \in V(G)$. We call a maximal connected subgraph, i. e., a connected subgraph to which no other vertex of $V(G)$ can be added without losing connectedeness, a *component* of the graph.
An *isolated vertex* is a vertex of degree 0.

Paths can also be used to measure some properties in general graphs, for example, that every vertex can be reached from every other vertex via a short path. The *diameter* of a graph is the maximum distance of two arbitrary vertices. The distance of two vertices u and v is measured by the length of the shortest u-v-path, i. e., the number of edges on this path.

Some Other Graph Classes and Their Properties

In the following, we explain the graph classes we examine in this thesis.

Cycles

Connecting the two end vertices of a path, we get a cycle. A *cycle* is a connected graph with exclusively vertices of degree 2. Cycles containing an even number of vertices are bipartite graphs, whereas cycles with an odd number of vertices are not bipartite.

Trees

Trees are graphs not containing any cycle as a subgraph. Special trees are *caterpillars* in which there is a path (called the *spine*) that contains every vertex of degree two or more. A *comb* is a caterpillar in which all vertices on the spine have vertex degree 3, except the first and the last vertex, which have degree 2. A caterpillar with a spine containing only one vertex is called a *spider graph*.
In trees, vertices of degree 1 are called *leaves* and all other vertices are referred to as *inner vertices*.

P_k-free graphs

We can also define graph classes by indicating which induced subgraphs are forbidden. A graph G not containing a path P_k on k vertices, for some $k \in \mathbb{N}$, as an induced subgraph is called a *P_k-free graph*. We will consider P_k-free bipartite graphs.

Boolean Formulas

For the following definitions we adopt the notation from the detailed lecture notes from [65]. *Satisfiability* (SAT) is the problem of deciding whether a Boolean formula evaluates to true. This problem is the most important $\mathcal{NP}$-complete problem [17]. Boolean formulas consist of variables that can take the values 1 (for true) and 0 (for false). They are referred to as *Boolean variables*. In this thesis we only consider formulas in conjunctive normal form (CNF). As operators in these formulas we use the conjunction ($\wedge$), disjunction ($\vee$), and the negation ($\neg$) (see Table 1.1).
A *literal* is a variable (*positive literal*) or the negation of a variable (*negative literal*), instead of $\neg x$ we write $\overline{x}$ for short for a negated variable.

Table 1.1. Truth tables for the conjunction, disjunction and negation for some Boolean variables x and y.

Conjunction

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

Disjunction

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Negation

x	$\neg x$
0	1
1	0

Definition 1.7 (Conjunctive Normal Form (CNF)). A Boolean formula Φ over a variable set $X = \{x_1, \ldots, x_n\}$ is in *conjunctive normal form*, *CNF* for short, if it is a conjunction of so-called clauses, where a clause is a disjunction of literals. A CNF formula with m clauses is of the form

$$\bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} x_{i,j} \right)$$

for literals $x_{i,j}$.

In Chapter 4, we consider CNF-formulas with bounded clause sizes. If the number of literals in a clause is bounded by some constant k, we say that a Boolean formula Φ is in k-CNF.

Definition 1.8 (Assignment). Let $V = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. An *assignment* is a mapping $\alpha : V \to \{0, 1\}$ such that α assigns to every variable a truth value, e. g., $\alpha(x_1) = 0, \alpha(x_2) = 1, \ldots, \alpha(x_n) = 1$. For negative literals $\overline{x}$, we have $\alpha(\overline{x}) = 1 - \alpha(x)$.

A Boolean formula is *satisfiable* if there is an assignment α such that the formula evaluates to 1. This assignment α *satisfies* the formula.

1.2 Optimization Problems and Approximation

There are two important classes of computational problems. The inputs of so-called decision and optimization problems are *words*, encoded over a given finite set of symbols (the *alphabet*). Note that for these problems, we usually consider alphabets of size at least two, since a unary encoding blows up the size of encodings over a binary alphabet by an exponential factor. The change of the size of the encodings between alphabets of size greater than one is only polynomial. A subset of the set of all words over a given alphabet is called a *language*. A detailed description of both problem classes and many paradigmatic examples can be found in [37].

Decision Problems

In *decision problems*, one has to decide whether a given input has a described property or not. One of the most famous decision problems is the *satisfiability problem* (SAT), where for a given Boolean formula in CNF, we have to decide if the formula is satisfiable or not.

Definition 1.9 (Satisfiability Problem (SAT)).

Input: A CNF-formula Φ.

Output: "yes" if Φ is satisfiable, and "no" otherwise.

Decision problems are always defined for an infinite set of problem instances from a certain class[1]. For the problem SAT, the set of instances contains all CNF-formulas. We can also define a decision problem via the language of all "yes" answers. In the case of SAT, the language

$$\text{SAT} = \{\Phi \mid \Phi \text{ is a satisfiable CNF-formula}\}$$

describes the decision problem in the sense that all words of this language would involve a "yes" answer.
For decision problems, the famous classes $\mathcal{P}$ and $\mathcal{NP}$ denote the sets of all languages (decision problems) that are decidable within polynomial time on a deterministic or a nondeterministic Turing machine, respectively. We consider a language (decision problem) $L \in \mathcal{P}$ as practically solvable.
One usually assumes that $\mathcal{P} \neq \mathcal{NP}$. There is no known method for proving directly for a language $L \notin \mathcal{NP}$, that $L \notin \mathcal{P}$. Therefore, we define a relative set of languages in $\mathcal{NP}$ such that if, for one of these languages one could find an efficient algorithm, all of the problems in this set would be efficiently solvable. This can be done using the following type of reductions.

Definition 1.10 ($\mathcal{NP}$-hard, $\mathcal{NP}$-complete). Let L_1 and L_2 be two languages over some alphabets Σ_1 and Σ_2. We say the L_1 is *polynomial-time reducible* to L_2 and write $L_1 \leq_p L_2$, if there exists a polynomial-time bounded algorithm $A\colon \Sigma_1 \to \Sigma_2$ such that, for all $I \in \Sigma_1$,

$$I \in L_1 \iff A(I) \in L_2.$$

A language L is called $\mathcal{NP}$*-hard* if $\tilde{L} \leq_p L$, for all $\tilde{L} \in \mathcal{NP}$. A language L is $\mathcal{NP}$*-complete* if $L \in \mathcal{NP}$ and L is $\mathcal{NP}$-hard.

We consider the class of $\mathcal{NP}$-complete problems to be the hardest problems in $\mathcal{NP}$. If one could show for an $\mathcal{NP}$-complete problem that $L \in \mathcal{P}$, we would have $\mathcal{P} = \mathcal{NP}$.

[1] For a finite set of instances, there always exists an algorithm with a finite look-up table of precomputed answers.

If we can show a reduction of an $\mathcal{NP}$-hard language L_1 to a language L_2, we can conclude that L_2 is $\mathcal{NP}$-hard, too. Whereas, if for a language L_2 that is decidable in polynomial time, we have $L_1 \leq_p L_2$, also L_1 is decidable in polynomial time.

Optimization Problems

In *optimization problems*, the best solution from a set of feasible solutions has to be found. In the *maximum satisfiability problem* (Max-Sat), for a given Boolean formula in CNF, the goal is to find an assignment that satisfies as many clauses as possible. All assignments for a given formula are feasible solutions. The best of them is the one which maximizes the number of satisfied clauses.

Definition 1.11 (Maximum Satisfiability Problem (Max-Sat)).

Input: A CNF-formula Φ.

Set of feasible solutions: All assignments for Φ.

Cost: Number of satisfied clauses.

Goal: Maximum.

In this thesis, we focus on optimization problems. Formally, an optimization problem is defined as follows.

Definition 1.12 (Optimization Problem). An *optimization problem* can be described by a 4-tuple $U = (\mathcal{I}, \mathcal{M}, \text{cost}, \text{goal})$, where

- $\mathcal{I}$ is the set of admissible inputs I,
- $\mathcal{M}(I)$ is the set of feasible solutions for an input I,
- $\text{cost}(I, O)$ is the cost function assigning to every pair (I, O), for a feasible solution $O \in \mathcal{M}(I)$ of an input $I \in \mathcal{I}$, a positive real number, and
- $\text{goal} = \min$ is the optimization goal for *minimization problems*, and $\text{goal} = \max$ for *maximization problems*.

A feasible solution $O \in \mathcal{M}(I)$ is called *optimal* for an input I if

$$\text{cost}(I, O) = \text{goal}\{\text{cost}(I, \tilde{O}) \mid \tilde{O} \in \mathcal{M}(I)\}$$

holds. We denote by $\text{Opt}_U(I)$ the cost of an optimal solution.

If the input I is clear from the context, we write $\text{cost}(O)$ instead of $\text{cost}(I, O)$. We also abbreviate $\text{Opt}_U(I)$ by $\text{Opt}(I)$. An algorithm A is called *consistent* for an

optimization problem U, if $A(I) \in \mathcal{M}(I)$, for all $I \in \mathcal{I}$, i. e., if every output $A(I)$, for any input $I \in \mathcal{I}$, of A is a feasible solution.

We can carry the notion of $\mathcal{NP}$-hardness of decision problems over to optimization problems. Therefore we need a correspondence between these two types of problems. For an optimization problem, we can define a corresponding decision problem, called the *threshold language*, by asking if an optimal solution for an instance falls below (for minimization problems) or exceeds (for maximization problems) a certain threshold.

Definition 1.13 (Threshold Language). Let $U = (\mathcal{I}, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem. For minimization problems, the *threshold language* for U is defined as

$$\text{Lang}_U = \{(I, a) \in L \times \{0,1\}^* \mid \text{Opt}_U(I) \leq \text{Number}(a)\},$$

where Number(a) denotes the natural number whose binary representation is a.
For maximization problems, the *threshold language* for U is defined as

$$\text{Lang}_U = \{(I, a) \in L \times \{0,1\}^* \mid \text{Opt}_U(I) \geq \text{Number}(a)\}.$$

This definition allows us to transfer the notion of $\mathcal{NP}$-hardness also to optimization problems. We say that an optimization problem is $\mathcal{NP}$-hard, if its threshold language is $\mathcal{NP}$-hard. This is justified by the fact that, assuming $\mathcal{P} \neq \mathcal{NP}$, there is no polynomial-time algorithm for an $\mathcal{NP}$-hard optimization problem U.

Note that the threshold language of an optimization problem is not always the classical decision problem. In the case of Max-Sat, the threshold language of Max-Sat does not comply with the decision problem Sat, where all clauses have to be satisfied if the answer of the decision is "yes".

Approximation of Optimization Problems

Because $\mathcal{NP}$-hard optimization problems are not solvable within reasonable time in general, one can consider approximation algorithms allowing a suboptimal solution. To measure the quality of such algorithms, we can compute the ratio between the cost of the optimal solution and the cost the algorithm can reach for a given instance.

Definition 1.14 (Approximation Ratio). Let $U = (\mathcal{I}, \mathcal{M}, \text{cost}, \text{goal})$ be an optimization problem, let A be a consistent algorithm for U. For every $I \in \mathcal{I}$, the *approximation ratio* of A on I is defined as

$$R_A(I) = \max\left\{\frac{\text{cost}(A(I))}{\text{Opt}(I)}, \frac{\text{Opt}(I)}{\text{cost}(A(I))}\right\},$$

where $\text{Opt}(I)$ denotes the costs of an optimal solution for I.
For $n \in \mathbb{N}$, the *approximation ratio* of A is defined as

$$r(n) = \max\{R_A(I) \mid I \in \mathcal{I} \text{ and } |I| = n\}.$$

For some $\delta > 1$, we say that A is a δ-approximation algorithm if $r(n) \le \delta$, for all $n \in \mathbb{N}$.
For some function $f\colon \mathbb{N} \to \mathbb{R}^{\ge 0}$, we say that A is a $f(n)$-approximation algorithm if $r(n) \le f(n)$, for all $n \in \mathbb{N}$.

The first term in the definition of the approximation ratio is used for minimization problems, the second one for maximization problem. In this way, the approximation ratio is defined to be always ≥ 1.
By $\mathcal{APX}$, we denote the set of all optimization problems which admit a polynomial-time c-approximation algorithm, for some constant $c \ge 1$.
We say that an optimization problem admits a *polynomial-time approximation scheme* (PTAS) if, for any $\varepsilon > 0$, there is a polynomial-time $(1+\varepsilon)$-approximation algorithm for U.
Since hardly any proof techniques are known that allow to show unconditional lower bounds on the time complexity or the achievable approximation ratio for some problem, hardness is usually shown by using reductions, i. e., one shows that an efficient solution for one problem would imply efficient solution for a whole class of problems.
Similarly as for the time complexity, one can use reductions to prove that, under some complexity-theoretic assumption like $\mathcal{P} \neq \mathcal{NP}$, it is hard to reach a certain approximation ratio within polynomial time. Such a so-called *approximation-preserving reduction*, or *AP-reduction* for short, does not only have to take only polynomial time, but it has to be constructed in such a way that the approximation ratio carries over from one problem to the other.

Definition 1.15 (AP-Reduction). An AP-reduction (see Figure 1.1) between two optimization problems U_1 and U_2 consists of one function F transforming inputs I for U_1 into inputs $F(I,\delta)$, for any $\delta \in \mathbb{Q}^+$, for U_2 within polynomial time and a second function H transforming feasible solutions for $F(I,\delta)$ back to feasible solutions for I such that the approximation ratio stays the same or is at most changed by some constant factor α.

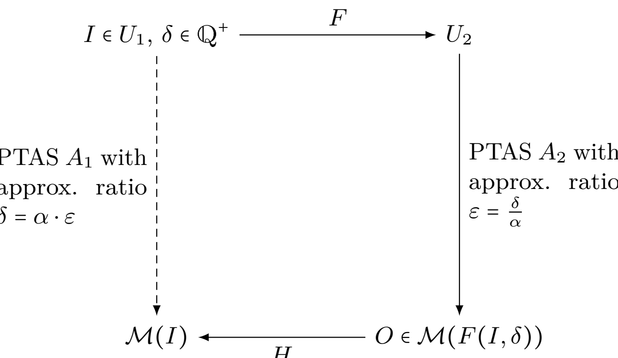


Figure 1.1. The structure of an AP-reduction according to Definition 1.15.

If there exists an AP-reduction from U_1 to U_2, we say that U_1 is *AP-reducible* to U_2 and write $U_1 \leq_{AP} U_2$. For a formal definition of AP-reductions, see [37].
We say that a problem U is $\mathcal{APX}$-hard if, for all $\tilde{U} \in \mathcal{APX}$, $\tilde{U} \leq_{AP} U$. Similar to the $\mathcal{NP}$-hardness of decision problems, if there is an AP-reduction from an optimization problem U_1 to an optimization problem U_2, i. e., if $U_1 \leq_{AP} U_2$, we can conclude the following:

1. If U_1 is $\mathcal{APX}$-hard, then also U_2 is $\mathcal{APX}$-hard, i. e., if there is no PTAS for U_1, there is also no PTAS for U_2.

2. If there is a PTAS for U_2, there is also one for U_1.

3. U_1 can be approximated with the same ratio as U_2, up to a constant factor, i. e., if U_2 allows a polynomial-time $(1 + \varepsilon)$-approximation, we can design a polynomial-time $(1 + \alpha\varepsilon)$-approximation algorithm for U_1, for some $\alpha = \frac{\delta}{\varepsilon}$.

1.3 Online Algorithms and Competitive Analysis

In an online optimization problem, the input is revealed piecewise in consecutive time steps and an irrevocable part of the output has to be produced at each time step. Obviously, it is harder and sometimes even impossible to compute the best solution for an instance if the instance is not shown in its whole length from the beginning. For a detailed introduction and an overview of online optimization problems and algorithms, see, e. g., [14]. Online optimization problems represent those real-life situations, in which the entire input is not available before the algorithm starts to work. Incrementally, the algorithm gets more and more information and it has to take irrevocable decisions on every new part of the input.
Consider, for example, the work of an employment agency. The job center receives job offers daily from companies and, independent from these offers, job applications

of candidates with a certain profile. Clearly, the job center cannot wait infinitely long to find the best candidate for an open position. Sometimes, the corresponding company has a deadline when they need a new employee, but also the employment agency is interested in finding as soon as possible some candidate since they earn their money from the commission. So, it may happen that the best candidate applies for a job right after filling the announced position. If the job center would know this, for sure they would wait one day longer to assign a candidate to the job. Actually, in this scenario no human being knows if at a time step i the best candidate appeared, since, if the employment agency does not break down, this procedure goes on and on.

The probably most famous introductory example for an online optimization problem is the ski-rental problem (see, e. g., [14]): A tourist being for the first time in the Swiss mountains and not possessing any pair of skis, has to decide in the beginning of his vacation if he wants to rent skis, paying x CHF per day, or if he buys a new pair of skis for y CHF, for some prices $x \ll y$. Suppose, he is in the mountains for the whole season, but he only wants to ski on sunny days. Therefore, the decision depends on the number of nice days. But he does not know in advance, how many days he needs his skis. For sure, if there are many sunny days, the best strategy would be to buy the skis. But, if he uses the skis less than $\lceil \frac{y}{x} \rceil$ times, to rent them would be a better idea. In this example, the goal is to minimize the cost of skiing.

In general, an online optimization problem is defined as follows. In this thesis, we use the notation from [45].

Definition 1.16 (Online Optimization Problem). An *online optimization problem* consists of a set $\mathcal{I}$ of inputs and a cost function. Every input $I \in \mathcal{I}$ is a sequence of requests $x_1, x_2, \ldots, x_n$, denoted as $I = (x_1, x_2, \ldots, x_n)$. Every input $I \in \mathcal{I}$ is associated with a set $\mathcal{O}$ of feasible solutions (outputs). Every solution $O \in \mathcal{O}$ is composed of a sequence of answers $O = (y_1, y_2, \ldots, y_n)$. The cost function assigns to every pair of inputs and solutions, (I, O), a real value $\text{cost}(I, O)$. The goal is to optimize the cost, i. e.,

- to minimize the costs in *online minimization problems*, or
- to maximize the costs in *online maximization problems*.

If the input I is clear from the context, we omit the I in the notation of the cost function $\text{cost}(I, O)$ and write $\text{cost}(O)$ instead. For an instance $I = (x_1, x_2, \ldots, x_n)$, the index i of an x_i indicates, in which *time step* the request x_i is given.

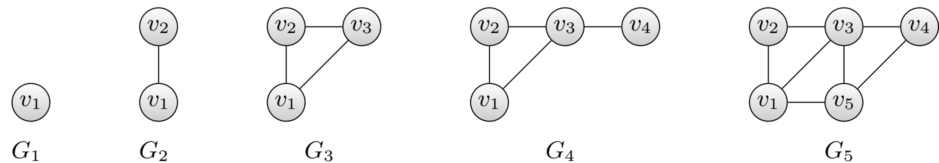


Figure 1.2. The sequence $(G_1, G_2, G_3, G_4, G_5)$ is a possible online presentation of the (offline) graph $G = G_5 = (V, E)$ with vertex set $V = \{v_1, v_2, v_3, v_4, v_5\}$ and edge set E as depicted above.

Definition 1.17 (Optimal (Offline) Solution). An *optimal (offline) solution* for an instance $I \in \mathcal{I}$ is a solution $\mathrm{Opt}(I) \in \mathcal{O}$ such that

$$\mathrm{cost}(\mathrm{Opt}(I)) = \min_{O \in \mathcal{O}} \{\mathrm{cost}(O) \mid O \text{ is a feasible solution for } I\}$$

in case of an online minimization problem, or

$$\mathrm{cost}(\mathrm{Opt}(I)) = \max_{O \in \mathcal{O}} \{\mathrm{cost}(O) \mid O \text{ is a feasible solution for } I\}$$

in case of an online maximization problem.

In this thesis, we discuss online graph problems. A possible online presentation of a graph $G = (V, E)$ is to reveal the graph vertex by vertex with the edges adjacent to already present vertices as shown in Figure 1.2. For a vertex set $V = \{v_1, v_2, \ldots, v_n\}$, the index i of a vertex v_i indicates the order in which the vertices appear.

In the first part of the thesis, we focus on the problem of coloring an online graph properly, i. e., assigning to every vertex a color from a set of colors, e. g., from the set $\{1, 2, \ldots, n\}$, such that no two neighbors get the same color. In the online scenario of this problem, the online algorithm has to decide immediately after a vertex v_i from the vertex set V is revealed, which color it should receive (see Figure 1.3 for an example). The already defined colors cannot be changed in later time steps. The goal is to use as few colors as possible.

To make the definition of an online optimization problem easier to read, we will use the following input/output notation for online optimization problems in this thesis.

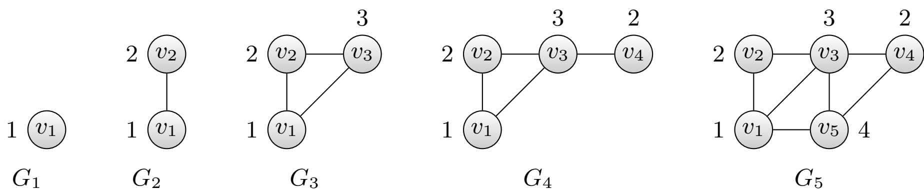


Figure 1.3. A possible coloring for the online graph instance $G^{\prec}$ given by the sequence $(G_1, G_2, G_3, G_4, G_5)$. This is not an optimal coloring. If the algorithm would choose color 1 for v_4, vertex v_5 could be colored with color 1, resulting in a coloring with only three colors.

Definition 1.18 (Online Optimization Problem). An *online optimization problem* is defined as:

Input: An online instance $I = (x_1, x_2, \dots, x_n)$ of the instance class $\mathcal{I}$, consisting of requests x_i for every time step $i \in \{1, 2, \dots, n\}$.

Output: Sequence of solutions $O = (y_1, y_2, \dots, y_n)$ such that y_i is computed only based on the first part $(x_1, \dots, x_i)$ of the sequence of requests.

Cost: The cost function assigns to every pair (I, O) the value $\text{cost}(I, O)$.

Goal: Maximizing (*online maximization problem*) or minimizing (*online minimization problem*) the cost.

Algorithms solving an online optimization problem are referred to as *online algorithms.*

Definition 1.19 (Online Algorithm). Let $I = (x_1, x_2, \dots, x_n)$ be an input of an online optimization problem. An *online algorithm* computes the output sequence $A(I) = (y_1, y_2, \dots, y_n)$ such that y_i is computed from the first i requests $x_1, \dots, x_i$ and the already determined corresponding outputs $y_1, \dots, y_{i-1}$ only.
We denote by $\text{cost}(A(I))$ the *cost of the algorithm* A used to compute a solution for the instance I. The *cost of the optimal (offline) solution* is identified with $\text{cost}(\text{Opt}(I))$.

In the online coloring problem of graphs, the cost function counts the number of colors used. In the ski-rental problem, the cost is determined by the amount of money the tourist has to invest, i. e., for k sunny days out of n days we have

$$\text{cost} = \begin{cases} k \cdot x\,, & \text{if the tourist decides to rent the skis} \\ y\,, & \text{else}\,. \end{cases}$$

In online scenarios, the algorithm cannot foresee future requests. Therefore, it is at a disadvantage with respect to solving the problem offline where the whole input is given in advance. In the example of Figure 1.3 for an online coloring instance, we see that an optimal (offline) algorithm would, for example, assign color 1 to v_4 enabling to color v_5 with color 2, yielding a better coloring using only 3 colors.
To measure the quality of online algorithms, we compare the cost of the computed solution with the cost of an optimal solution (computed offline) in a ratio of these two costs. For the online coloring algorithm in Figure 1.3, we have a ratio of

$$\frac{\text{cost}(A(I))}{\text{cost}(\text{Opt}(I))} = \frac{4}{3}\,.$$

To avoid problems with small-size special cases, one defines the competitive ratio of an online algorithm in a slightly more general way.

Definition 1.20 (Competitive Ratio).

- An online algorithm A for a **minimization problem** is *c-competitive* if there exists a constant $\alpha \geq 0$ such that, for every input I,

$$\text{cost}(A(I)) \leq c \cdot \text{cost}(\text{Opt}(I)) + \alpha\,.$$

- An online algorithm A for a **maximization problem** is called *c-competitive* if there is a constant $\alpha \geq 0$ such that, for every input I, we have

$$\text{cost}(\text{Opt}(I)) \leq c \cdot \text{cost}(A(I)) + \alpha\,.$$

The constant c is called the *competitive ratio* of A.
If the inequality holds for $\alpha = 0$, then the algorithm A is called *strictly c-competitive*.

Note that, in this thesis, we define the so-called competitive ratio, such that this ratio is always greater or equal to 1. In the literature, one can also find definitions for which the competitive ratio is smaller or equal to 1 for maximization problems. In both definitions, the online algorithm behaves optimally if this ratio turns out to be 1.
Classifying online algorithms based on their competitive ratio is called *competitive analysis* in the literature. An online algorithm is called *competitive*, if it has a competitive ratio not depending on the length of the input, whereas it is *non-competitive* if there is no constant competitive ratio for this algorithm. The constant α in the definition of the competitive ratio is necessary to allow the algorithm to work on short instances arbitrarily (they can be covered in the constant α), as long as the algorithm reaches a good competitive ratio for long instances.
More about *competitive analysis* can be found in [40,60,67]. See [14] for the history of competitive analysis.

Figure 1.4. An example of an online path $P^{\prec} = (P, \prec)$ with $P = (V, E)$ and $V = \{v_1, v_2, v_3, v_4\}$ such that some intermediate graphs are no paths.

In the following two chapters of this thesis, we will consider online optimization problems based on online graph instances that appear vertex by vertex and with each new vertex v_i all incident edges to previous vertices are revealed.

Definition 1.21 (Online Graph). An *online graph instance* $G^{\prec} = (G, \prec)$ consists of a graph $G = (V, E)$ and a linear ordering $\prec$ on the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ with $v_i \prec v_j$ for $i < j$. In the online presentation $G^{\prec}$ of the graph G, the vertices of V appear in the order determined by $\prec$.

For $V_i = \{v_1, v_2, \ldots, v_i\}$, we denote by $G^{\prec}[V_i]$ the online subgraph of $G^{\prec}$ induced by a subset V_i of the vertex set V. Note that $G^{\prec}[V_n] = G^{\prec}$ is the final graph. In other words, $G^{\prec}[V_i]$ is derived from $G^{\prec}[V_{i-1}]$ by adding the vertex v_i together with its edges incident to vertices from V_{i-1}.
To describe the online presentation of the graph, we can represent the online graph instance $G^{\prec}$ as a sequence of graphs $(G_1, G_2, \ldots, G_n)$ such that $G_i = G[V_i]$ for a prefix of the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ with an ordering given by $v_i \prec v_j$ for $i < j$, as we did in Figures 1.2 and 1.3.

Definition 1.22 (Set of Online Graphs). Let $\mathcal{G}_n$ denote the *set of all online graph instances on n vertices*. Then, $\mathcal{G}$ is the *set of all possible online graph instances* for all $n \in \mathbb{N}^+$, i. e., $\mathcal{G} = \bigcup_{n \in \mathbb{N}^+} \mathcal{G}_n$.

In this thesis, we focus on special graph classes as paths, for example. Note that we only require that the final graph $G = G[V_n]$ has the claimed graph property. In the case of paths, one can easily see that some presentation orders lead to intermediate graphs $G[V_i]$, for some time steps $i < n$, which are not paths (see Figure 1.4).
In the following figures containing online graphs, we will only depict the final graph G. The presentation order is given either by the index of the vertices v_i or by a time line as shown in Figure 1.5. If two vertices are depicted on the same time level, it does not matter which of the two vertices is shown earlier in the online presentation of the graph.
The same observation holds for the quality of the solution in intermediate time steps. We only require that the cost of the solution in the final graph has to satisfy some competitive ratio, but not that the solution has to achieve the same ratio in earlier time steps.
In some proofs, we compare different online presentations of fixed graphs. Then, we denote the vertex set of the online graph by $\{w_1, w_2, \ldots, w_n\}$ and the vertices of the online presentation by $(v_1, v_2, \ldots, v_n)$, giving the vertices w_i an order.

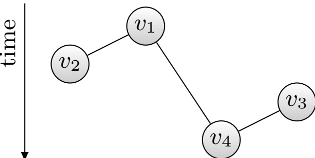


Figure 1.5. The time line emphasizes the order of the vertices in which they are revealed in the online presentation of the graph.

If it is clear from the context that an *online* graph is meant, we write just “graph” instead of “online graph”.

1.4 Online Algorithms With Advice

The main drawback in the competitive analysis of online algorithms is that an online algorithm has a huge disadvantage compared to an offline algorithm by not knowing the future parts of the input. This seems to be a rather unfair comparison since there is no way to use an offline algorithm in an online setting. The model of advice complexity of online problems has been introduced to enable a more fine-grained analysis of the hardness of online optimization problems. The idea is to measure what amount of information about the yet unknown parts of the input is necessary to compute an optimal (or near-optimal) solution online [12, 21, 25, 38]. For this, we analyze online algorithms that have access to an arbitrary prefix of an infinite tape with *advice bits* that was computed by some oracle knowing the whole input in advance.

Recall the ski-rental problem in which the algorithm (run by the tourist) has to decide in the beginning of the season if it wants to rent the skis on every sunny day, or if it should better buy the skis at the beginning of the season. Note that the option of buying the skis only makes sense in the beginning. If the tourist first rents the skis for some days and decides only later to buy them, he wastes money (assuming that the bought skis hold for the whole season and do not break down earlier). Therefore, if an oracle (a meteorologist which is never wrong) could foresee the weather for the whole season, it would compute an advice string containing only one bit, which is 0, if the algorithm should buy the skis, and 1, if it should rent the skis.

Definition 1.23 (Online Algorithm With Advice [12, 38]).
Let $I = (x_1, \ldots, x_n)$ be an input of an online optimization problem. An *online algorithm A with advice* computes the output sequence $A^{\varphi}(I) = (y_1, \ldots, y_n)$ such that y_i is computed from $\varphi, x_1, \ldots, x_i$, where φ is the content of the advice tape, i. e., an infinite binary sequence.

For the ski-rental problem, the algorithm would thus read just a single bit from the advice tape.
The computation of the algorithm A^{φ} works again step by step. In each time step i, A^{φ} reads x_i and irrevocably produces y_i using all the information read so far and possibly reading some bits of the advice string. Note that the definition does not restrict the computational power of the online algorithms. But nevertheless, all the algorithms in this thesis will be deterministic and they will all have a polynomial time complexity.
At first sight, introducing a clairvoyant oracle with unlimited computational power in order to solve an optimization problem optimally seems to be weird. There is no such oracle in real life and one also cannot implement such a powerful tool. But this new type of algorithms enables us to introduce a new measure of complexity with which the difficulties of online optimization problems can be compared with respect to the amount of advice that has to be read by every algorithm to solve the problem. So, the length of the advice string that is necessary to read measures the disadvantage we have by not knowing the future. Again, the quality of an online algorithm with advice is measured by the competitive ratio comparing the optimal (offline) solution and the solution an algorithm A^{φ} with advice computes.

Definition 1.24 (Advice Complexity). An online algorithm A for some online optimization problem is *c-competitive with advice complexity* $b(n)$ if there exists some non-negative constant α such that, for every n and for each input sequence I of length at most n, there exists some φ such that

$$\text{cost}(A^{\varphi}(I)) \leq c \cdot \text{cost}(\text{Opt}(I)) + \alpha \qquad \text{(for minimization problems)}$$
$$\text{cost}(\text{Opt}(I)) \leq c \cdot \text{cost}(A^{\varphi}(I)) + \alpha \qquad \text{(for maximization problems)}$$

and at most the first $b(n)$ bits of φ have been accessed during the computation of $A^{\varphi}(I)$.
If $\alpha = 0$, then A is called *strictly* c-competitive. A is *optimal* if it is strictly 1-competitive.

The *advice complexity* of online algorithms with advice measures how many advice bits the algorithm needs to read during its computation in order to achieve a desired competitive ratio. As usual, the advice complexity of an online optimization problem is defined as the minimum amount of advice needed by some algorithm solving the problem. We are especially interested in *lower bounds* on the advice complexity. Such lower bounds do not only tell us something about the information content [38] of online optimization problems, but they also carry over to a randomized setting where they imply lower bounds on the number of random decisions needed to compute a good solution [47].
It turns out that, for some problems, very little advice can drastically improve the competitive ratio of an online algorithm. The artificial ski-rental problem is a nice example. As we have seen above, one bit of advice is sufficient for solving this

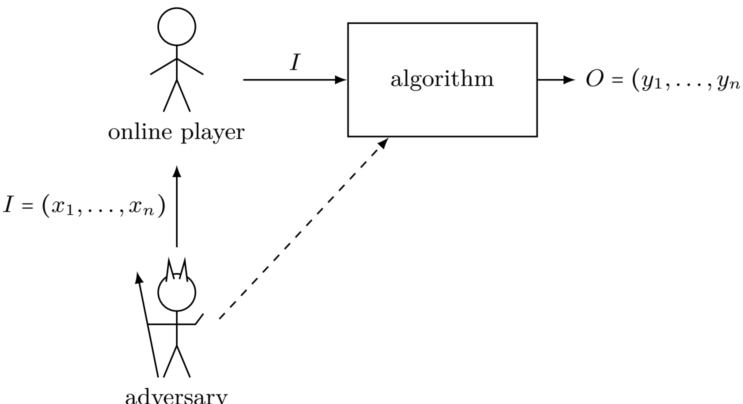


Figure 1.6. The game between the adversary and the online player. The dashed edge indicates that the adversary knows the algorithm.

problem optimally. On the other hand, no deterministic online algorithm can be better than strictly $\left(2 - \frac{1}{k}\right)$-competitive, where k is the ratio of the cost of buying and renting [39]. Also for the simple knapsack problem, i. e., when the value and weight of each item are equal, a single bit of advice is sufficient to jump from being non-competitive at all to 2-competitiveness [10]. On the other hand, many problems require a linear (or even higher) amount of advice bits for computing an optimal solution [10, 12, 25].

Plenty of more problems have already been analyzed within this model, e. g., paging [12], job shop scheduling [12, 47], the k-server problem [11], online set cover [46], string guessing [9], online independent set [19], online knapsack [10], online bin packing [15], disjoint path allocation [4], metrical task systems [25], online graph exploration [20], and online graph coloring [6, 7, 30, 58], which we discuss in more detail in Chapter 2.

The relationship between advice complexity and randomized algorithms has been discussed in [11, 25, 48].

Analyzing Online Algorithms: A Game Between an Online Player and an Adversary

Borodin and El-Yaniv described in [14] a nice view on the problem of analyzing online algorithms. One can view the problem as a game between an *online player* feeding an online algorithm with an input determined by an malicious *adversary* creating hard inputs in the sense that the competitive ratio is maximized (see Figure 1.6). The adversary knows the algorithm and therefore the behaviour on all possible instances. This knowledge allows him to pick a worst possible instance for the online player, in the sense that:

- the cost of the online algorithm running on this instance is maximized, and
- the cost of the best optimal (offline) solution is minimized.

The adversary has to try to find an instance maximizing the ratio of these two measures.
This game can be extended to online algorithms with advice (see Figure 1.7): A new entity, called the *oracle*, comes into play. The oracle creates, knowing the input instance $I = (x_1, x_2, \ldots, x_n)$ an advice tape containing advice bits that are intended to help the algorithm to solve this problem instance. This ideal world is a little bit clouded by the fact that the adversary also knows which advice string the adversary will create for a particular instance. This makes the game again harder.

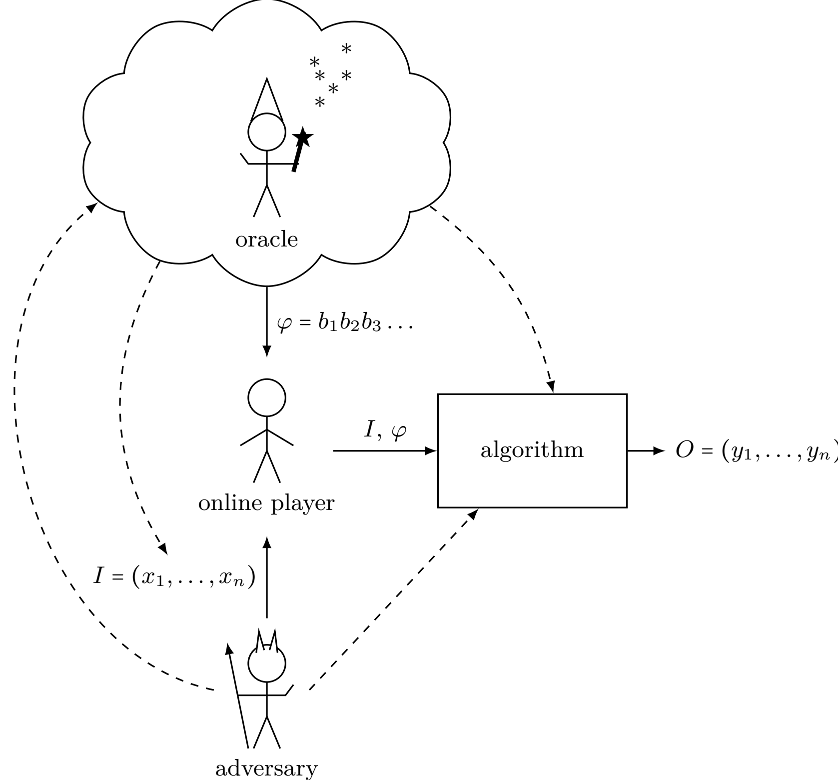


Figure 1.7. The game between the adversary and the online player who receives access to an advice tape produced by an oracle. The dashed edges indicate who sees what.

Note that the advice string is a powerful tool helping the online player. Reading m bits of advice means that the online player can choose out of a list of 2^m different, but fixed, deterministic algorithms the one that solves the instance best (see Figure 1.8 for an example of two advice bits).

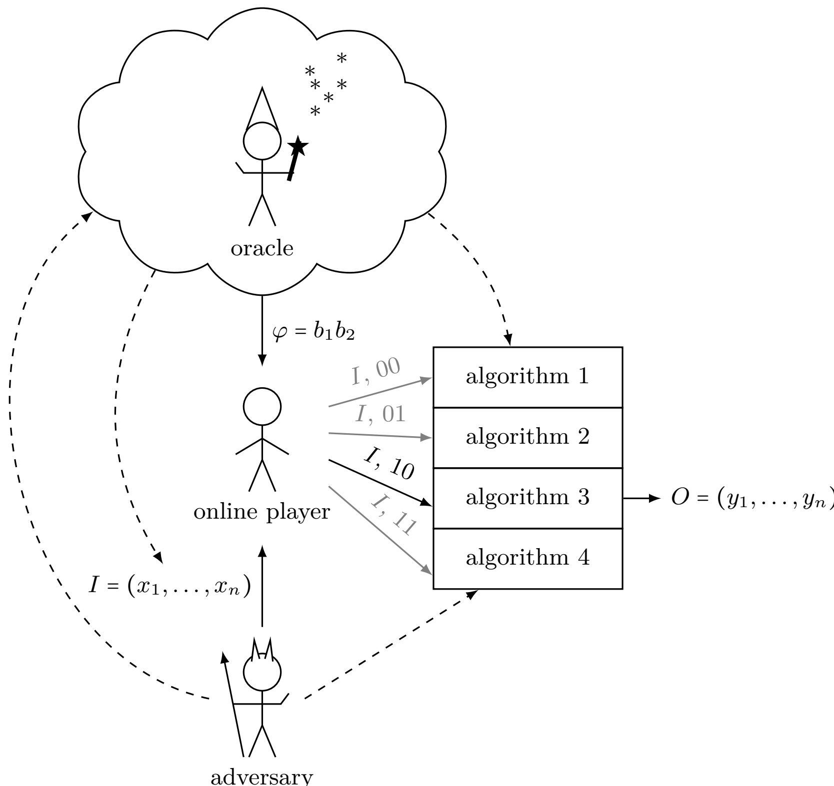


Figure 1.8. Two advice bits enable the online player to choose the best algorithm out of four for the instance he receives from the adversary.

Self-Delimiting Codes

In some scenarios, the oracle needs to encode a natural number on the advice tape. The following observations concerning this topic are taken from [45]. For a natural number smaller than n, at most $\lceil \log(n+1) \rceil$ bits are necessary to encode n. We can simplify this formula if we know that n is greater than zero, what usually happens. Therefore, the oracle can write $n-1$ to the advice tape using at most $\lceil \log(n) \rceil$ bits, and the algorithm decodes the depicted number by adding 1 to the result.

Recall that the advice tape is infinite and the algorithm does not know how long the online instance is. So, it can only process a constant number of bits per time step. But, if an algorithm is intended to read as many advice bits from the tape such that it can learn a natural number n, the oracle has to somehow delimit the number of bits on the tape. If the number of bits depends on the length of the

instance, the oracle has to transmit this to the algorithm by encoding the number n in a self-delimiting way.
For small x, i. e., $x = 1$ or $x = 2$, the first bit of the tape is 0 and the second bit encodes $x - 1$. For a number $x \in \{3, 4, \ldots, n\}$, the encoding consists of two parts:

- The last $\lceil \log(n) \rceil$ bits encode the number x, since $m = \lceil \log(x) \rceil \leq \lceil \log(n) \rceil$ holds.
- In the first part of the advice tape, the length m of the encoded number x is transmitted using $2\lceil \log(m) \rceil$ bits, in the way such that the bits of the binary representation of $m - 1$ are written on odd positions of the tape and the even positions are 0 as long as the bit right of this 0 still belongs to the binary representation of $m - 1$. A symbol 1 at an even position indicates that the binary encoding of m has finished.

The whole encoding is summarized in the following lemma.

Lemma 1.25 (Komm [45]). *For a strictly positive value $x \in \{1, \ldots, n\}$,*

$$\max\{2, \lceil \log(n) \rceil + 2\lceil \log(\lceil \log(n) \rceil) \rceil\}$$

bits are sufficient to encode the number x in a self-delimiting way. □

As an example,

$$\underbrace{1\ \mathbf{0}\ 0\ \mathbf{0}\ 1\ \mathbf{0}\ 0\ \mathbf{1}}_{\text{encoding of } \lceil \log(1418) \rceil - 1} \qquad \underbrace{1\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0}_{\text{encoding of } x - 1 = 1418}$$

encodes the value $x = 1419$. The bold bits in the first part are just there to indicate where the binary encoding of $\log(m)$ terminates.

Chapter 2

Online Coloring of Graphs

One of the most studied online scenarios is the problem of coloring a graph online. Here, the vertices of the graph are revealed one after the other, together with the edges connecting them to the already present vertices. The goal is to assign the minimum number of colors to these vertices in such a way that no two adjacent vertices get the same color. As usual in an online setting, each vertex has to be colored before the next one arrives. For an overview of results on the online graph coloring problem, see, e. g., [43, 44].

k-Colorable Graphs

It turns out that online coloring is a very hard online optimization problem for which no constant competitive ratio is possible [33]. For an overview of results on the online graph coloring problem, see, e. g., [43, 44]. In particular, some bounds on the online chromatic number of the class $\Gamma(k,n)$ of k-colorable graphs on n vertices have been proven: For all k and infinitely many n, there exists a $G \in \Gamma(k,n)$ such that any online coloring algorithm for G needs at least

$$\Omega\left(\left(\frac{\log(n)}{4k}\right)^{k-1}\right)$$

colors [64]. On the other hand, there exists an online algorithm for coloring any graph $G \in \Gamma(k,n)$ with

$$O\left(\frac{n \cdot \log^{(2k-3)}(n)}{\log^{(2k-4)}(n)}\right)$$

colors [49], where $\log^{(k)}$ is the log-function iterated k times.
In [58], the authors consider online graph coloring for all 3-colorable graphs, and for particular subsets such as chordal graphs and maximal outerplanar graphs.

For the first two graph classes, the authors show that log(3) bits per vertex are sufficient and also necessary to produce an optimal coloring online. For maximal outerplanar graphs, the authors show a lower bound of 1.0424 and an upper bound of 1.2932 advice bits per vertex. They also develop algorithms for a 4-coloring in these graph classes. 3-colorable chordal and outerplanar graphs can be solved using 0.9865 and for general 3-colorable graphs, at most 1.1583 bits per vertex has to be accessed.

Bipartite Graphs

Even for the very restricted class of bipartite, i. e., two-colorable, graphs, any online algorithm can be forced to use at least

$$\lfloor \log(n) \rfloor + 1$$

colors for coloring some bipartite graph on n vertices [5]. On the other hand, an online algorithm coloring every bipartite graph with at most

$$2\log(n)$$

colors is known [49]. In the first part of this chapter, we improve the lower bound for bipartite graphs to $\lfloor 1.13746 \cdot \log(n) - 0.49887 \rfloor$.
This chapter is organized as follows. In Section 2.1, we formally define the online coloring problem and fix our notation. In Section 2.2, we consider online algorithms without advice and present the improved lower bound on the number of necessary colors for deterministic online coloring algorithms. The proof of this lower bound is contained in Section 2.3, while Section 2.4 is devoted to the advice complexity of the online coloring of bipartite graphs. In Section 2.5, we discuss the advice complexity on paths, cycles, and spider graphs.

2.1 Preliminaries

First, we want to fix our notation and formally define the problem we are dealing with.

Definition 2.1 (Coloring). Let $G = (V, E)$ be an undirected and unweighted graph with vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set E. A *(proper) coloring of a graph G* is a function $\mathrm{col}: V \to S$ which assigns to every vertex $v_i \in V$ a color $\mathrm{col}(v_i) \in S$ and has the property that $\mathrm{col}(v_i) \neq \mathrm{col}(v_j)$, for all $i, j \in \{1, 2, \ldots, n\}$ with $\{v_i, v_j\} \in E$.

Usually, we consider the set $S = \{1, 2, \ldots, n\} \subset \mathbb{N}^+$. Let $V' \subseteq V$, then we denote by $\mathrm{col}(V')$ the set of colors assigned to the vertices in V'. To distinguish the coloring functions used by different algorithms, we denote, for an algorithm A, its coloring function by col_A.
With this, we can formally define the online coloring problem.

Definition 2.2 (Online Coloring Problem).

Input: $G^{\prec} = (G, \prec) \in \mathcal{G}$ with $G = (V, E)$ and $V = \{v_1, v_2, \dots, v_n\}$.

Output: $(c_1, c_2, \dots, c_n) \in (\mathbb{N}^+)^n$ such that $\mathrm{col}(v_i) = c_i$ and $\mathrm{col}\colon V_i \to \mathbb{N}^+$ is a coloring, for all $i \in \{1, 2, \dots, n\}$.

Cost: Number of colors used by the coloring.

Goal: Minimum.

The coloring of online graphs $G^{\prec}$ depends on the used algorithm and the ordering of the vertices. Therefore, we denote by $\mathrm{col}_{A,G^{\prec}}(V')$ the coloring function of a given algorithm A and an online graph $G^{\prec} \in \mathcal{G}_n$ for the vertex set $V' \subseteq V_n$. We omit the subscript whenever A and $G^{\prec}$ are clear from the context.
In the following, we restrict our attention to the class of bipartite graphs.

Definition 2.3 (BipCol). We denote the subproblem of the online coloring problem restricted to bipartite input graphs by BipCol.

Recall that, in a bipartite graph $G = (V, E)$, the vertex set V can be partitioned into two subsets, called *shores* and denoted by $S_1(G)$ and $S_2(G)$, with the property that the edges in E connect only vertices from different shores. We say that a color α is *common* in a bipartite graph if it appears on both shores of the bipartition.
We want to analyze BipCol, giving bounds on the number of colors used in the online coloring process. These bounds will always depend on the number n of vertices in the final graph $G^{\prec} = G^{\prec}[V_n]$. Let A be an online coloring algorithm. We denote by $F_A(G^{\prec}) = |\mathrm{col}_{A,G^{\prec}}(V_n)|$ the number of colors used by A to color the graph G. Then, $F_A(n) = \max_{G \in \mathcal{G}_n} F_A(G^{\prec})$ is the maximum number of colors A uses to color any online graph instance with n vertices in the final graph $G^{\prec}$.

Definition 2.4 (Upper Bound). We say that $U\colon \mathbb{N} \to \mathbb{N}$ is an *upper bound* on the number of colors sufficient for online coloring, if there exists an online algorithm A such that, for all $n \in \mathbb{N}$, we have $F_A(n) \leq U(n)$.

Hence, to get an upper bound U on the number of used colors, it is sufficient to find a deterministic online algorithm A coloring each graph from $\mathcal{G}_n$ using $U(n)$ colors. Similarly, a lower bound is defined.

Definition 2.5 (Lower Bound). A function L is a *lower bound* on the number of colors necessary for online coloring any graph if, for any online algorithm A, there exists an infinite subset $X \subseteq \mathbb{N}$ such that $L\colon X \to \mathbb{N}$ and, for all $n \in X$, we have $L(n) \leq F_A(n)$, i. e., if, for every algorithm A and for infinitely many n, there is an online graph $G^{\prec}_A(n) \in \mathcal{G}_n$ for which A needs at least $L(n)$ colors.

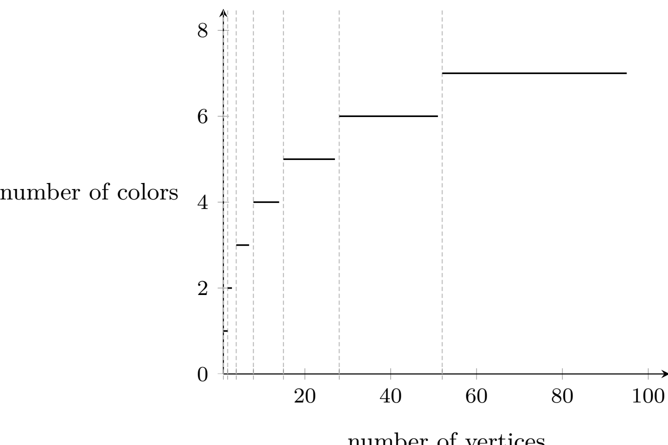


Figure 2.1. A function $\tilde{L}(n)$ for the lower bound on the number of vertices that are necessary such that every online algorithm for solving the coloring problem needs at least n colors on an instance of size $\tilde{L}(n)$.

Observe that a lower bound $L\colon X \to \mathbb{N}$ also implies a lower bound $\tilde{L}\colon \mathbb{N} \to \mathbb{N}$, where $\tilde{L}(n) = L(m)$ with $m = \max\{k \in X \mid k \leq n\}$ (see Figure 2.1).

2.2 Online Coloring Without Advice

In this section, we deal with the competitive ratio of deterministic online algorithms without advice. The following upper bound is well known. We recall the proof since we need the idea behind the algorithm in the proof later to prove Theorem 2.14.

Theorem 2.6 (Lovász, Saks, and Trotter [49]). *There is an online algorithm using at most $\lceil 2\log(n)\rceil$ colors for coloring any bipartite graph of n vertices.*

Proof. Let $G^{\prec}$ be an input instance. We describe an online algorithm A that works as follows: at each step t, consider the component $C_t(v_t)$ containing the last revealed vertex v_t. If $C_t(v_t)$ contains only v_t, i. e., if v_t was revealed isolated, A outputs $\mathrm{col}_A(v_t) = 1$, otherwise it assigns to v_t the smallest color not present on the opposite shore of this component. More formally, assuming $v_t \in S_1(C_t(v_t))$, A outputs

$$\mathrm{col}_A(v_t) = \min\{c \geq 1 \mid c \neq \mathrm{col}_A(v) \text{ for all } v \in S_2(C_t(v_t))\}.$$

By calling $B(k)$ the minimum number of vertices required for A to output color k, we have that $B(2) = 2$ and $B(3) = 4$ (see Figure 2.2). We inductively show that $B(k) \geq 2^{\frac{k}{2}}$, which implies that, on an instance of n vertices, A uses at most $2\log(n)$ colors.

If $\mathrm{col}_A(v_t) = k$ and $v_t \in S_1(C_t(v_t))$, it means that on the shore $S_2(C_t(v_t))$ all colors from 1 to $k-1$ are present. Similarly, since color $k-1$ was assigned to some vertex in $S_2(C_t(v_t))$, then on the shore $S_1(C_t(v_t))$ all colors from 1 to $k-2$ are present. Since there are two vertices $v_p \in S_1(C_t(v_t))$ and $v_q \in S_2(C_t(v_t))$ such that $\mathrm{col}_A(v_p) = \mathrm{col}_A(v_q) = k-2$, the only way A would assign that color is if v_p and v_q were on two different components of G_r, where $r = \max\{p, q\}$. By induction hypothesis, each of these components must have at least $2^{\frac{k-2}{2}}$ vertices, therefore $B(k) \geq 2 \cdot 2^{\frac{k-2}{2}} = 2^{\frac{k}{2}}$. □

There is also a well-known lower bound which even holds for trees. The idea of the construction of the instances in the following proof will also be used later to prove Lemma 2.8.

Theorem 2.7 (Bean [5]). *For every online coloring algorithm A and every $k \in \mathbb{N}^+$, there exists a tree $T_k^\prec$ on 2^{k-1} vertices such that $\mathrm{col}_A(T_k^\prec) \geq k$.*

Proof. The class of trees $T_k^\prec$ for which every deterministic online coloring algorithm A has to use at least k colors to color this tree, is built recursively.
Starting with a tree $T_1^\prec$ containing only one vertex, the adversary can force every algorithm A to use one color. In a tree $T_2^\prec$ on two vertices which are connected by an edge, every algorithm has to use two colors. Therefore, $T_2^\prec$ contains, for every online algorithm A, one color that does not appear in $T_1^\prec$.
To force a third color, the adversary introduces a new vertex which is connected to $T_1^\prec$ and the vertex with a different color in $T_2^\prec$. Therefore, every algorithm has to use a new color. Since the new vertex is only connected to one vertex in each subtree, there cannot arise a cycle.
In general, the adversary designs the tree $T_k^\prec$ recursively, using all the trees $T_1^\prec$ to $T_{k-1}^\prec$ such that a new vertex that is connected to one vertex in each subtree such that the new vertex has neighbors with $k-1$ different colors. This is always possible, since T_{i+1} has always at least one color which is not present in T_i, for all $i \in \{1, 2, \ldots, k-1\}$. Therefore, the new vertex receives a kth color for every algorithm A. In Figure 2.2, some base cases and the general construction are shown.

Due to the recursive construction, the number of vertices in $T_k^\prec$ is given by

$$|V(T_k^\prec)| = 1 + \sum_{i=1}^{k-1} |V(T_i^\prec)| .$$

We show that $|V(T_k^\prec)| = 2^{k-1}$. Let $t_k = |V(T_k^\prec)|$ be the number of vertices in the online tree $T_k^\prec$ constructed as shown above. Subtracting the equation $t_{k-1} = 1 + \sum_{i=1}^{k-2} t_i$ from $t_k = 1 + \sum_{i=1}^{k-1} t_i$ yields

$$t_k - t_{k-1} = t_{k-1} ,$$

and therefore, we get the recursion $t_k = 2t_{k-1}$. The starting condition $t_1 = 1$ immediately leads to the claimed 2^{k-1} vertices in a tree $T_k^\prec$. □

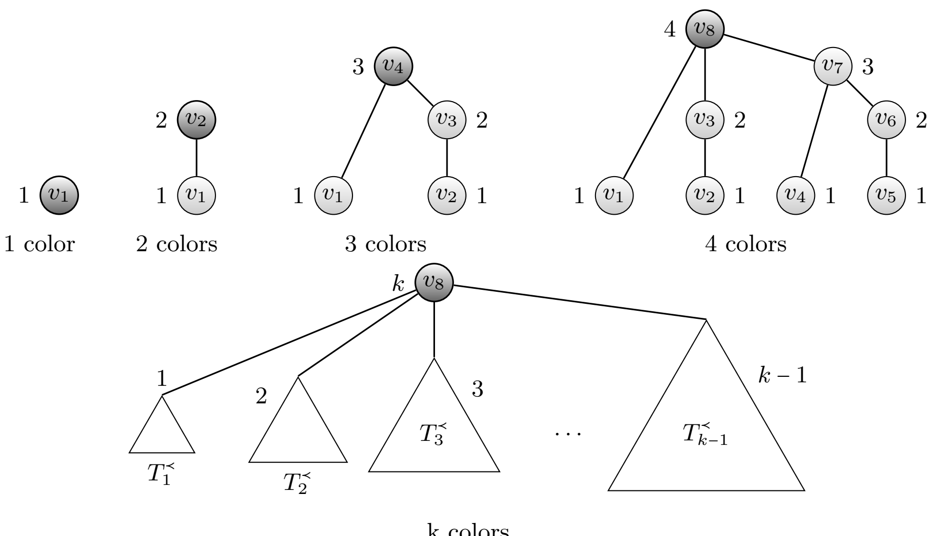


Figure 2.2. The construction of the trees $T_k^\prec$ in Theorem 2.7 which contain at least k colors, independent of the chosen deterministic online coloring algorithm A. To build a tree $T_k^\prec$, a vertex is connected to one vertex of each tree $T_i^\prec$ for all $i < k$. Each of these trees $T_{i+1}^\prec$ contains at least one color not contained in the smaller tree $T_i^\prec$.

Theorem 2.7 immediately implies that there exists an infinite number of trees (and thus of bipartite graphs) forcing any online algorithm to use at least $\lceil \log(n) \rceil + 1$ colors on any graph on n vertices from this class. In the remainder of this section, we improve on this result by describing a graph class, which forces every coloring algorithm A to use even more colors on infinitely many graphs of the class. This class is built recursively. In the proof, we will focus, for a fixed deterministic online coloring algorithm A, only on those $G^\prec[V_i]$'s in an instance $G^\prec \in \mathcal{G}_n$ in which the new vertex v_i gets a new color with respect to the previous graph $G^\prec[V_{i-1}]$.

Lemma 2.8. *For every $k \in \mathbb{N}^+$ and every online coloring algorithm A, there exists an online graph $G_A^\prec(k)$ such that:*

1. $F_A(G_A^\prec(k)) \geq k$,
2. $F_A(S_1(G_A^\prec(k))) \geq k - 2$,
3. $F_A(S_2(G_A^\prec(k))) \geq k - 1$,
4. $|V(G_A^\prec(k))| \leq W(k) := W(k-1) + W(k-2) + W(k-3) + 1$, *for* $k \geq 3$, *and* $W(0) = 0$, $W(1) = 1$, *and* $W(2) = 2$.

Consequently, $W(k)$ is the maximum number of vertices that a graph needs in order to force an arbitrary algorithm A to use at least k colors.

We will prove Lemma 2.8 in the following section. The recurrence given by property 4 of Lemma 2.8 can be resolved as follows.

Lemma 2.9. *Let $W(k)$ be defined as in Lemma 2.8. Then,*

$$W(k) \le 1.35527 \cdot 1.83929^k - 0.400611\,.$$

Proof. It can be easily shown by induction that $W(k) = \sum_{n=0}^{k+1} T(n)$, where $T(n)$ is the n-th *Tribonacci number* (see Definition 1.1) defined as

$$T(n) = T(n-1) + T(n-2) + T(n-3)$$

for an integer $n > 2$ and $T(0) = T(1) = 0$, $T(2) = 1$.

Base cases: We need to show that the equality holds for $k = 0, 1, 2$:

$$0 = W(0) = \sum_{n=0}^{1} T(n) = T(0) + T(1) = 0 + 0 \qquad \checkmark$$

$$1 = W(1) = \sum_{n=0}^{2} T(n) = T(0) + T(1) + T(2) = 0 + 0 + 1 = 1 \qquad \checkmark$$

$$2 = W(2) = \sum_{n=0}^{3} T(n) = T(0) + T(1) + T(2) + T(3) = 0 + 0 + 1 + 1 = 2 \qquad \checkmark$$

Inductive step: Assume that $W(i) = \sum_{n=0}^{i+1} T(n)$ holds for all $i < k$. Then, we can compute $W(k)$ as follows.

$$\begin{aligned}
W(k) &= W(k-1) + W(k-2) + W(k-3) + 1 \\
&= \sum_{n=0}^{k} T(n) + \sum_{n=0}^{k-1} T(n) + \sum_{n=0}^{k-2} T(n) + 1 \\
&= \underbrace{\big(T(k) + T(k-1) + T(k-2)\big)}_{T(k+1)} + \underbrace{\big(T(k-1) + T(k-2) + T(k-3)\big)}_{T(k)} \\
&\quad + \ldots + \underbrace{\big(T(2) + T(1) + T(0)\big)}_{T(3)} + T(1) + T(0) + \underbrace{T(0)}_{=0} + \underbrace{1}_{=T(2)} \\
&= T(k+1) + T(k) + \ldots + T(3) + T(2) + T(1) + T(0) \\
&= \sum_{n=0}^{k+1} T(n)
\end{aligned}$$

The number $T(n)$ can be computed as follows (see Equation 1.1 of Chapter 1):

$$T(n) = 3b \cdot \frac{\left(\frac{1}{3}(a_+ + a_- + 1)\right)^n}{b^2 - 2b + 4} \le 0.336229 \cdot 1.83929^n,$$

where $a_+ = (19 + 3\sqrt{33})^{\frac{1}{3}}$, $a_- = (19 - 3\sqrt{33})^{\frac{1}{3}}$, and $b = (586 + 102\sqrt{33})^{\frac{1}{3}}$. Summing up the values of $T(n)$ for $n \in \{0, \ldots, k+1\}$ gives the claimed result. □

Theorem 2.10. *For any online coloring algorithm A, there exists an infinite sequence of online graphs $G_A^{\prec}(k) \in \mathcal{G}_{n_k}$ with $n_k < n_{k+1}$ for all $k \in \mathbb{N}$ such that A needs at least*

$$\lfloor 1.13746 \cdot \log(n_k) - 0.49887 \rfloor$$

colors to color $G_A^{\prec}(k)$.

Proof. The claim follows immediately from Lemma 2.8 and Lemma 2.9 by resolving the following inequality for k:

$$n_k \le 1.35527 \cdot 1.83929^k .$$

Taking the logarithm on both sides yields

$$\log(n_k) \le \log(1.35527) + k \cdot \log(1.83929) ,$$

and resolving for k gives

$$k \ge \frac{\log(n_k) - \log(1.35527)}{\log(1.83929)} = 1.13746 \cdot \log(n_k) - 0.49887 .$$

□

2.3 Proof of Lemma 2.8

In this section, we prove Lemma 2.8. We proceed by an induction over k, the number of colors. For every k, we generate a class $\tilde{\mathcal{G}}(k)$ consisting of online graphs defined as

$$\tilde{\mathcal{G}}(k) = \{G_B^{\prec}(k) \mid B \text{ is an online coloring algorithm and properties 1 to 4 of Lemma 2.8 are satisfied}\} .$$

Hence, for a fixed k, we will find in $\tilde{\mathcal{G}}(k)$, for every online coloring algorithm B, an instance $G_B^{\prec}(k)$ that forces B to use at least k colors to color $G_B^{\prec}(k)$. Those instances are built inductively. We will prove that we can construct, for any online coloring algorithm A, a hard instance $G_A^{\prec}(k)$, using three online graphs $G_{k-1}^{\prec} \in \tilde{\mathcal{G}}(k-1)$, $G_{k-2}^{\prec} \in \tilde{\mathcal{G}}(k-2)$, $G_{k-3}^{\prec} \in \tilde{\mathcal{G}}(k-3)$, that are revealed in this order, and an additional vertex v (see Figure 2.3).
Let $H(k)$ be the **induction hypothesis**, formulated as follows:

H(k): For all $j \le k$ and all online algorithms B, there exists a graph $G_B^{\prec}(j) \in \tilde{\mathcal{G}}(j)$.

Assuming $H(k-1)$ holds, the hypothesis states that, for every online algorithm A, a graph $G_{k-1}^{\prec} = G_A^{\prec}(k-1) \in \tilde{\mathcal{G}}(k-1)$ satisfying all the properties of Lemma 2.8 exists. To show the existence of the second and third constructed subgraph, $G_{k-2}^{\prec} \in \tilde{\mathcal{G}}(k-2)$

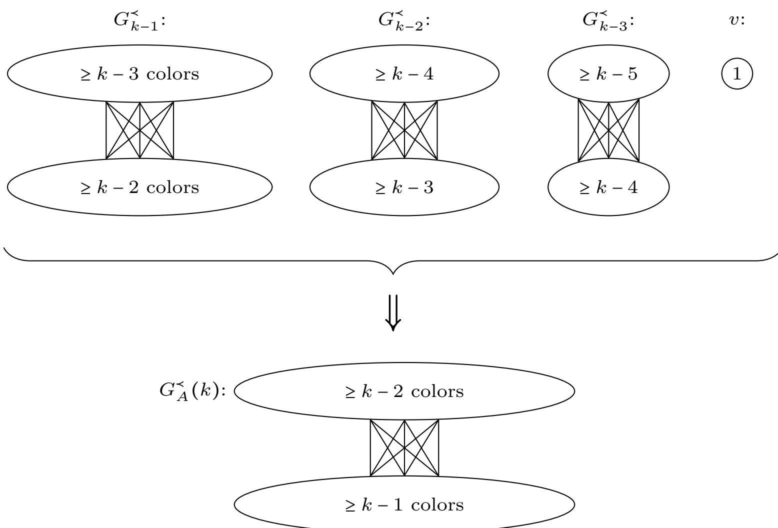


Figure 2.3. Using three online graphs $G^{\prec}_{k-1} \in \tilde{\mathcal{G}}(k-1)$, $G^{\prec}_{k-2} \in \tilde{\mathcal{G}}(k-2)$, $G^{\prec}_{k-3} \in \tilde{\mathcal{G}}(k-3)$ and a vertex v, we can construct, for any online algorithm A, a new online graph $G^{\prec}_{A}(k) \in \tilde{\mathcal{G}}(k)$.

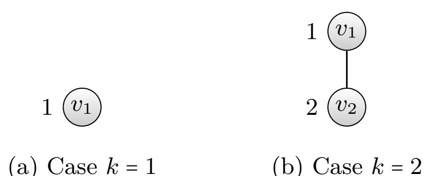
and $G^{\prec}_{k-3} \in \tilde{\mathcal{G}}(k-3)$, we have to take into account that the algorithm A already knows a part of the instance, and hence it may behave differently from the case where there is no part known.
We merge the shores of $G^{\prec}_{k-1}$, $G^{\prec}_{k-2}$, and $G^{\prec}_{k-3}$ in an appropriate way and, using an additional vertex v, we ensure that the resulting graph $G^{\prec}_{A}(k)$ is in $\tilde{\mathcal{G}}(k)$. In some cases, we do not need all four components to guarantee that all four properties of Lemma 2.8 are satisfied.
We merge two online graph instances $G^{\prec}[V] = (G, \prec) \in \mathcal{G}$ with $V = \{v_1, v_2, \ldots, v_n\}$ and $\overline{G}^{\prec}[V'] = (G', \prec) \in \mathcal{G}$ with $V' = \{v'_1, v'_2, \ldots, v'_m\}$ to an instance $M^{\prec}[V''] = G^{\prec}[V] \circ \overline{G}^{\prec}[V']$, defined as

$$M^{\prec}[V''] = (G \cup G', \prec) \in \mathcal{G}\,,$$

where, for two graphs $G = (V, E)$ and $G' = (V', E')$ with $V \cap V' = \varnothing$, $G \cup G'$ is defined as the graph $(V'' = \{v_1, \ldots, v_n, v'_1, \ldots, v'_m\}, E \cup E')$ and $v_n \prec v'_1$.

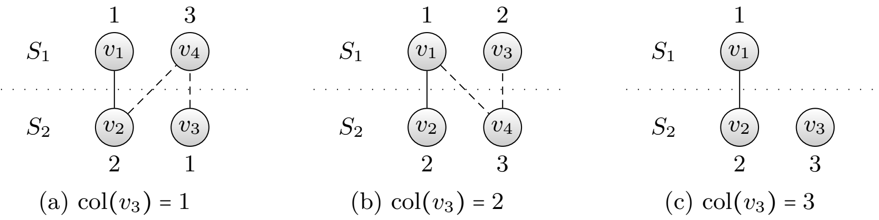
Base Cases ($k \leq 3$)

For $k \in \{0, 1, 2\}$, it is easy to see that the hypothesis $H(k)$ is satisfied (see Figure 2.4). In case $k = 3$, for every online coloring algorithm A, $G^{\prec}_{A}(3)$ can be constructed recursively using two graphs $G^{\prec}_{2} \in \tilde{\mathcal{G}}(2)$, $G^{\prec}_{1} \in \tilde{\mathcal{G}}(1)$, and possibly a new vertex. The vertices of $G^{\prec}_{2} = G^{\prec}_{A}(3)[\{v_1, v_2\}]$ are colored, w.l.o.g., with 1 and 2, and $G^{\prec}_{1} = G^{\prec}_{A}(3)[\{v_3\}]$ can be colored, w.l.o.g., with 1, 2 or 3 (see Figure 2.5).



(a) Case $k = 1$ (b) Case $k = 2$

Figure 2.4. Base cases: W.l.o.g., the vertices are colored as indicated. The indices of the vertices indicate their order of appearance.



(a) $\mathrm{col}(v_3) = 1$ (b) $\mathrm{col}(v_3) = 2$ (c) $\mathrm{col}(v_3) = 3$

Figure 2.5. Case $k = 3$: For any behaviour of an algorithm A, we can construct the graph $G_A^{\prec}(3)$ using graphs $G_2^{\prec} = G_A^{\prec}(3)[\{v_1, v_2\}] \in \tilde{\mathcal{G}}(2)$ and $G_1^{\prec} = G_A^{\prec}(3)[\{v_3\}] \in \tilde{\mathcal{G}}(1)$ that force A to use a third color.

If the algorithm colors v_1, v_2, and v_3 with different colors, as shown in case (c) of Figure 2.5, obviously all properties of $H(3)$ are already satisfied. Otherwise, we have to add one new vertex v_4, which is connected to two vertices with different colors to force every online coloring algorithm A to use a third color (see (a) and (b) of Figure 2.5).

Inductive Step ($k \geq 4$)

For every online algorithm A and every $k \in \mathbb{N}^+$, we construct $G_A^{\prec}(k)$ in four steps using three graphs $G_{k-1}^{\prec} \in \tilde{\mathcal{G}}(k-1)$, $G_{k-2}^{\prec} \in \tilde{\mathcal{G}}(k-2)$, $G_{k-3}^{\prec} \in \tilde{\mathcal{G}}(k-3)$, and an additional vertex v.
First, assuming that $H(k-1)$ holds, we show that, for every algorithm A, we can construct three graphs $G_{k-1}^{\prec}$, $G_{k-2}^{\prec}$, and $G_{k-3}^{\prec}$ in this order satisfying all the properties of Lemma 2.8. Then, we show that we can merge them, using an additional vertex v, to a graph $G_A^{\prec}(k) \in \tilde{\mathcal{G}}(k)$.

Existence of the graphs $G_{k-1}^{\prec}$, $G_{k-2}^{\prec}$, and $G_{k-3}^{\prec}$

We assume that $H(k-1)$ holds. Hence, for every online coloring algorithm B and $j \in \{k-1, k-2, k-3\}$, there exists a graph $G_B^{\prec}(j) \in \tilde{\mathcal{G}}(j)$.

Step 1: Because of $H(k-1)$, we know that a graph $G_{k-1}^{\prec} = G_A^{\prec}(k-1)$ exists.

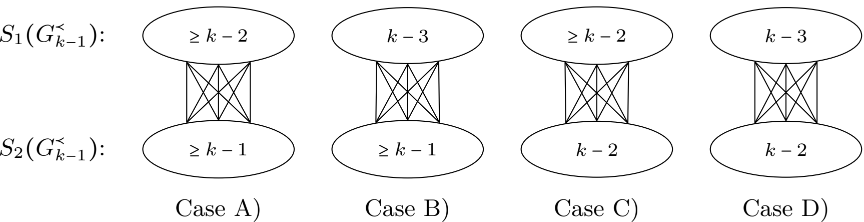


Figure 2.6. There are four cases how the colors in $G^{\prec}_{k-1}$ can be distributed.

Step 2: In the next phase, algorithm A receives a second subgraph. We cannot simply use the graph $G^{\prec}_{A}(k-2)$ here, whose existence is guaranteed by $H(k-2)$, since $H(k-2)$ guarantees the hardness of this input only in the case that A reads it from its initial configuration. Having already read $G^{\prec}_{k-1}$, A might behave differently on $G^{\prec}_{A}(k-2)$. We denote by $A|_{G^{\prec}_{k-1}}$ the work of algorithm A having already processed the graph $G^{\prec}_{k-1}$. $A|_{G^{\prec}_{k-1}}$ can be simulated by an algorithm B, which does the same work as $A|_{G^{\prec}_{k-1}}$ but which did not receive any other graph before. In other words, if we think of the algorithms as Turing machines, B uses the same transitions as A, but its initial configuration (state and tape content) is the same as the configuration reached by A after processing $G^{\prec}_{k-1}$. Because of $H(k-1)$, and thus $H(k-2)$, we know that, for such an algorithm B, there is a graph $G^{\prec}_{k-2} = G^{\prec}_{B}(k-2) = G^{\prec}_{A|_{G^{\prec}_{k-1}}}(k-2) \in \tilde{\mathcal{G}}(k-2)$.

Step 3: Now, algorithm A gets a third subgraph. Again, the work of $A|_{G^{\prec}_{k-1} \circ G^{\prec}_{k-2}}$ can be simulated by an algorithm C. Because of the induction hypothesis, a graph $G^{\prec}_{k-3} = G^{\prec}_{C}(k-3) = G^{\prec}_{A|_{G^{\prec}_{k-1} \circ G^{\prec}_{k-2}}}(k-3)$ exists.

Hence, we have the graphs $G^{\prec}_{k-1}$, $G^{\prec}_{k-2}$, and $G^{\prec}_{k-3}$ at our disposal and can force a new color, possibly with the help of an additional vertex v, as follows.

Construction of graph $G^{\prec}_{A}(k)$

The graphs $G^{\prec}_{k-1}$, $G^{\prec}_{k-2}$, and $G^{\prec}_{k-3}$ are presented in this order. Beginning with graph $G^{\prec}_{k-1}$, we distinguish four possible cases for an online algorithm A (see Figure 2.6):

A) A uses, w.l.o.g., $\geq k-2$ colors on $S_1(G^{\prec}_{k-1})$ and $\geq k-1$ colors on $S_2(G^{\prec}_{k-1})$.

B) A uses, w.l.o.g., $k-3$ colors on $S_1(G^{\prec}_{k-1})$ and $\geq k-1$ colors on $S_2(G^{\prec}_{k-1})$.

C) A uses, w.l.o.g., $\geq k-2$ colors on $S_1(G^{\prec}_{k-1})$ and $k-2$ colors on $S_2(G^{\prec}_{k-1})$.

D) A uses $k-3$ colors on $S_1(G^{\prec}_{k-1})$ and $k-2$ colors on $S_2(G^{\prec}_{k-1})$.

Since $H(k-1)$ holds, graph $G^{\prec}_{k-1}$ is colored by algorithm A with at least $k-1$ colors, and $S_1(G^{\prec}_{k-1})$ contains at least $k-3$ and $S_2(G^{\prec}_{k-1})$ at least $k-2$ colors. Therefore, after constructing $G^{\prec}_{k-1}$, the algorithm A encounters one of the four cases above. To finish the construction, we have to ensure that the final graph contains at least k colors. If this is not yet the case for $G^{\prec}_{k-1}$, we need to force the algorithm to use a new color k.
We will show that, in order to satisfy the properties 2 and 3 of Lemma 2.8, either the graphs $G^{\prec}_{k-2}$ and $G^{\prec}_{k-3}$ contain some of the colors that appear only on one shore of $G^{\prec}_{k-1}$, or there are enough additional colors in $G^{\prec}_{k-2}$ and $G^{\prec}_{k-3}$, which do not appear in $G^{\prec}_{k-1}$. Only in some cases will we need all three graphs $G^{\prec}_{k-1}$, $G^{\prec}_{k-2}$, $G^{\prec}_{k-3}$ and the vertex v. In many cases, a subset of those graphs is sufficient to construct a graph $G^{\prec}_{A}(k) \in \tilde{\mathcal{G}}(k)$.

A) $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| \geq k-2$ **and** $|\mathrm{col}(S_2(G^{\prec}_{k-1}))| \geq k-1$

If the graph $G^{\prec}_{k-1}$ contains at least k colors, properties 1, 2, and 3 of Lemma 2.8 are satisfied. Furthermore, we have

$$|V(G^{\prec}_{A}(k))| = |V(G^{\prec}_{k-1})| \leq W(k-1) \leq W(k-1) + W(k-2) + W(k-3) + 1 = W(k)\,.$$

Hence, all the properties of Lemma 2.8 are satisfied and we can finish the construction without using additional subgraphs.
Now, assume $G^{\prec}_{k-1}$ contains only $k-1$ colors. To satisfy property 1, we need to force every algorithm to use one more color. Connecting an additional vertex v to all vertices in $S_2(G^{\prec}_{k-1})$, and thus adding it to $S_1(G^{\prec}_{k-1})$, forces the algorithm to use color k:

$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup \{v\})$: $1\ 2\ \cdots\ k-2$ — k

$\mathrm{col}(S_2(G^{\prec}_{k-1}))$: $1\ 2\ \cdots\ k-2\ \ k-1$

We have

$$\begin{aligned}|V(G^{\prec}_{A}(k))| &= |V(G^{\prec}_{k-1})| + 1 \leq W(k-1) + 1\\ &\leq W(k-1) + W(k-2) + W(k-3) + 1 = W(k)\end{aligned}$$

and therefore all properties of Lemma 2.8 are satisfied.

B) $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| = k-3$ **and** $|\mathrm{col}(S_2(G^{\prec}_{k-1}))| \geq k-1$

Because we assume that the induction hypothesis holds, $G^{\prec}_{k-2}$ is colored by algorithm A with at least $k-2$ colors. Therefore, there exists, w.l.o.g., a color $a \in \mathrm{col}(S_1(G^{\prec}_{k-2}))$ such that $a \notin \mathrm{col}(S_1(G^{\prec}_{k-1}))$. Hence, merging $G^{\prec}_{k-1}$ and $G^{\prec}_{k-2}$ such that color a is added to shore S_1, we obtain an analogous situation as in case A):

$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2}))$: $1\ 2\ \cdots\ k-3\ \ a$

$\mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2}))$: $1\ 2\ \cdots\ k-2\ \ k-1$

C) $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| \geq k-2$ **and** $|\mathrm{col}(S_2(G^{\prec}_{k-1}))| = k-2$

If $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| \geq k-1$ holds, we can swap the shores and obtain case A). Therefore, we can assume that $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| = k-2$.
Because $G^{\prec}_{k-1}$ is colored with at least $k-1$ colors, its shores can have at most $k-3$ common colors:

$$\mathrm{col}(S_1(G^{\prec}_{k-1})) : \boxed{1\ 2\ \cdots\ k-3\ \ b}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1})) : \boxed{1\ 2\ \cdots\ k-3\ \ a}$$

Hence, $G^{\prec}_{k-2}$ has to contain either color a, b or a new color $c \notin \mathrm{col}(G^{\prec}_{k-1})$, w.l.o.g. $b \in \mathrm{col}(S_1(G^{\prec}_{k-2}))$ or $c \in \mathrm{col}(S_1(G^{\prec}_{k-2}))$. Merging $G^{\prec}_{k-1}$ and $G^{\prec}_{k-2}$ in an appropriate way leads again to a situation as in case A):

$$\begin{array}{lll} \mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2})) : \boxed{1\ 2\ \cdots\ k-3\ \ b} & & \boxed{1\ 2\ \cdots\ k-3\ \ b} \\ \mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2})) : \boxed{1\ 2\ \cdots\ k-3\ \ a}\boxed{b} & \text{or} & \boxed{1\ 2\ \cdots\ k-3\ \ a}\boxed{c} \end{array}$$

D) $|\mathrm{col}(S_1(G^{\prec}_{k-1}))| = k-3$ **and** $|\mathrm{col}(S_2(G^{\prec}_{k-1}))| = k-2$

The shores of $G^{\prec}_{k-1}$ contain at most $k-4$ common colors since we have to have at least $k-1$ colors in total:

$$\mathrm{col}(S_1(G^{\prec}_{k-1})) : \boxed{1\ 2\ \cdots\ k-4\ \ c}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1})) : \boxed{1\ 2\ \cdots\ k-4\ \ a\ b}$$

To satisfy all properties of Lemma 2.8, we need either one additional color on each shore or two additional colors in $S_1(G^{\prec}_{k-1})$. And we have to force a new color k for the construction of graph $G^{\prec}_A(k)$. We distinguish five cases according to the sets of colors present in $G^{\prec}_{k-2}$ and $G^{\prec}_{k-3}$:

1. **There are $d \in \mathrm{col}(G^{\prec}_{k-2})$ and $e \in \mathrm{col}(G^{\prec}_{k-3})$ with $d, e \notin \mathrm{col}(G^{\prec}_{k-1})$:** W.l.o.g., $d \in \mathrm{col}(S_1(G^{\prec}_{k-2}))$ and $e \in \mathrm{col}(S_1(G^{\prec}_{k-3}))$. Then the following combination of the shores leads to $G^{\prec}_A(k) \in \tilde{\mathcal{G}}(k)$:

$$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2}) \cup S_2(G^{\prec}_{k-3})) : \boxed{1\ 2\ \cdots\ k-4\ \ c}\boxed{d}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2}) \cup S_1(G^{\prec}_{k-3})) : \boxed{1\ 2\ \cdots\ k-4\ \ a\ b}\boxed{e}$$

The number of colors in $S_1(G^{\prec}_A(k))$ is larger than $k-2$ and in $S_2(G^{\prec}_A(k))$ it is larger than $k-1$. The total number of colors is at least k (in the case of $d \neq e$ we have at least $k+1$ colors). And with

$$\begin{aligned} |V(G^{\prec}_A(k))| &= |V(G^{\prec}_{k-1})| + |V(G^{\prec}_{k-2})| + |V(G^{\prec}_{k-3})| \\ &\leq W(k-1) + W(k-2) + W(k-3) \leq W(k)\,, \end{aligned}$$

we have $G^{\prec}_A(k) \in \tilde{\mathcal{G}}(k)$.

2. **col $(G^{\prec}_{k-2}) \subseteq \mathrm{col}(G^{\prec}_{k-1})$ and one new color $e \in \mathrm{col}(G^{\prec}_{k-3}) \setminus \mathrm{col}(G^{\prec}_{k-1})$:** Since $G^{\prec}_{k-2}$ contains at least $k-2$ colors and it is colored with a subset of colors of $G^{\prec}_{k-1}$, it has to contain at least one color from the set $\{a, b\}$, say $a \in S_1(G^{\prec}_{k-2})$ and $e \in \mathrm{col}(S_1(G^{\prec}_{k-3}))$. Then we get

$$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2}) \cup S_2(G^{\prec}_{k-3})): \quad \boxed{1\ 2\ \cdots\ k-4\ \ c}\ \boxed{a}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2}) \cup S_1(G^{\prec}_{k-3})): \quad \boxed{1\ 2\ \cdots\ k-4\ \ a\ b}\ \boxed{e}$$

   Then $G^{\prec}_A(k) \in \tilde{\mathcal{G}}(k)$ since

$$\begin{aligned}|V(G^{\prec}_A(k))| &= |V(G^{\prec}_{k-1})| + |V(G^{\prec}_{k-2})| + |V(G^{\prec}_{k-3})| \\ &\leq W(k-1) + W(k-2) + W(k-3) \leq W(k)\,.\end{aligned}$$

3. **col $(G^{\prec}_{k-3}) \subseteq \mathrm{col}(G^{\prec}_{k-1})$ and one new color $d \in \mathrm{col}\,(G^{\prec}_{k-2}) \setminus \mathrm{col}(G^{\prec}_{k-1})$:** With the same argumentation as above, we conclude that $G^{\prec}_{k-3}$ must contain at least one of the colors $\{a, b, c\}$. Then, we have the same situation as in the case before (in the case of $c \in \mathrm{col}(G^{\prec}_{k-3})$, the merging of the shores is reversed).

In the following cases, we can assume that

$$\mathrm{col}(G^{\prec}_{k-2}) \subseteq \mathrm{col}(G^{\prec}_{k-1}) \text{ and } \mathrm{col}(G^{\prec}_{k-3}) \subseteq \mathrm{col}(G^{\prec}_{k-1})\,.$$

Because $G^{\prec}_{k-3}$ is colored with at least $k-3$ colors, it contains one of the three colors $\{a, b, c\}$. Analogously, $G^{\prec}_{k-2}$ contains two of the three colors $\{a, b, c\}$. Then, either c appears in one of the graphs $G^{\prec}_{k-2}$ or $G^{\prec}_{k-3}$, or one of the colors $\{a, b\}$ is in $G^{\prec}_{k-3}$ and both a and b are in $G^{\prec}_{k-2}$:

4. **$c \in \mathrm{col}\,(G^{\prec}_{k-2}) \cup \mathrm{col}(G^{\prec}_{k-3})$:** W.l.o.g. , assume $c \in \mathrm{col}(S_1(G^{\prec}_{k-2}))$. The other cases are analogous.

$$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2})): \quad \boxed{1\ 2\ \cdots\ k-4\ \ c}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2})): \quad \boxed{1\ 2\ \cdots\ k-4\ \ a\ b}\ \boxed{c}$$

   $S_1(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2})$ contains at least $k-3$ colors and $S_2(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2})$ consists of at least $k-1$ colors. The k-th color, say color k, can be forced by adding a new vertex v, which is connected to all vertices in $S_2(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2})$. Then, the resulting graph $G^{\prec}_A(k)$ will look as follows:

$$\mathrm{col}(S_1(G^{\prec}_{k-1}) \cup S_2(G^{\prec}_{k-2}) \cup \{v\}): \quad \boxed{1\ 2\ \cdots\ k-4\ \ c} \quad \text{(}k\text{)}$$
$$\mathrm{col}(S_2(G^{\prec}_{k-1}) \cup S_1(G^{\prec}_{k-2})): \quad \boxed{1\ 2\ \cdots\ k-4\ \ a\ b}\ \boxed{c}$$

Hence, we have $G_A^{\prec}(k) \in \tilde{\mathcal{G}}(k)$ because $S_1(G_A^{\prec}(k))$ contains $k-2$ colors, $S_2(G_A^{\prec}(k))$ is colored with at least $k-1$ colors, and

$$\begin{aligned} |V(G_A^{\prec}(k))| &= |V(G_{k-1}^{\prec})| + |V(G_{k-2}^{\prec})| + 1 \\ &\leq W(k-1) + W(k-2) + 1 \leq W(k)\,. \end{aligned}$$

5. **$a, b \in \mathrm{col}(G_{k-2}^{\prec})$ and one of those colors is in $G_{k-3}^{\prec}$:**
W.l.o.g., let $a \in \mathrm{col}(S_1(G_{k-2}^{\prec}))$ and $b \in \mathrm{col}(S_1(G_{k-3}^{\prec}))$. The shores can be matched as follows:

$$\mathrm{col}(S_1(G_{k-1}^{\prec}) \cup S_1(G_{k-2}^{\prec}) \cup S_1(G_{k-3}^{\prec})) : \boxed{1\ 2\ \cdots\ k-4\ \ c}\ \boxed{a}\ \boxed{b}$$

$$\mathrm{col}(S_2(G_{k-1}^{\prec}) \cup S_2(G_{k-2}^{\prec}) \cup S_2(G_{k-3}^{\prec})) : \boxed{1\ 2\ \cdots\ k-4\ \ a\ \ b}$$

We can force an algorithm to use color k by introducing a new vertex v that is connected to all vertices in $S_1(G_{k-1}^{\prec}) \cup S_1(G_{k-2}^{\prec}) \cup S_1(G_{k-3}^{\prec})$:

$$\mathrm{col}(S_1(G_{k-1}^{\prec}) \cup S_1(G_{k-2}^{\prec}) \cup S_1(G_{k-3}^{\prec})) : \boxed{1\ 2\ \cdots\ k-4\ \ c}\ \boxed{a}\ \boxed{b}$$

$$\mathrm{col}(S_2(G_{k-1}^{\prec}) \cup S_2(G_{k-2}^{\prec}) \cup S_2(G_{k-3}^{\prec}) \cup \{v\}) : \boxed{1\ 2\ \cdots\ k-4\ \ a\ \ b}\quad \text{(k)}$$

Now, both shores contain at least $k-1$ colors and hence $G_A^{\prec}(k) \in \tilde{\mathcal{G}}(k)$ because

$$\begin{aligned} |V(G_A^{\prec}(k))| &= |V(G_{k-1}^{\prec})| + |V(G_{k-2}^{\prec})| + |V(G_{k-3}^{\prec})| + 1 \\ &\leq W(k-1) + W(k-2) + W(k-3) + 1 = W(k)\,. \end{aligned}$$

Note that this is the hardest case, leading to exactly the recurrence from property 4 in Lemma 2.8.

□

2.4 Advice Complexity in Bipartite Graphs

Now we investigate the advice complexity of the online coloring problem on bipartite graphs. This section emerged from a joint work with Maria Paola Bianchi.
We observe first that, whenever a graph is 2-colorable, the algorithm does not need to read any advice bit for non-isolated vertices in the online presentation of the graph.

Lemma 2.11. *Let $\mathcal{H}_n$ be the set of all online graph instances such that each offline graph H_n underlying the online graph instance $H^{\prec} = (H_n, \prec)$ is 2-colorable. An online algorithm with advice for the problem of coloring $H^{\prec}$ optimally does not need to read advice bits whenever the degree of the currently processed vertex v_i is positive in the graph $H_n[V_i]$ that is present in time step i.*

Proof. If the degree of v_i in $H_n[V_i]$ is positive, this vertex must be adjacent to some vertex v_j for some $j < i$. The online algorithm already assigned a color to v_j, and now it must use the other color for v_i. This can be done without advice. □

In this section, we restrict our attention to strict competitiveness. For simplicity, we call a strictly c-competitive algorithm c-competitive here.
We start with giving an upper bound on the amount of advice needed for achieving an optimal coloring.

Theorem 2.12. *There exists an online algorithm for* BipCol*, which uses at most $n - 2$ advice bits to be optimal on every instance on n vertices.*

Proof. Due to Lemma 2.11, we give advice only for those vertices that appear without connections to the vertices appearing before them. In the following, we call those vertices *isolated* (although they might get connected to some other vertices appearing later). We cannot reach the upper bound of $n - 2$ by simply asking for one bit of advice for every vertex that is isolated, except the first and the last (if isolated) vertices in the input sequence, because this strategy would require knowing the input length in advance, since the advice tape is infinite and it is up to the algorithm to decide how many bits to read.
Therefore, in order to achieve the desired bound, we present Algorithm 2.1 that works as follows.

- The first vertex receives color 1.
- Then the algorithm asks for one bit of advice: if it is 1, then Algorithm 2.1 will assign color 1 to every isolated vertex, otherwise it will ask for a bit of advice for every further isolated vertex, to decide whether to assign color 1 or 2.
- Any vertex that has an edge to some previously received vertex v receives the opposite color with respect to v.

It is easy to see that, on an input consisting of n vertices, whenever there are at least $n - 1$ isolated vertices, assigning color 1 to every isolated vertex is an optimal strategy, therefore the appropriate advice is a string of length one. This implies that the first advice bit is 0 only when at most $n - 2$ vertices are isolated in the input sequence. Since the first vertex is among them and does not need any advice, the upper bound of $n - 2$ holds. □

We can complement this result by an almost matching lower bound.

Theorem 2.13. *Any deterministic online algorithm for* BipCol *needs at least $n - 3$ advice bits to be optimal on every instance on n vertices.*

Proof. For a contradiction, assume there exists an algorithm $\hat{A}$ for BipCol that uses 2 colors and less than $n - 3$ bits of advice. Given, for any $0 \leq \alpha \leq n - 2$, the

Algorithm 2.1 An Optimal Coloring For BipCol

```
INPUT: a bipartite graph G^≺ ∈ 𝒢_n, for some n ∈ ℕ
 1: c_1 = col(v_1) = 1
 2: read an advice bit σ
 3: if σ = 1 then
 4:     for i = 2 to n do
 5:         if v_i is isolated then
 6:             c_i = col(v_i) = 1
 7:         else
 8:             choose an appropriate color c_i for v_i
 9:         output c_i
10: else
11:     for i = 2 to n do
12:         if v_i is isolated then
13:             read an advice bit σ
14:             if σ = 1 then
15:                 c_i = col(v_i) = 1
16:             else
17:                 c_i = col(v_i) = 2
18:         else
19:             choose an appropriate color c_i for v_i
20:         output c_i
OUTPUT: (c_1, c_2, ..., c_n), for some n ∈ ℕ
```

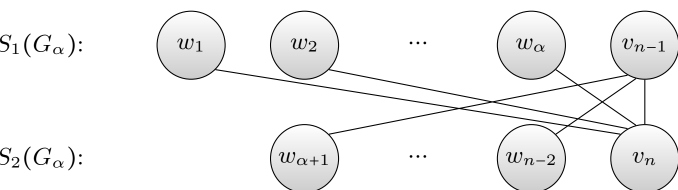
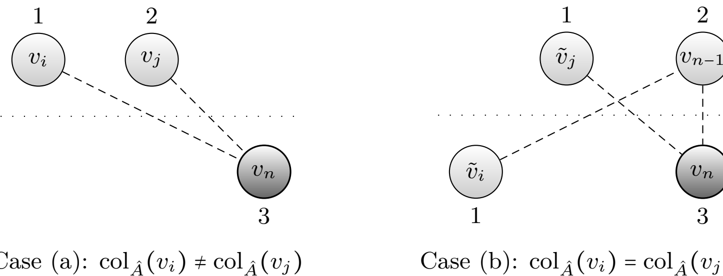


Figure 2.7. Structure of the graph G_α in the proof of Theorem 2.13. The set of edges is $E = \{\{w_i, v_n\} \mid 1 \le i \le \alpha\} \cup \{\{w_i, v_{n-1}\} \mid \alpha + 1 \le i \le n-2\} \cup \{\{v_{n-1}, v_n\}\}$.

graph G_α on n vertices described in Figure 2.7, we consider as the set of possible instances of $\hat{A}$ any online presentation of G_α, for all $0 \le \alpha \le n-2$, such that the first $n-2$ vertices $v_1, v_2, \ldots, v_{n-2}$ in the online presentation are a permutation of the vertices $\{w_1, w_2, \ldots, w_{n-2}\}$. This means, the algorithm will always receive isolated vertices until time step $n-2$. Hence, $\hat{A}$ will be able to color v_{n-1} and v_n with values in $\{1, 2\}$ only if $v_1, \ldots, v_\alpha$ all have the same color, and $v_{\alpha+1}, \ldots, v_{n-2}$ all have the opposite color.



Case (a): $\text{col}_{\hat{A}}(v_i) \neq \text{col}_{\hat{A}}(v_j)$ Case (b): $\text{col}_{\hat{A}}(v_i) = \text{col}_{\hat{A}}(v_j)$

Figure 2.8. Two non-equivalent instances with the same advice string in the proof of Theorem 2.13.

We define an equivalence relation $\sim$ on the possible online representations of the vertices of the graph G_α in the following way: we say that two instances are equivalent iff the order of the first $n-2$ vertices reflects the same shore partition. More formally, for all $i \leq n-2$, let

$$S(v_i) := \begin{cases} S_1(G_\alpha) & \text{if } v_i \in S_1(G_\alpha) \\ S_2(G_\alpha) & \text{else} \end{cases}$$

be the shore containing v_i. Let $(v_1, \ldots, v_{n-2}, v_{n-1}, v_n)$ and $(\tilde{v}_1, \ldots, \tilde{v}_{n-2}, v_{n-1}, v_n)$ be two online representations of the vertices $\{w_1, w_2, \ldots, w_n\}$. Then,

$$(v_1, v_2, \ldots, v_{n-2}, v_{n-1}, v_n) \sim (\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_{n-2}, v_{n-1}, v_n)$$

iff, for all $i, j \leq n-2$,

$$S(v_i) = S(v_j) \iff S(\tilde{v}_i) = S(\tilde{v}_j)\,,$$

i. e., v_i and v_j are on the same shore iff $\tilde{v}_i$ and $\tilde{v}_j$ lie on the same shore. It is not hard to see that $\sim$ is an equivalence relation and, by a counting argument, the number of equivalence classes of $\sim$ is $\frac{2^{n-2}}{2} = 2^{n-3}$, since the shore partition is symmetric with respect to S_1 and S_2.
To prove the claimed lower bound, it is sufficient to show that $\hat{A}$ needs a different advice string for each equivalence class. Suppose, for contradiction, that $(v_1, v_2, \ldots, v_{n-2}, v_{n-1}, v_n) \nsim (\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_{n-2}, v_{n-1}, v_n)$ and $\hat{A}$ receives the same advice string for both instances. Then, since all instances look the same until time step $n-2$, this implies $\text{col}_{\hat{A}}(v_i) = \text{col}_{\hat{A}}(\tilde{v}_i)$, for all $i \leq n-2$.
Because the two instances are not equivalent, there are two values $i, j \leq n-2$, with $i \neq j$, such that v_i and v_j are on the same shore, while $\tilde{v}_i$ and $\tilde{v}_j$ are on opposite shores. We then have two cases (see Figure 2.8):

Case (a): If $\text{col}_{\hat{A}}(v_i) \neq \text{col}_{\hat{A}}(v_j)$, then, in the instance associated to $(v_1, \ldots, v_n)$, either v_{n-1} or v_n is forced to have a third color assigned, since one of them will be on the opposite shore with respect to both v_i and v_j.

Case (b): If $\text{col}_{\hat{A}}(v_i) = \text{col}_{\hat{A}}(v_j)$, then $\text{col}_{\hat{A}}(\tilde{v}_i) = \text{col}_{\hat{A}}(\tilde{v}_j)$. W.l.o.g., we can assume that, in the instance associated to $(\tilde{v}_1, \ldots, \tilde{v}_{n-2}, v_{n-1}, v_n)$, the vertex v_{n-1} is on the shore opposite to $\tilde{v}_i$, hence there is an edge $\{v_{n-1}, \tilde{v}_i\}$ and as a consequence we must have $\text{col}_{\hat{A}}(v_{n-1}) \neq \text{col}_{\hat{A}}(\tilde{v}_i)$ and therefore $\text{col}_{\hat{A}}(v_{n-1}) \neq \text{col}_{\hat{A}}(\tilde{v}_j)$, but since v_{n-1} and $\tilde{v}_j$ are on the same shore, which is opposite to v_{n-2}, the algorithm $\hat{A}$ is forced to assign a third color to v_{n-2}.

Therefore, the algorithm $\hat{A}$ cannot read the same advice string for the two instances associated to $(v_1, \ldots, v_n)$ and to $(\tilde{v}_1, \ldots, v_n)$ in order to be optimal. □

We now analyze how much advice is sufficient to guarantee a given constant competitive ratio.

Theorem 2.14. *For any integer constant $k > 2$, there exists an online algorithm for* BipCol *that needs less than $\frac{n}{\sqrt{2^{k-1}}}$ advice bits to color every instance on n vertices with at most k colors.*

Proof. We will consider an algorithm, Algorithm 2.2, that is an adaptation of the algorithm A used in the proof of Theorem 2.6: the idea is to make the algorithm ask for an advice bit only when it is about to assign color $k-1$, in order to avoid assigning that color to vertices on both shores of the final graph. This implies that the algorithm will always have vertices of color $k-1$ (if any) only on one shore and vertices of color k (if any) only on the other shore, so that color $k+1$ will never be needed.

Algorithm 2.2 A Coloring For BipCol with k colors

INPUT: a bipartite graph $G^{\prec} \in \mathcal{G}_n$, for some $n \in \mathbb{N}$
1: **for** $i = 1$ to n **do**
2:     let $S_1(C_i(v_i))$ denote the shore of $C_i(v_i)$ containing v_i
3:     **if** $S_2(C_i(v_i))$ does not contain all colors smaller than $k-1$ **then**
4:         $c_i = \text{col}_{A_k}(v_i) = \min\{c \geq 1 \mid c \notin \text{col}(S_2(C_i(v_i)))\}$
5:     **else if** either $k-1 \in \text{col}(S_2(C_i(v_i))$ or $k \in \text{col}(S_1(C_i(v_i))$ **then**
6:         $c_i = k$
7:     **else if** either $k \in \text{col}(S_2(C_i(v_i))$ or $k-1 \in \text{col}(S_1(C_i(v_i))$ **then**
8:         $c_i = k-1$
9:     **else**
10:         read an advice bit σ
11:         **if** $\sigma = 1$ **then**
12:             $c_i = k-1$
13:         **else**
14:             $c_i = k$
15:     **output** c_i
OUTPUT: $(c_1, c_2, \ldots, c_n)$, for some $n \in \mathbb{N}$

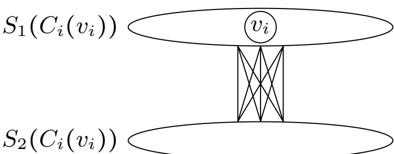


Figure 2.9. The component $C_i(v_i)$ containing the vertex v_i at time step i in Theorem 2.14.

We now describe the work of Algorithm 2.2 at time step i, when the vertex v_i is revealed. Consider the connected component $C_i(v_i)$ to which v_i belongs at time step i. Assume, w.l.o.g., $v_i \in S_1(C_i(v_i))$ (see Figure 2.9).
Algorithm 2.2 will choose its output as follows:

1. If the opposite shore $S_2(C_i(v_i))$ does not contain all the colors smaller than $k-1$, then $\text{col}_{A_k}(v_i) = \min\{c \geq 1 \mid c \notin \text{col}(S_2(C_i(v_i)))\}$.
2. If either $k-1 \in \text{col}(S_2(C_i(v_i))$ or $k \in \text{col}(S_1(C_i(v_i))$, then $\text{col}_{A_k}(v_i) = k$.
3. If either $k \in \text{col}(S_2(C_t(v_i))$ or $k-1 \in \text{col}(S_1(C_i(v_i))$, then $\text{col}_{A_k}(v_i) = k-1$.
4. If $\text{col}(S_2(C_i(v_i)) = \{1, 2, \ldots, k-2\}$, then Algorithm 2.2 asks for one bit of advice to decide whether to assign color $k-1$ or k to v_i.

Algorithm 2.2 asks for an advice bit only when it is about to assign color $k-1$, which may happen at most every $2^{\frac{k-1}{2}}$ vertices, as shown in the proof of Theorem 2.6 for algorithm A, so the maximum number of advice bits required is $\frac{n}{\sqrt{2^{k-1}}}$. □

The proof of Theorem 2.14 can be easily extended to the case of using a non-constant number of colors, only the size n of the input has to be encoded into the advice string. Since the advice tape is infinite and it is up to the algorithm to decide how many bits to read, we need to encode the value n in a self-delimiting way (see Lemma 1.25), otherwise the algorithm could not determine where the encoding of n stops and where the actual advice string starts. Therefore, the new advice string will have $\lceil\log(n)\rceil + 2\lceil\log(\lceil\log(n)\rceil)\rceil$ additional bits. This leads to the following corollary.

Corollary 2.15. *There is an online algorithm for* BipCol *that needs at most $O(\sqrt{n})$ advice bits to color every instance on n vertices with at most $\lceil\log(n)\rceil$ colors.* □

In the remainder of this section, we sketch the case of near-optimal coloring using 3 colors. For this case, Theorem 2.14 gives the following upper bound on the advice complexity.

Corollary 2.16. *There exists an online algorithm for* BipCol *that needs at most $\frac{n}{2}$ advice bits to color every instance on n vertices with at most 3 colors.* □

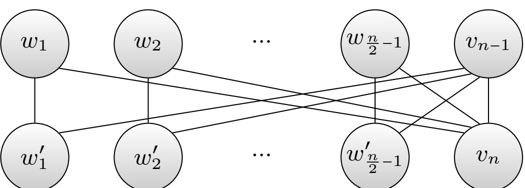


Figure 2.10. Pattern of the graph used in the proof of Theorem 2.17. The edges are $E = \{\{w_i, w'_i\}, \{w_i, v_n\}, \{w'_i, v_{n-1}\}, \{v_{n-1}, v_n\} \mid 1 \le i \le \frac{n}{2} - 1\}$.

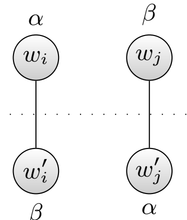


Figure 2.11. If an advice string would lead to $\text{col}(w_i) = \text{col}(w'_j)$ and $\text{col}(w_j) = \text{col}(w'_i)$, two colors would be common.

We conclude this section with an almost matching lower bound for coloring with 3 colors. The detailed proof can be found in [6].

Theorem 2.17 (Bianchi et al. [6]). *Any deterministic online algorithm with advice for* BipCol *needs at least* $\frac{n}{2} - 4$ *advice bits to color every instance on* n *vertices with at most* 3 *colors.*

Proof sketch. Let $n \ge 4$ be even. Consider the graph described in Figure 2.10. The adversary shows in a first step one vertex from each pair $\{w_i, w'_i\}$. After these $\frac{n}{2} - 1$ vertices are revealed, the corresponding neighbors appear. Finally, v_{n-1} and v_n are revealed with edges to all vertices of the opposite shore.

It is easy to see that an instance of the form considered above can be colored with at most 3 colors only if at most 1 color is common, i. e., appears on both shores, in the first $n - 2$ vertices. Therefore, we can never have an advice string such that $\text{col}(w_i) = \text{col}(w'_j)$ and $\text{col}(w_j) = \text{col}(w'_i)$ for some $i, j \in \{1, \ldots, \frac{n}{2} - 1\}$ (see Figure 2.11).

Counting how many instances can get the same advice string, one can estimate the number of instances which need a different advice string in order to receive only 3 colors by any online algorithm with advice. □

2.5 Advice Complexity in Paths, Cycles, and Spider Graphs

In this section, we present results on coloring online paths, cycles and spider graphs with advice. The results from this section are partially derived as a joint work with Michal Forišek and Monika Steinová. We start with the simplest graph class, the paths. Let $P^{\prec} = (P_n, \prec)$ be an online path instance from the set $\mathcal{P}_n$ of all online path instances on n vertices.

Paths

We can take advantage of Lemma 2.11 in coloring paths online, since paths are obviously 2-colorable. If the online presentation of a path does not contain any isolated vertex except the first one, a greedy algorithm solves the problem optimally since the first vertex can receive, w.l.o.g., color 1. In the case of isolated vertices in the online presentation, we can rely on the following theorem.

Theorem 2.18. *There is an online algorithm solving the online coloring problem in online paths on n vertices reading $\lceil \frac{n}{2} \rceil - 1$ advice bits.*

Proof. As suggested by Lemma 2.11, our algorithm only asks for advice (i. e., reads the next bit of the advice string) whenever the vertex v_i in time step i appears isolated. One bit of advice per isolated vertex is sufficient, since the advice can be interpreted as the correct color to use. The above only applies for $i > 1$, as we may pick an arbitrary color for the first isolated vertex.
Let S be the set of vertices that were isolated at the moment when we processed them. Clearly, no two of them are adjacent in G, hence S is an independent set in G and therefore $|S| \leq \lceil \frac{n}{2} \rceil$. □

The following theorem gives a matching lower bound. The complete proof can be found in [30].

Theorem 2.19 (Forišek et al. [30]). *Any online algorithm finding an optimal coloring in online paths needs at least $\lceil \frac{n}{2} \rceil - 1$ bits of advice in the worst case.*

Proof sketch. To prove the lower bound, we describe a set of online path instances on n vertices which start with the same prefix.
Note that, in the online presentation of a path on n vertices, at most $\lceil \frac{n}{2} \rceil$ vertices can be revealed isolated. If the adversary shows only $\lceil \frac{n}{2} \rceil - 1$ vertices isolated, he can place the isolated vertices such that some of them are on odd positions and some of them are on even positions. We call the set of isolated vertices on odd positions P_x and the set of the other isolated vertices Q_x. So, all of these instances start with a prefix of $\lceil \frac{n}{2} \rceil - 1$ isolated vertices. An example for $n = 14$ is shown in Figure 2.12.

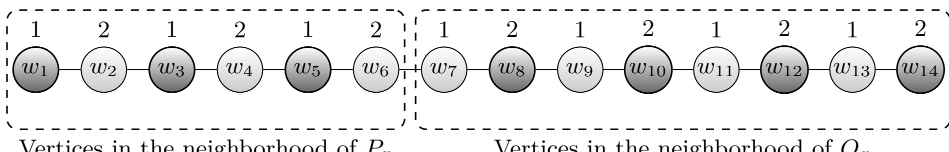


Figure 2.12. An example for the instances proving the lower bound in Theorem 2.19 with $n = 14$ and $x = 3$: $P_x = \{w_1, w_3, w_5\}$ and $Q_x = \{w_8, w_{10}, w_{12}, w_{14}\}$.

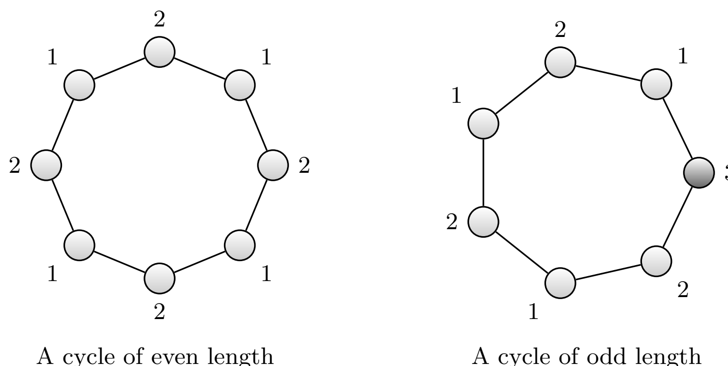


Figure 2.13. A cycle of even length is 2-colorable and a cycle of odd length 3-colorable.

In a path of an even number of vertices, all the vertices in P_x have to get the same color and all vertices in Q_x the other color in order to be optimal on such a path. To count the number of online presentations that need difference advice strings, we pick, for every string $w \in \{\mathtt{p}\} \cdot \{\mathtt{p}, \mathtt{q}\}^{k-1}$, the lexicographically smallest instance from all the instances such that we have $v_i \in P_x$ if and only if the ith letter in w is a $\mathtt{p}$.

In order to color these instances optimally, the algorithm has to treat the instances pairwise differently already on the prefix of isolated vertices. Therefore it needs as many advice strings as there are instances in this set. Since the number of instances is S is $2^{\lceil \frac{n}{2} \rceil - 1}$, we can show the bound of $\lceil \frac{n}{2} \rceil - 1$ advice bits that every algorithm needs to read in order to be optimal.

For odd n, we have to use a more careful analysis to force the extra bit. □

Cycles

In cycles, we have to distinguish cycles of odd and of even length. Cycles with an even number of vertices can be optimally colored using 2 colors and cycles with an odd number of vertices using 3 colors (see Figure 2.13). We start with the easier case, cycles of odd length.

Theorem 2.20. *There is a deterministic online algorithm that solves the problem of coloring an online cycle of odd length without using advice.*

Proof. A greedy strategy provides this upper bound on the number of advice bits that are necessary to color any cycle of odd length. As we are allowed to use three colors and each vertex has at most 2 neighbors, we always have at least one color we may use. □

For even cycles, we cannot exploit the benefit of having a third color on disposal when connecting two subpaths with end vertices of different colors, since cycles of even length are 2-colorable. Therefore, the algorithm needs to read approximately as much advice as in the case of online paths.

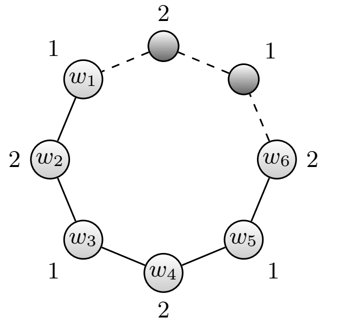
Theorem 2.21. *There is a deterministic online algorithm with advice solving the online coloring problem on cycles of even length reading $\frac{n}{2} - 1$ advice bits.*

Proof. We use the following greedy algorithm: Color the first vertex arbitrarily. In the rest of the instance, whenever being shown an isolated vertex, use an advice bit to color it 1 or 2. Whenever being shown a vertex that is not isolated, use the smallest available color (1 or 2).
In an instance that is a cycle with n vertices there can be at most $\frac{n}{2}$ isolated vertices, hence this algorithm always uses at most $\frac{n}{2} - 1$ advice bits. □

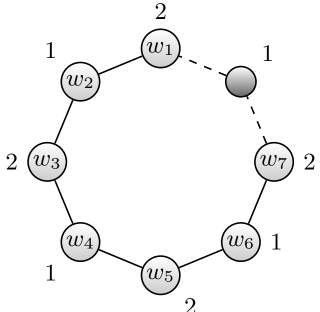
It is easily verified that, for any instance, there is an advice string such that the algorithm produces an optimal coloring – using two colors for even n, three for odd n. (For even n there is one such advice string – the correct coloring. For odd n, all possible advice strings work.)
We can complement this result with an almost matching lower bound.

Theorem 2.22. *Any deterministic online algorithm with advice solving the problem of coloring cycles of even length needs at least $\frac{n}{2} - 2$ bits of advice in the worst case.*

Proof. Observe that optimal colorings for paths and cycles of even length look almost the same – they use two alternating colors. We will use this fact to show that any algorithm solving the coloring problem on cycles of even length can be used to solve the online coloring problem on paths.
Assume that there exists a deterministic online algorithm A_C with advice, solving the online coloring problem in even cycles accessing $f(n)$ advice bits. We will use A_C as a subroutine to define a deterministic online algorithm A_P with advice, solving the online coloring problem on paths with m vertices with at most $f(m+2)$ of advice bits.
Suppose that A_P is given a path on m vertices. Let $n = m + 2$ if m is even, and $n = m + 1$ if m is odd. The algorithm A_P will interpret all incoming isolated vertices as vertices on a cycle with n vertices, and process them by calling A_C. The



A path containing an even number of vertices



A path containing an odd number of vertices

Figure 2.14. A cycle of even length is 2-colorable and a cycle of odd length is 3-colorable.

remaining vertices can be handled directly by A_P (using Lemma 2.11). Figure 2.14 shows an example how these two types of paths can be joined to a cycle.
Clearly, any valid coloring of the cycle corresponds to a valid coloring of the path and vice versa. So the coloring A_C constructed for the cycle directly maps to a valid coloring of the path. Note that A_P does not know m and A_C does not know n. The algorithm A_P never actually computes n, it just checks each vertex for adjacency to the previously processed vertices.
As A_P does not use any additional advice bits, the advice complexity of A_P for a path with m vertices is equal to the advice complexity of A_C for a cycle with n vertices. By Theorem 2.19 we know that $\lceil \frac{m}{2} \rceil - 1$ advice bits are necessary for a path with m vertices. Hence

$$f(n) \geq \left\lceil \frac{m-2}{2} \right\rceil - 1 = \frac{n}{2} - 2 \,.$$

□

Spider Graphs

In this section, we consider special trees where all vertices except for one have degree at most 2. We call those graphs spider graphs (see [31]).

> **Definition 2.23 (Spider Graph).** A *spider graph* is a tree in which all the vertices except of one have degree at most 2 (see Figure 2.15 for an example). The vertex with degree larger than 2 is referred to as the *center* of the spider and the paths from the center are called the *legs* of the spider graph.

Note that, since spider graphs are trees, and trees are bipartite, spider graphs are 2-colorable.

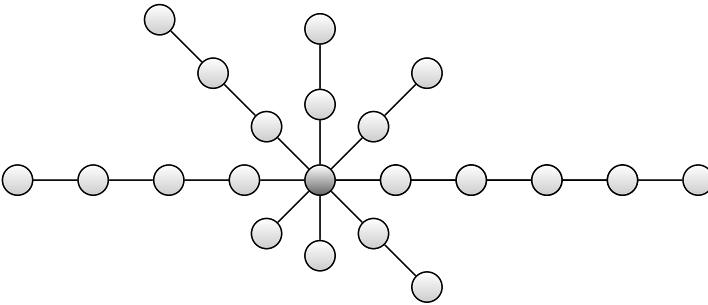


Figure 2.15. Example of a spider graph with 8 legs. The center is marked with dark color. The longest leg contains 5 vertices.

Theorem 2.24. *There is an online algorithm without advice that can color any spider graph using at most 3 colors.*

Proof. The naive greedy strategy "always use the smallest available color" has this property. Any vertex other than the center has at most two neighbors, so it will surely receive a valid color. Now consider the time step i when the online algorithm processes the center v_i. Some neighbors of the center may have been processed before. Consider any such neighbor v_j with $j < i$. At the moment when v_j was revealed, it had at most one already present neighbor – because it has at most two, and v_i was not present yet. Therefore v_j surely received one of the colors 1 and 2. But this means that it is certainly possible to color the center using color 3. □

Note that from Theorem 2.24 it follows that for spider graphs it still does not make much sense to consider the tradeoff between advice size and the competitive ratio of the online algorithm. For spider graphs, a ratio of $\frac{3}{2}$ can be achieved without advice, and we are only able to guarantee a better competitive ratio by solving the problem optimally.

Theorem 2.25. *Any online coloring algorithm with advice needs to read at least least $\frac{2}{3}n - \frac{7}{3}$ advice bits in order to color an online spider graph optimally.*

Proof. For infinitely many n, we define a set of instances on n vertices such that every algorithm has to use a different advice string for each two of those instances in order to color them optimally.
Let $n = 3k + 2$, for some $k \in \{1, 2, 3, \ldots\}$. For a fixed n, the adversary first reveals $\frac{2n-1}{3}$ isolated vertices. Some subset of size $\frac{n+1}{3}$ of these will be extended to legs of length 1 and $\frac{n-2}{3}$ to legs of length 2. In Figure 2.16, the vertices $v_1, s_1, s_2, ..., s_{\frac{n-2}{3}}$ are the isolated vertices in the legs of length 1; moreover, in the legs of length 2, either l_j^1 or l_j^2 will be an isolated vertex, for every $j \in \{1, 2, \ldots, \frac{n-2}{3}\}$.
In the online instance, first, the vertex v_1 is given. This vertex can be colored arbitrarily, but once the color is chosen, it fixes one of the two possible colorings of

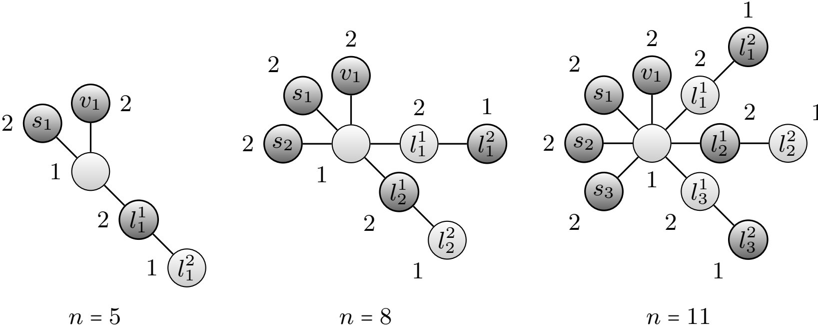


Figure 2.16. Examples for $n = 5, 8, 11$ together with an optimal coloring. Vertices of the same type have to be colored with the same color. The marked vertices indicate the possible isolated vertices in an online presentation of the given spider graph.

the instance. Therefore, we need to know, for every following isolated vertex, what color it has to get: All the vertices $\{s_1, s_2, ..., s_{\frac{n-2}{3}}\}$ will get the same color. And in the legs of length 2, if for some $j \in \{1, 2, \ldots, \frac{n-2}{3}\}$, l_j^1 is the isolated vertex, it will get the same color, and l_j^2 will get the other color.
To count the number of possible instances of this type, we have to note that we are looking for instances in which at least $\frac{n-2}{3}$ isolated vertices will get the same color. The number of instances in which k vertices out of $\frac{2n-4}{3}$ have the same color is

$$\binom{\frac{2n-4}{3}}{k},$$

and this k can range between $\frac{n-2}{3}$ and $\frac{2n-4}{3}$. Therefore, the total number of instances is

$$\sum_{k=\frac{n-2}{3}}^{\frac{2n-4}{3}} \binom{\frac{2n-4}{3}}{k} = \sum_{k=0}^{\frac{n-2}{3}} \binom{\frac{2n-4}{3}}{k} \geq \frac{1}{2} \sum_{k=0}^{\frac{2n-4}{3}} \binom{\frac{2n-4}{3}}{k} = \frac{1}{2} \cdot 2^{\frac{2n-4}{3}} = 2^{\frac{2n-7}{3}}.$$

Since all of these instances start with the same prefix, namely $\frac{2n-1}{3}$ isolated vertices, every algorithm has to distinguish each two of those instances already on this prefix. Therefore, any algorithm needs a different advice string for each of those instances. Hence, any deterministic algorithm needs to read at least $\log\left(2^{\frac{2n-7}{3}}\right) = \frac{2n-7}{3}$ advice bits in order to be optimal.

In [29], we present an almost matching upper bound on the number of advice bits necessary to color an online spider graph optimally. In the proof, we assign to every general spider graph a so-called 1-2-spider graph.

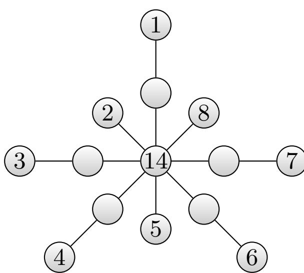


Figure 2.17. A partially labeled 1-2-spider graph on $n = 14$ vertices. The end vertices are continuously labeled with 1 to l, where l is the number of legs, and the center gets the label n.

Definition 2.26 (1-2-Spider Graph). A spider graph is called a *1-2-spider graph* if each leg consists of at most two vertices. A 1-2-spider graph is *partially labeled* if the end vertices of the legs are labeled 1 through ℓ (where ℓ is the number of legs), the center is labeled n (where n is the total number of vertices) and the other vertices are unlabeled (see Figure 2.17 for an example).

Theorem 2.27 (Forišek et al. [29]). *There is a deterministic online algorithm with advice solving the online coloring problem in spider graphs with advice complexity* $\log(\phi) \cdot n + 3\log(n) + O(1)$*, where* $\phi = \frac{1+\sqrt{5}}{2}$ *denotes the golden ratio.*

Proof sketch. The general idea of the proof is that if the algorithm only needs to read advice for coloring isolated vertices, each possible spider graph is equivalent to a partially labeled 1-2-spider graph in which the isolated vertices are precisely the endpoints of all legs. Furthermore, multiple spider graphs correspond to the same 1-2-spider graph, which decreases the amount of advice we need.
The advice string for the algorithm consists of three parts.

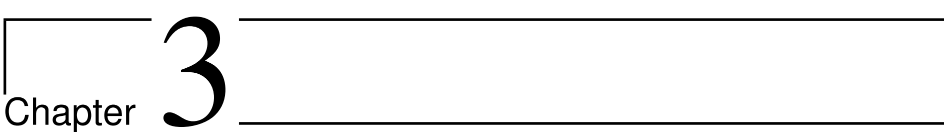
1. In the first part, the number n of vertices n is encoded.
2. Then, the number of the time step, in which the center vertex is revealed follows. We fix the color of the center vertex to be 2.
3. The rest of the advice string contains the correct coloring of all vertices that appear isolated in a certain time step since due to Lemma 2.11, these are the only interesting vertices where the online algorithm needs advice.

The oracle computes the advice string as follows. It transforms the online spider graph $(S, \prec)$ to a partially labeled 1-2-spider graph S' by adding, for every vertex that appears isolated and should get color 1, a leg of length 1 to the graph and, for each isolated vertex with optimal color 2, a leg of length 2. The end vertices of the legs are labeled in the order in which they were added.

According to the above observation, the resulting instance I' will never have more than n vertices. If it has less, the oracle adds additional legs of length 1 until the number of vertices is correct. Then, the algorithm transmits the number of this graph S' in the lexicographic order of all 1-2-spider graphs on n vertices.
One can show that there are F_n possible instances S', with F_n being the n-th Fibonacci number, and hence the oracle needs $\log(F_n) = \log(\phi) \cdot n$ bits to describe any of them. □

2.6 Conclusion

In this chapter, we first showed an improved lower bound on the number of colors that any deterministic online algorithm without advice is forced to use to color an online bipartite graph. We conjecture that this inductive technique can be also transferred to other graph classes with a structure similar to paths and bipartite graphs, namely to graph classes which can be built recursively, i. e., where every subgraph of such a graph has the same properties as the whole graph.
Furthermore, we investigated the number of advice bits a deterministic online algorithm has to access in order to be optimal in an online bipartite graph or in subclasses such as paths, cycles, and spider graphs, and proved almost matching lower and upper bounds in most of these cases. This could be extended to other graph classes. Moreover, a lower bound for a general tradeoff between the number of advice bits that have to be accessed and the competitive ratio is left open.



Online Matching in Bipartite Graphs

The matching problem appears, for example, in economics where the question arises who or what is associated with whom or what (see [54] for examples). These problems can be modeled as matching problems in bipartite graphs. If, for example, a company introduces a new sector, it has to occupy some new jobs. Some people apply for one or more of these jobs, and the company has to choose at most one of the suitable applicants for each open position (see Figure 3.1). The goal is to occupy as many open positions as possible.

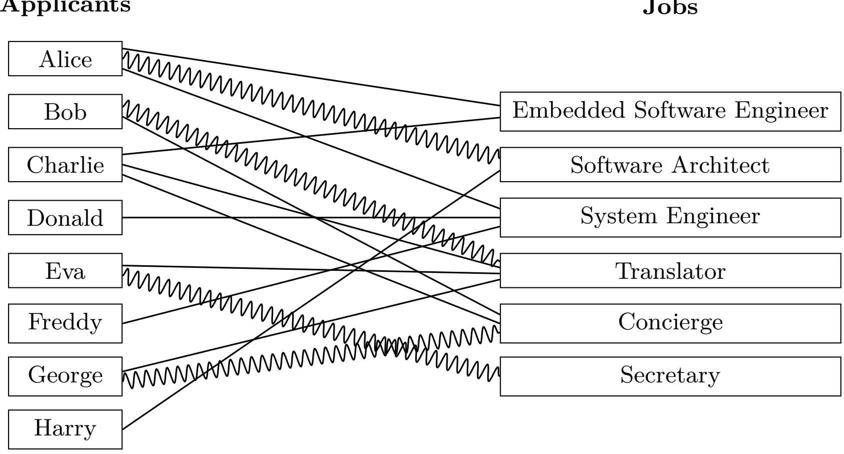
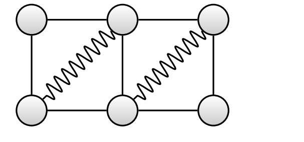


Figure 3.1. The black lines are the applications that are suitable and the wiggly lines show the employments.


(a) A maximal matching

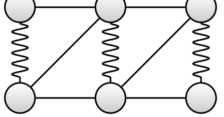

(b) A maximum matching

Figure 3.2. The maximal matching in (a) cannot be extended, (b) shows one possible maximum matching, i. e., a maximal matching of maximum size.

Figure 3.1 can be seen as a graph $G = (V, E)$ with vertex set

$$V = \{\text{Alice}, \text{Bob}, \dots, \text{Harry}, \text{Embedded Software Engineer}, \dots, \text{Secretary}\}$$

and the edge set

$$E = \{\{\text{Alice}, \text{Embedded Software Engineer}\}, \{\text{Alice}, \text{Software Architect}\}, \dots\}\,.$$

The wiggled edges are called *matching edges*.

Definition 3.1 (Matching). Let $G = (V, E)$ be a graph. A subset $M \subseteq E$ is called a *matching* if the edges in M are pairwise not adjacent, i. e., no two edges have a common end vertex.
The *size of a matching* is measured by the number of edges in the matching.
We call a vertex v *matched* in a matching M if there is an edge $e \in M$ such that v is an end vertex of e.

In our example, the company would like to achieve a matching that matches all vertices of the shore containing the jobs since every open position should be occupied if possible. Since the shore containing the jobs is smaller than the other shore and since there are no edges within a shore, a matching which matches all jobs cannot be extended.

Definition 3.2 (Maximal and Maximum Matching). A matching is said to be *maximal* if the matching cannot be extended by adding a new edge, i. e., every edge $e \in E \setminus M$ is adjacent to at least one edge in M.
A maximal matching is *maximum* if there does not exist any maximal matching containing more edges, i. e., a maximum matching is a matching of maximum size among all possible matchings.

Note that every maximum matching is maximal, but the converse does not need to hold. Figure 3.2 shows a maximal and a maximum matching for a graph.
The offline version of the problem of finding a maximum matching in a graph, i. e., when the whole graph is known in the beginning, can be solved with polynomial algorithms (see, e. g., [2, 24, 50, 51, 53]). This problem has been widely investigated, an overview of the matching theory can be found in [57]. Edmonds used in [24] so-called augmenting paths to find a maximum matching.

Figure 3.3. In an augmenting path relative to a matching M, the edges can be flipped in order to get a matching M' containing one edge more than M.

Definition 3.3 (Augmenting Path [1]). An *augmenting path* relative to a matching M is a path which connects two unmatched vertices and in which the edges alternate from matching to non-matching.

Since the end vertices are unmatched, the edges in an augmenting path can be flipped without causing any conflict, leading to a matching containing one edge more than the original matching (see Figure 3.3 for an example).
Usually, the job and applicant situations changes from time to time. Therefore, it makes sense to allow that the jobs and the applicants appear one by one with the edges to already present vertices. This is the online version of the problem of finding a maximum size matching in a graph. An online algorithm receives the graph instance vertex by vertex, and, in every time step, it has to decide immediately if one of the incident edges is a matching edge or not. The goal is to find a maximum matching.

Definition 3.4 (Online Matching Problem).

Input: An online graph $G^{\prec} = (G, \prec)$ with $G = (V, E)$ and $V = \{v_1, v_2, \ldots, v_n\}$

Output: $(M_1, M_2, \ldots, M_n)$ such that $M_i \subseteq E$ is a matching on $G^{\prec}[V_i]$ and $M_i \subseteq M_j$ for all $i < j$ with $i \in \{1, 2, \ldots, n-1\}$ and $j \in \{2, 3, \ldots, n\}$

Cost: $|M_n|$

Goal: Maximum

Note that we are interested in the matching in the final graph $G^{\prec}[V_n]$ being maximum. Therefore, a matching in a subgraph $G^{\prec}[V_i]$, for some $i < n$, does not necessarily have to be a maximum matching in the subgraph induced by the vertex set V_i, but it has to be part of a maximum matching of the final graph, i. e., $M_i \subseteq M_n$ for all $i < n$.
In the literature, the described online matching problem is sometimes called the *fully* online matching problem since the vertices appear in arbitrary order, independent of the two shores. A better studied online matching problem is the *one-sided* online matching problem in bipartite graphs where one shore is given (e. g., the jobs) and only the vertices from the second shore (e. g., the applicants) appear in an online manner. This problem, first introduced in [41], is one of the basic problems that is subject of competitive analysis. The authors showed that the

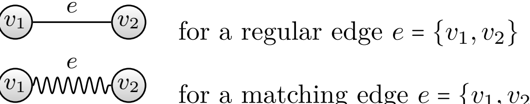
greedy algorithm, which assigns every possible edge to the matching, computes a 2-competitive solution in an arbitrary bipartite graph. It is easy to see that, in a general bipartite graph, there is no algorithm with a better competitive ratio. The instances showing this lower bound, consist of paths of four vertices. The adversary always shows the middle edge of each path first, which prevents the algorithm to take the two adjacent edges to the matching since it always assigns the first edge to the matching.

Theorem 3.5 (Karp et al. [41]). *A greedy algorithm can solve the one-sided online matching problem with a competitive ratio of 2. No other deterministic online algorithm can reach a better competitive ratio.* □

We will only consider the fully online matching problem in this thesis, and therefore we call this problem *online matching problem* for short. The lower bound on the competitive ratio for the one-sided online matching problem in general graphs of Theorem 3.5 carries over to the fully online matching problem. In [26], the authors complemented this result with an upper bound for the fully online matching problem on general graphs.

Theorem 3.6 (Favrholdt et al. [26]). *The competitive ratio of the greedy algorithm for solving the (fully online) maximum matching in general graphs is 2 and this is optimal among deterministic online algorithms.* □

In the remainder of the chapter, we will use the following notation in the figures:



for a regular edge $e = \{v_1, v_2\}$

for a matching edge $e = \{v_1, v_2\}$

3.1 Paths and Trees Without Advice

In a first step, we will summarize in this subsection which competitive ratio is achievable without using any advice. We explore both paths and trees.

Competitive Ratio for Online Algorithms Without Advice on Paths

Let $P^{\prec} = (P_n, \prec)$ be an online graph instance where $P_n = (V, E)$ is a path on n vertices. According to Definition 1.21, let $\mathcal{P}_n$ be the *set of all online path instances on n vertices.* We start with counting the number of maximum matchings in a path.

Lemma 3.7. *In a path P_n with n even, there is exactly one maximum matching, while a path with an odd number n of vertices contains exactly $\frac{n+1}{2}$ maximum matchings.*

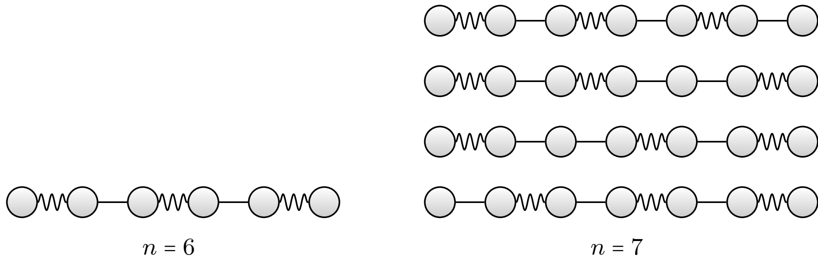


Figure 3.4. The path with an even number of vertices on the left side has exactly one maximum matching. The path on the right side which contains an odd number of vertices has more than one maximum matching.

Proof. In the case of n even (see Figure 3.4 left), we have an odd number of edges and therefore it is clear that every second edge, including the two end edges, belong to the unique maximum matching.
In the case of n odd (see Figure 3.4 right), one vertex stays unmatched. Note that two unmatched vertices would lead to an augmenting path and therefore the matching would not be a maximum matching. The unmatched vertex can be chosen such that removing this vertex and its incident edges provides either one (in the two cases where the unmatched vertex is an end vertex) or two paths with an even number of vertices, because a path with an odd number of vertices would result in an additional unmatched vertex. The required properties for the unmatched vertex are satisfied by the two end vertices and every second inner vertex. Therefore, in the case of n odd, we have exactly $\frac{n+1}{2}$ possible maximum matchings. □

For the discussion of upper bounds, it is sometimes easier to design an algorithm that seeks for a matching that matches all inner vertices. Clearly, the following holds.

Lemma 3.8. *For every path P_n, there is always a maximum matching which matches all inner vertices.*

Proof. In the case of an even number of vertices, the unique matching satisfies this property. In the odd case, exactly two matchings match all inner vertices. Namely the two matchings with an unmatched end vertex. In Figure 3.4, for example, the first and the last matching in the case $n = 7$ are matchings satisfying this property. □

In [26], the authors showed, by induction, a matching lower and upper bound of $\frac{3}{2}$ on the competitive ratio in online paths. We give an alternative proof of this fact since we need the idea later, e. g., for solving the online matching problem on trees. To show upper bounds on the competitive ratio for the problem of finding a maximum matching on online paths, we will show that the greedy algorithm is reasonably good. This algorithm assigns every possible edge to the matching.

Algorithm 3.3 Greedy Algorithm for the Online Matching Problem on Online Graphs

INPUT: $G^{\prec} \in \mathcal{G}_n$ with $G^{\prec}$ an online graph, for some $n \in \mathbb{N}$
1: $M = \varnothing$
2: **for** $i = 1$ to n **do**
3: **for** all $e = \{v_i, v_j\} \in E_i$ **do**
4: **if** v_i and v_j are unmatched **then**
5: $M \leftarrow M \cup \{e\}$
6: **else**
7: $M \leftarrow M$
8: **output** $M_i = M$
OUTPUT: $M = \{e_{i_1}, e_{i_2}, \ldots, e_{i_m}\} = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

Especially, all isolated edges will become matching edges. First, we need to investigate how many edges a greedy algorithm assigns to the matching in the worst case.
To describe the algorithm in detail, we need a notation for the edges that appear in time step i with the revealed vertex v_i, and we have to give an order on these edges. Therefore, we first order the edges with respect to the order of the appearance of the vertices v_i.

Definition 3.9 (Set of Edges Appearing in Time Step i). Let $G^{\prec}$ be an online graph with $G = (V, E)$ and $V = \{v_1, v_2, \ldots, v_n\}$. We denote by E_i the set of edges that appear in time step i by adding the vertex v_i, i. e.,

$$E_i = \{\{v_i, v_j\} \in E \mid j < i\} \ ,$$

for all $i \in \{1, 2, \ldots n\}$.

Note that these edge sets E_i, for all $i \in \{1, 2, \ldots, n\}$, form a partition of the edge set E, i. e., $\bigcup_{i=1}^{n} E_i = E$ and $E_i \cap E_j = \varnothing$ for $i, j \in \{1, 2, \ldots, n\}$ with $i \neq j$.
Within a set E_i, the order of edges is defined as follows. Let v_j, v_k be two vertices adjacent to v_i that appear before the vertex v_i, i. e., $j, k < i$. Then $\{v_i, v_j\} \prec \{v_i, v_k\}$ if and only if $j < k$. This means that the order of the edges is given by the order of appearance of their incident vertices. Note that such a set E_i can also be empty if v_i is an isolated vertex. In particular, $E_1 = \varnothing$.
A detailed description of the greedy algorithm for an arbitrary online graph as input is given in Algorithm 3.3.

Lemma 3.10. *Algorithm 3.3 for finding a maximum matching in an online path on n vertices chooses a matching of size at least $\left\lfloor \frac{n}{3} \right\rfloor$.*

Proof. The greedy algorithm A assigns every possible edge to the matching. Therefore, A matches all isolated edges. A previously isolated matching edge prevents at

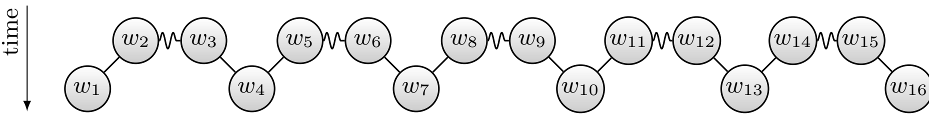


Figure 3.5. The worst-case online setting of a path on 16 vertices for the greedy algorithm of Lemma 3.10. The greedy algorithm assigns 5 instead of 8 edges to the matching.

most the two adjacent edges (or the single adjacent edge) to be matching edges. A non-isolated matching edge inhibits at most one adjacent edge to be a matching edge. Therefore, at least one of three consecutive edges is a matching edge yielding that the greedy algorithm determines at least $\lfloor \frac{n}{3} \rfloor$ matching edges in every path on n vertices. □

We want to describe a worst-case instance for the greedy algorithm. Note that every third edge can appear as an isolated edge in the online setting. In the worst case, the adversary shows every third edge to the algorithm starting with the second edge from one end, for a number n of vertices with $n = 3k + 1$ for any $k \in \mathbb{N}$ (see Figure 3.5). In this case, the greedy algorithm finds a matching containing k edges, which can be reformulated as

$$k = \left\lfloor k + \frac{1}{3} \right\rfloor = \left\lfloor \frac{3k+1}{3} \right\rfloor = \left\lfloor \frac{n}{3} \right\rfloor,$$

yielding the desired result.
This greedy algorithm leads to an upper bound on the competitive ratio for the online matching problem on paths. Note that every maximum matching for a path on n vertices contains $\lfloor \frac{n}{2} \rfloor$ matching edges (see Lemma 3.7 and the corresponding Figure 3.4).

Theorem 3.11. *Algorithm 3.3 for finding a maximum matching in online paths has a competitive ratio of* $\frac{3}{2}$.

Proof. For paths with at most two edges, Algorithm 3.3 obviously computes an optimal solution. Therefore, let I be an online path on n vertices, for $n \geq 3$. We showed in Lemma 3.7 that this path contains at least one maximum matching consisting of $\lfloor \frac{n}{2} \rfloor$ edges, leading to $\text{Opt}(I) = \lfloor \frac{n}{2} \rfloor$.
Together with Lemma 3.10, we have for the competitive ratio c

$$\left\lfloor \frac{n}{2} \right\rfloor = \text{cost}(\text{Opt}(I)) \leq c \cdot \text{cost}(A(I)) + \alpha = c \cdot \left\lfloor \frac{n}{3} \right\rfloor + \alpha, \tag{3.1}$$

for some $\alpha \geq 0$. With $\lfloor \frac{n}{3} \rfloor \geq \frac{3}{2} - 1$ and setting $c = \frac{3}{2}$ and $\alpha = \frac{3}{2}$, we can show that

$$\left\lfloor \frac{n}{2} \right\rfloor \leq c \cdot \left(\frac{n}{3} - 1 \right) + \alpha = \frac{3}{2} \cdot \left(\frac{n}{3} - 1 \right) + \frac{3}{2}.$$

holds, directly implying inequality (3.1). □

For some lengths of the path, we can even reach a strict competitive ratio of $\frac{3}{2}$.

Corollary 3.12. *Algorithm 3.3 for finding a maximum matching on online paths on n vertices with $n \bmod 6 \in \{0, 1, 3\}$ is strictly $\frac{3}{2}$-competitive.*

Proof. We will discuss the two cases $n \bmod 6 \in \{0, 1\}$ and $n \bmod 6 = 3$ separately:

$n \bmod 6 \in \{0, 1\}$: For $n = 6k$ and $n = 6k + 1$, for any $k \in \mathbb{N}$, there is a maximum matching of size

$$\left\lfloor \frac{n}{2} \right\rfloor = 3k\,.$$

The greedy algorithm assigns, due to Lemma 3.10, at least

$$\left\lfloor \frac{n}{3} \right\rfloor = 2k$$

edges to the matching. Therefore, we have the inequality

$$3k = \mathrm{cost}(\mathrm{Opt}(I)) \le c \cdot \mathrm{cost}(A(I)) + \alpha = c \cdot 2k + \alpha\,,$$

which holds for $c = \frac{3}{2}$ and $\alpha = 0$ since

$$3k \le \frac{3}{2} \cdot 2k = 3k\,.$$

$n \bmod 6 = 3$: In this case, the path contains a maximum matching of $\lfloor \frac{n}{2} \rfloor = 3k + 1$ and the greedy algorithm includes $\lfloor \frac{n}{3} \rfloor = 2k + 1$ edges into the matching. Therefore, the following inequality holds for $c = \frac{3}{2}$ and $\alpha = 0$

$$3k + 1 = \mathrm{cost}(\mathrm{Opt}(I)) \le c \cdot \mathrm{cost}(A(I)) + \alpha = \frac{3}{2} \cdot (2k + 1) = 3k + \frac{3}{2}\,.$$

□

In order to show a lower bound on the competitive ratio for arbitrary online paths, we need to investigate the relationship between unmatched vertices and the difference of edges in the chosen matching to the maximum matching.

Lemma 3.13. *If a matching M in a path P leaves k vertices unmatched, then a maximum matching in P has $\lfloor \frac{k}{2} \rfloor$ edges more than M.*

Proof. Let $w_{i_1}, w_{i_2}, \ldots, w_{i_k}$ be the k unmatched vertices from left to right on the path with $i_l < i_m$ for every $l < m$ in a fixed matching M. Then the $\lfloor \frac{k}{2} \rfloor$ subpaths from w_{i_j} to $w_{i_{j+1}}$, for an odd number $j = 2l + 1$ and for all $l \in \{1, 2, \ldots, \lfloor \frac{k}{2} \rfloor\}$, are vertex-disjoint augmenting paths (i.e., alternating paths that start and end with an unmatched vertex). Each of those augmenting paths gives rise to one edge that could be achieved more in a better local maximum matching on this subpath. Therefore, the global maximum matching contains $\lfloor \frac{k}{2} \rfloor$ matching edges more than the fixed matching M. □

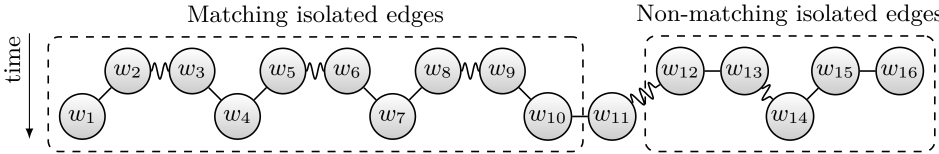


Figure 3.6. An example for $k = 2$ of an online instance with a possible solution proving the lower bound on the competitive ratio for an algorithm solving the online matching problem on an online path. The 5 edges in the upper level are presented as isolated edges in the beginning.

We show that no algorithm can reach a better competitive ratio for finding a maximum matching in a path on n vertices than Algorithm 3.3 of Theorem 3.11.

Theorem 3.14. *No deterministic online algorithm for finding a maximum matching in a path on n vertices with $n \geq \frac{3(\alpha+1)}{\varepsilon} - 2$ can be better than $\left(\frac{3}{2} - \varepsilon\right)$-competitive (with additive constant α), for arbitrary non-negative constants α and $\varepsilon \leq \frac{1}{2}$.*

Proof. To show the lower bound of $\left(\frac{3}{2} - \varepsilon\right)$, for some non-negative constant $\varepsilon \leq \frac{1}{2}$, on the competitive ratio of finding a maximum matching in a path on n vertices, we need to describe an instance class such that, for infinitely many n, every algorithm A has a competitive ratio of at least $\left(\frac{3}{2} - \varepsilon\right)$ on these instances. Therefore, for an instance I from this instance class and every fixed non-negative constant α,

$$\text{cost}(\text{Opt}(I)) \leq c \cdot \text{cost}(A(I)) + \alpha \tag{3.2}$$

only holds for $c \geq \left(\frac{3}{2} - \varepsilon\right)$.

Let $n = 6k + 4$ for some $k \in \mathbb{N}$. Every instance starts with $\frac{n-1}{3}$ isolated edges. A fixed algorithm A assigns some of these edges to the matching and some not. The adversary partitions these edges into two groups, the matching and the non-matching edges.

Assume first that the algorithm chooses at least one isolated edge for the matching. Then, the adversary connects each two of the isolated matching edges with new vertices such that a path emerges (see Figure 3.6 for an example). Appending one vertex to the left and one to the right of this subpath concludes the part with the matching edges. We call this subpath P_1. Then, if the second group of isolated edges is non-empty, each two of these non-matching edges are connected with a new vertex to a subpath P_2. To connect these two paths, P_1 and P_2, in a third step, if necessary, the adversary uses a single vertex.

If the algorithm A does not assign any isolated edge to the matching, the adversary appends the last two remaining vertices, after connecting the isolated edges by single vertices to a path as described in Figure 3.6, to the left and the right end of the subpath.

Now we estimate the competitive ratio with respect to the number of isolated edges that a fixed algorithm A takes into the matching. Assume that the algorithm

chooses, w.l.o.g., the connecting edge between the vertex that connects the two subpaths and P_2 into the matching (as shown in Figure 3.6). This assumption simplifies the estimation of the number of unmatched vertices.
Therefore, if the algorithm assigns $i \leq \frac{n-1}{3}$ of the isolated edges to the matching, P_1 contains exactly $i+1$ unmatched vertices and P_2 has at least $\frac{n-1}{3} - i - 1$ unmatched vertices, since there are $\frac{n-1}{3} - i$ isolated edges in P_2 and the leftmost vertex is already matched because of the assumption above. So, in total we have at least $\frac{n-1}{3}$ unmatched vertices in the instance if P_1 is non-empty. If P_1 is empty, the adversary can only guarantee at least $\frac{n-1}{3} - 1$ unmatched vertices for this instance. Summarizing, the adversary can force the algorithm to invoke at least $\frac{n-1}{3} - 1 = \frac{6k+4-1}{3} - 1 = 2k$ unmatched vertices, leading to a loss of k matching edges with respect to the maximum matching containing $\frac{n}{2} = 3k+2$ matching edges. Therefore, the matching fixed by algorithm A contains

$$3k + 2 - k = 2k + 2 = 2 \cdot \frac{n-4}{6} + 2 = \frac{n}{3} + \frac{2}{3}$$

matching edges. Therefore, for any instance I of this instance class and every non-negative constant α, it holds that

$$\frac{n}{2} = \mathrm{cost}(\mathrm{Opt}(I)) \leq c \cdot \mathrm{cost}(A(I)) + \alpha = c \cdot \left(\frac{n}{3} + \frac{2}{3}\right) + \alpha\,.$$

This yields for the competitive ratio that

$$c \geq \frac{\frac{n}{2} - \alpha}{\frac{n}{3} + \frac{2}{3}} = \frac{\frac{n-2\alpha}{2}}{\frac{n+2}{3}} = \frac{3(n-2\alpha)}{2(n+2)} = \frac{3n}{2n+4} - \frac{6\alpha}{2n+4}$$

Therefore, there exists, for every constant $\varepsilon > 0$ and every $\alpha > 0$, a sufficiently large n_0 such that, for every $n \geq n_0$, we have that

$$c \geq \frac{3}{2} - \varepsilon\,.$$

More precisely, n_0 has to satisfy the inequality

$$\frac{3n_0}{2n_0+4} - \frac{6\alpha}{2n_0+4} \geq \frac{3}{2} - \varepsilon$$

and therefore,

$$3n_0 - 6\alpha \geq (2n_0+4) \cdot \frac{3-2\varepsilon}{2} = \frac{6n_0 - 4n_0\varepsilon + 12 - 8\varepsilon}{2} = 3n_0 - 2n_0\varepsilon + 6 - 4\varepsilon\,.$$

This leads to

$$n_0 \geq \frac{3(\alpha+1)}{\varepsilon} - 2\,.$$

□

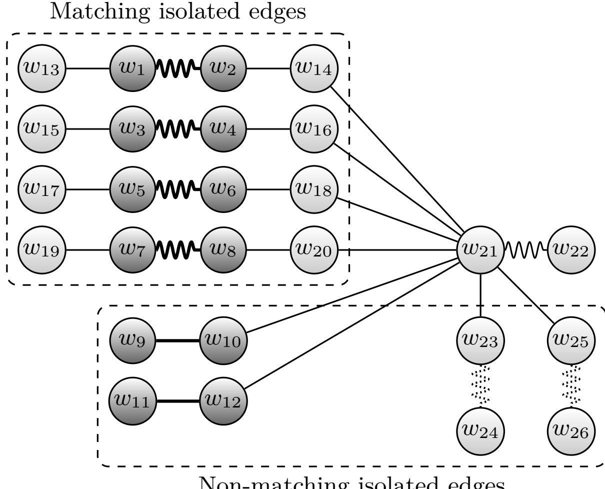


Figure 3.7. An example for $k = 6$ of an online tree instance with a possible solution proving the lower bound on the competitive ratio for an algorithm solving the online matching problem on an online tree. The six marked isolated edges are presented as isolated edges in the beginning. The dotted edges can either be assigned to the matching or not.

Competitive Ratio for Online Algorithms Without Advice on Trees

The upper bound of 2 on the competitive ratio for general bipartite online graphs carries over to online trees. Now we show a matching lower bound for online trees using the construction idea from Theorem 3.14 for trees.

Theorem 3.15. *No deterministic online algorithm for finding a maximum matching in a tree on n vertices can be better than 2-competitive.*

Proof. We describe an instance class of online trees such that, for infinitely many n, every deterministic online algorithm A has a competitive ratio of at least 2 on these instances, showing a lower bound of 2 on the competitive ratio for finding a maximum matching in an online tree on n vertices.
Let $n = 4k + 2$ for some $k \in \mathbb{N}$. The adversary shows first $k = \frac{n-2}{4}$ isolated edges to the algorithm. A fixed algorithm A assigns some of these edges to the matching and leaves some of them unmatched. Then, the adversary completes the tree in such a way that the decision of the online algorithm is wrong for each of the isolated edges. Therefore, the algorithm loses, for every isolated edge, one matching edge with respect to the maximum matching.
More precisely, the instance is expanded as shown in Figure 3.7. After showing the edges $\{w_i, w_{i+1}\}$ for $i \in \{1, 3, 5, \ldots, 2k-1\}$, the online algorithm assigns, w.l.o.g., the

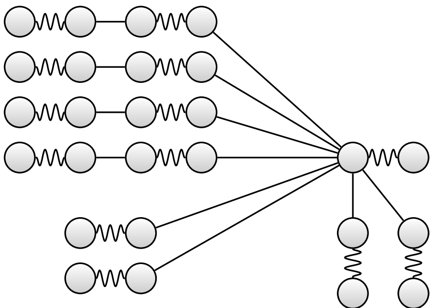


Figure 3.8. A maximum matching in the instance of Figure 3.7.

first $m \geq 0$ edges to the matching. In a next step, the adversary reveals the vertices $\{w_i, w_{i+2k}\}$, for $i \in \{1, 2, 3, \ldots, 2m\}$, which lie to the left or to the right of the first m isolated edges. Since the first m isolated edges are matching edges, the new edges cannot be assigned to the matching implying that the chosen matching has half the size of the maximum matching on the constructed paths of length 3. In a next step, the connecting vertex $w_{2k+2m+1}$ and an adjacent vertex $w_{2k+2m+2}$ are revealed. W.l.o.g., the algorithm chooses this new edge $\{w_{2k+2m+1}, w_{2k+2m+2}\}$ to the matching (also any of the edges incident to $w_{2k+2m+1}$ and previous vertices could be chosen). Then, in a last step, the adversary appends, for each previous non-matching isolated edge, one edge $\{w_i, w_{i+1}\}$ for every $i \in \{2m + 2k + 3, 2m + 2k + 5, \ldots, 4k - 1\}$ to the vertex $w_{2k+2m+1}$ in order to construct equally large instances with an optimal matching of the same size, no matter how many previously isolated edges are assigned to the matching. We do this padding just for the ease of calculation.

Now we want to estimate the size of the matching chosen by a fixed online algorithm A. Since we assumed that the edge $\{w_{2m+2k+1}, w_{2m+2k+2}\}$ is a matching edge, the algorithm has no possibility to assign another edge incident to $w_{2m+2k+1}$ to the matching. But it can take the edges $\{w_i, w_{i+1}\}$ for all $i \in \{2m + 2k + 3, 2m + 2k + 5, \ldots, 4k - 1\}$ to the matching. Therefore, the algorithm assigns at most $k + 1$ edges to the matching leading to a matching of size

$$k + 1 = \frac{n + 2}{4}$$

in the best case.

To determine the competitive ratio, we need to know the size of the maximum matching. Figure 3.8 shows the unique maximum matching of size

$$2k + 1 = \frac{n}{2}\,.$$

Summarizing, for an online instance on n vertices from this special instance class and for any $\alpha \geq 0$, we have

$$\frac{n}{2} = \text{cost}(\text{Opt}(I)) \leq c \cdot \text{cost}(A(I)) + \alpha = c \cdot \frac{n+2}{4} + \alpha\,.$$

This inequality is satisfied, for all input sizes n, only if $c \geq 2$. □

3.2 Optimality in Paths and Cycles

In this section, we will show how much advice an algorithm needs to read in order to solve the online matching problem in paths and cycles optimally. For this, we will mostly focus on matchings which satisfy the property stated in Lemma 3.8, i. e., matchings which match all inner vertices.

In the online setting of a path P_n, the isolated edges play a special role. Since, at a certain time step i, they are not appended to any other subpath, they get no information from the neighborhood about whether they are matching or non-matching edges. Therefore, an online algorithm for solving the online matching problem on a path should get some external information from the advisor on these edges in order to be optimal. We show that there cannot appear too many isolated edges in an online path instance.

Theorem 3.16. *There exists an online algorithm with advice which uses at most $\lceil \frac{n}{3} \rceil$ advice bits to find a maximum matching in a path $P^{\prec} \in \mathcal{P}_n$ on n vertices.*

Proof. We consider an online algorithm with advice which matches all inner vertices. Due to Lemma 3.8, there is always such a maximum matching and therefore we can choose an algorithm striving for this special maximum matching. Let $P^{\prec} \in \mathcal{P}_n$ be the online path on n vertices. In time step i, the vertex v_i can be either isolated or connected to one previous vertex v_j or to two previous vertices v_{j_1} and v_{j_2} with $j, j_1, j_2 < i$. Since the algorithm will look for a matching that matches all inner vertices, the algorithm knows what to do with an edge $\{v_i, v_j\}$ if the vertex v_j is not isolated in time step i: If v_j is not part of a matching edge until time step i, the edge $\{v_i, v_j\}$ will be a matching edge. Analogously, the algorithm knows which one of the two edges $\{v_{j_1}, v_i\}$ and $\{v_{j_2}, v_i\}$ is a matching edge if not both vertices v_{j_1} and v_{j_2} are isolated (see Figure 3.9).

Note that the case that a vertex v_i is connected to two non-isolated and unmatched vertices v_{j_1}, v_{j_2} (see Figure 3.10) cannot occur. Algorithm 3.4 strives for a matching that matches all inner vertices. This case would be a contradiction to this fact because either v_{j_1} or v_{j_2} would stay unmatched after choosing the matching edge incident to v_i.

It is also not possible that both vertices v_{j_1} and v_{j_2} are already matched in a previous time step since in this case, vertex v_i would remain unmatched (see Figure 3.11).

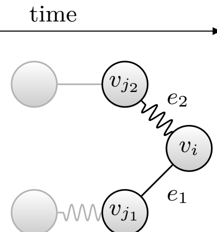


Figure 3.9. If at least one of the two previous vertices v_{j_1} or v_{j_2} is not isolated, the algorithm knows from the context which one of the two edges $\{v_i, v_{j_1}\}$ and $\{v_i, v_{j_2}\}$ should be a matching edge.

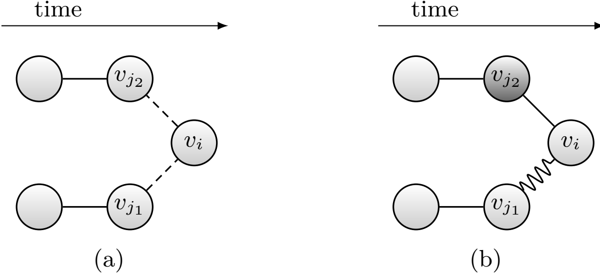


Figure 3.10. It cannot appear that a vertex v_i is adjacent to two unmatched vertices because, w.l.o.g., v_{j_2} would stay unmatched.

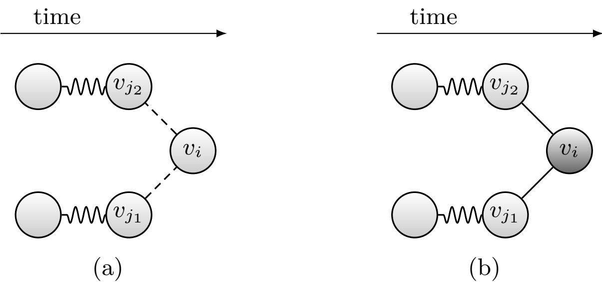


Figure 3.11. It cannot happen that both vertices v_{j_1} and v_{j_2} are matched in a previous time step since otherwise v_i would stay unmatched.

In all other cases, the algorithm reads a bit of advice to decide whether the new edges belong to the matching or not. The algorithm encounters one of the two situations (a) or (b) shown in Figure 3.12. If v_i has only one adjacent isolated vertex in time step i, the advice bit gives the algorithm the hint if the edge $\{v_i, v_j\}$ belongs to the matching or not. In the case with two adjacent isolated vertices v_{j_1} and v_{j_2}, the bit says whether the edge $\{v_{j_1}, v_i\}$ or the edge $\{v_{j_2}, v_i\}$ is part of the matching. The details of the algorithm are described in Algorithm 3.4. The case distinction is also visualized in Figure 3.12.

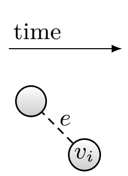
Algorithm 3.4 Maximum Matching on Paths

INPUT: $P^{\prec} \in \mathcal{P}_n$, for some $n \in \mathbb{N}$

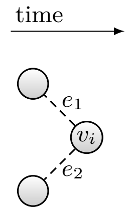
1: $M = \varnothing$
2: **for** $i = 1$ to n **do**
3: **if** (a) v_i is connected by exactly one edge e to a previously isolated vertex **then**
4: read an advice bit σ to decide whether e is a matching edge:
5: **if** $\sigma = 1$ **then**
6: $M \leftarrow M \cup \{e\}$
7: **else**
8: $M \leftarrow M$
9: **else if** (b) v_i has two edges e_1 and e_2 to previously isolated vertices **then**
10: use an advice bit σ to decide whether e_1 or e_2 is a matching edge:
11: **if** $\sigma = 0$ **then**
12: $M \leftarrow M \cup \{e_1\}$
13: **else**
14: $M \leftarrow M \cup \{e_2\}$
15: **else if** (c) v_i is connected to some non-isolated and unmatched vertex by an edge e **then**
16: $M \leftarrow M \cup \{e\}$
17: **else if** (d) v_i has an edge e_1 to a matched vertex and an edge e_2 to an isolated vertex **then**
18: $M \leftarrow M \cup \{e_2\}$
19: **else**
20: $M \leftarrow M$
21: **output** $M_i = M$

OUTPUT: $M = \{e_{i_1}, e_{i_2}, \ldots, e_{i_m}\} = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

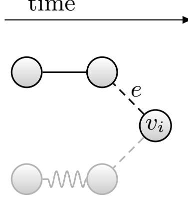
Now we have to estimate the number of advice bits needed in the worst case for an instance $P^{\prec} \in \mathcal{P}_n$ on n vertices. Only in the two cases (a) and (b) of Figure 3.12, the algorithm has to read advice for the vertex v_i. In all other cases, we showed above that no advice is needed since the decision on the new edges is given by the neighborhood of these edges. The subpaths of (a) and (b) in Figure 3.12 need to be separated by at least one vertex in order to ensure that every of these subpaths has



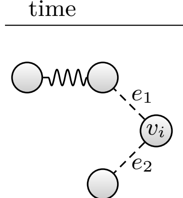
(a) One isolated previous neighbor (lines 3–8).



(b) Two isolated previous neighbors (lines 9–14).



(c) A non-isolated and an unmatched neighbor (lines 15–16).



(d) A matched and an isolated neighbor (lines 17–18).

Figure 3.12. The four cases in Algorithm 3.4.

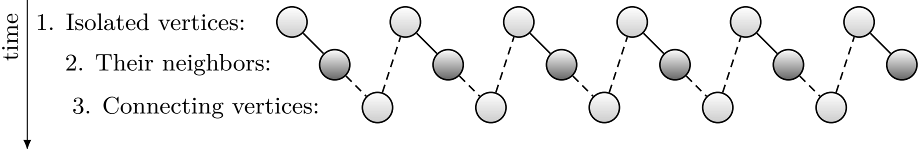


Figure 3.13. One worst-case instance with 17 vertices for the algorithm: In a first phase, the adversary gives some isolated vertices, then the dark vertices appear, which need to ask for an advice bit. In the end, the path will be connected with the remaining vertices.

to ask for an advice bit. In the worst case, the online path $P^{\prec}$ consists of subpaths of length 1 as shown in Figure 3.12 (a) and each two of those subpaths will be separated by a vertex (see Figure 3.13). In such online paths with n vertices, at most $\lceil \frac{n}{3} \rceil$ advice bits are needed, which concludes the proof. □

We can view Algorithm 3.4 as follows. Algorithm 3.4 asks advice for every new *vertex* if necessary and applies the answer of this bit to the adjacent edges. The intuition behind this algorithm is that the algorithm has no information from the neighborhood for *isolated edges* as in case (a) or for an *isolated path of length 2* that arises in time step i by connecting two isolated vertices by a new vertex v_i as in case (b). Therefore, the advice bits can also be directly associated with the edges. Then, case (c) describes a *greedy algorithm for all non-isolated edges*. The number of advice bits used by the algorithm depends therefore on the number of isolated edges or isolated subpaths of length 2. The worst case is achieved with

the maximum number of isolated edges, which is $\lceil \frac{n}{3} \rceil$ in a path on n vertices. We will use this intuition later in the discussion of tradeoffs between advice bits and the competitive ratio of the algorithm using this amount of advice bits.
We already observed that, in a path with an odd number of vertices, there are $\frac{n+1}{2}$ different maximum matchings. But only two of these matchings satisfy the property of Lemma 3.8, namely that all inner vertices are matched. If the algorithm would admit also maximum matchings that do not satisfy this property, we could save two additional bits of advice. The modified algorithm will not read any advice bit for the first and the second isolated edge, but it will decide to take them both into the matching. One of the optimal matchings contains these edges for sure and therefore the oracle can guide the algorithm to this optimal matching. Still, we have to modify Algorithm 3.4 slightly because the cases of Figures 3.10 and 3.11 can appear if we allow matchings that do not match all inner vertices. For the second case, we do not have to change the algorithm since any of the two edges will be taken into the matching. For the first case, we have to replace lines 15 to 16 by

else if (c') v_i has two edges e_1 and e_2 to two non-isolated and unmatched vertices **then**
  $M \leftarrow M \cup \{e_1\}$
else if (c") v_i is connected to one non-isolated and unmatched vertex by an edge e **then**
  $M \leftarrow M \cup \{e\}$

We call Algorithm 3.4 with the changes described above Algorithm 3.4* in the proof of the following theorem.

Theorem 3.17. *In the case of an online path $P^{\prec} \in \mathcal{P}_n$ with n odd, there is an algorithm which uses at most $\lceil \frac{n}{3} \rceil - 2$ bits of advice to find a maximum matching in $P^{\prec}$.*

Proof. We have to show that Algorithm 3.4* can complete the matching which contains the first two isolated edges of $P^{\prec} \in \mathcal{P}_n$ to a valid maximum matching. Let e_1 be the first and e_2 be the second of these isolated edges. Then there are two possibilities with respect to the number of edges in the final graph $P^{\prec}[V_n]$:

1. There is an odd number of edges between e_1 and e_2.

2. There is an even number of edges between e_1 and e_2.

Observe that, in every maximum matching, there is exactly one unmatched vertex. Since this vertex can be an arbitrary vertex of the $\frac{n+1}{2}$ vertices described in Lemma 3.7, there will always be a possibility to complete the matching where e_1 and e_2 are matching edges. In case 1, the unmatched vertex will be left of e_1 or right of e_2. Therefore the matching will look as described in Figure 3.14. Note that some of the vertices in the lower levels of Figures 3.14 and 3.15 can appear as isolated vertices earlier than e_1 and e_2 in the online setting of $P^{\prec}$. But for the sake of readability, all of the vertices are depicted in the lower level.

Figure 3.14. Odd number of edges between e_1 and e_2.

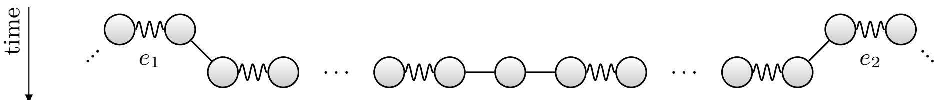


Figure 3.15. Even number of edges between e_1 and e_2.

In case 2, the unmatched vertex has to be somewhere in between e_1 and e_2 and the final maximum matching will look as shown in Figure 3.15.
Alltogether, there is always a possibility to finish the matching correctly in order to get a valid maximum matching. □

Observe that, in case 2 of the proof of Theorem 3.17, there are still several maximum matchings possible if there are more than 4 edges between e_1 and e_2. But trying to save an additional advice bit by forcing the algorithm also to take the third isolated edge into the matching will not work since, in the worst case, this would create two subpaths with an even number of vertices and we cannot afford a second unmatched vertex.
The upper bound in a cycle can be developed similarly. Let $C^{\prec} = (C_n, \prec)$ be an online cycle instance with C_n a cycle on n vertices. A maximum matching in C_n looks as shown in Figure 3.16. In the case of a cycle of even length, there are two edge-disjoint maximum matchings. If a cycle $C^{\prec} = (C_n, \prec)$ has an odd number n of

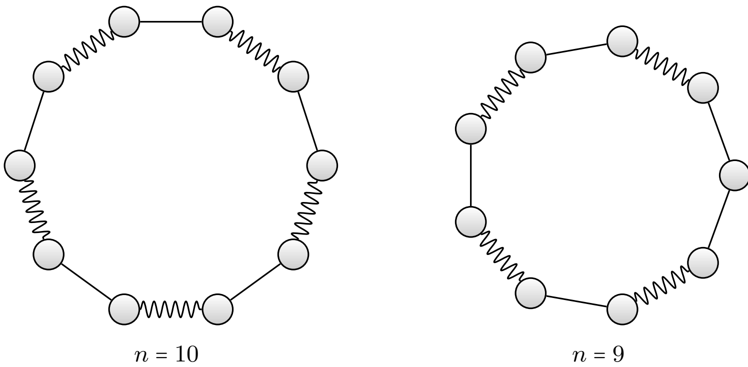


Figure 3.16. One of two possible maximum matchings for a cycle with an even number of vertices and one of several for a cycle with an odd number of vertices.

vertices, we have n maximum matchings, since the only unmatched vertex can be any from the whole set of vertices.

Theorem 3.18. *There is an online algorithm which uses at most* $\lfloor \frac{n}{3} \rfloor - 1$ *advice bits to find a maximum matching in a cycle* $C^{\prec} \in \mathcal{C}_n$ *of length* n*.*

Proof. In an online cycle $C^{\prec} \in \mathcal{C}_n$ with n vertices, maximally $\lfloor \frac{n}{3} \rfloor$ isolated edges of type (a) in Figure 3.12 can appear. The algorithm A can take the first isolated edge e to the matching since in both cases, n even and n odd, there exists a maximum matching containing e. Therefore, at most $\lfloor \frac{n}{3} \rfloor - 1$ isolated edges could be left which need an advice bit each. This leads to the upper bound of $\lfloor \frac{n}{3} \rfloor - 1$ for the algorithm A. □

In the case of a cycle of odd length, we have analogously to paths more maximum matchings than in the case of a cycles of even lengths. The unmatched vertex helps us again to save an additional bit.

Corollary 3.19. *For all online cycles* $C^{\prec} \in \mathcal{C}_n$ *with* n *odd, there exists an algorithm which uses at most* $\lfloor \frac{n}{3} \rfloor - 2$ *bits of advice to find a maximum matching in* $C^{\prec}$*.*

Proof. We modify the algorithm A from the proof of Theorem 3.18, analogously to the modification for the paths, in the way that also the second isolated edge will be taken into the matching. Then, the first two isolated edges e_1 and e_2 cut the cycle into two paths. One of these paths contains an even number of edges and the other one an odd number of edges since the number of edges without e_1 and e_2 is also odd and an odd number can only be the sum of an even and an odd number. In the path with an odd number of edges, the algorithm has to complete the matching as shown in Figure 3.14 and if there are an even number of edges in the path, the algorithm can follow one of the possible matchings according to Figure 3.15. Therefore, there is always a possibility for the algorithm to find, with the help of $\lfloor \frac{n}{3} \rfloor - 2$ advice bits, a valid maximum matching, given that the first two isolated edges e_1 and e_2 are matching edges. □

Now, we show that there exist almost matching lower bounds for the online matching problem on paths and cycles.

Theorem 3.20. *Any deterministic online algorithm for the problem of finding a maximum matching in an online path instance* $P^{\prec} \in \mathcal{P}_n$ *needs to read at least*

$$\left\lfloor \frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{2\pi}}\right) \right\rfloor$$

advice bits in order to be optimal.

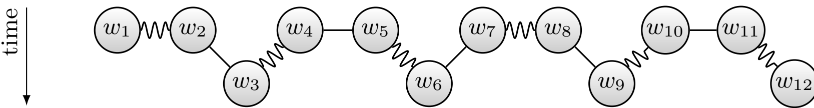


Figure 3.17. Structure of the online instances for the lower bound on advice bits used by every algorithm solving the online matching problem on an online path. In this example, we have $n = 12$ and therefore $k = 2$.

Proof. We give a proof by contradiction. The proof is based on the following idea. We assume that there exists an algorithm $A_<$ that uses less than

$$b(n) = \frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{2\pi}}\right)$$

advice bits to find a maximum matching in an online path instance with n vertices. Then, we describe a special set of pairwise different instances with pairwise different optimal solutions. The number of these instances will be at least $2^{b(n)}$. Therefore, by the pigeonhole principle, there exists a pair of instances such that the algorithm has to use the same advice string to solve them both optimally. Because the optimal solutions of these two instances differ, there is a smallest time step in which the algorithm should start to behave differently on these two instances. The set of instances has to be constructed in such a way that one is forced to behave differently already in the time when one cannot distinguish between them, because they have the same prefix. If the algorithm would do the same on both instances at this point, one of these instances would not be solved optimally. Therefore the algorithm cannot use the same advice string to solve them both optimally. This is achieved by choosing the problem instances in such a way that they all have a long common prefix and so any online algorithm can perform differently on this long prefix if and only if the advice is different.
For the proof, we will assume that $n = 6k$ is an even number such that all the instances contain exactly one maximum matching. The special instances all start with a common prefix, namely with $\frac{n}{3} = 2k$ isolated edges

$$\{\{w_{3i+1}, w_{3i+2}\} \mid i \in \{0, 1, 2, \ldots, 2k - 1\}\}$$

given in all possible orders. Note that each online algorithm has to decide immediately which of these edges are in the machting and which ones are not. In the second step, each two of those edges are connected by one of the vertices in $\{w_{3i} \mid i \in \{1, 2, \ldots, 2k\}\}$ in order to get a path (see Figure 3.17).
The described instances only differ in the presentation order of the edge set

$$\{\{w_{3i+1}, w_{3i+2}\} \mid i \in \{0, 1, 2, \ldots, 2k - 1\}\}$$

containing $2k$ vertices. Thereby, it does not matter, if the vertex w_{3i+1} or the vertex w_{3i+2} appears first. Note that half of these edges, namely the edge set

$\{\{w_1, w_2\}, \dots, \{w_{6k-5}, w_{6k-4}\}\}$ has to be a set of matching edges and the remaining edges $\{\{w_4, w_5\}, \dots, \{w_{6k-2}, w_{6k-1}\}\}$ are non-matching edges. The solution only depends on the order in which the matching and the non-matching edges appear. This divides the set of all instances that can be reached with the described construction in equivalence classes and for each of this classes, there is exactly one unique optimal solution. The unique solution ist determined by a string $x_1x_2x_3\dots x_{2k}$ with $x_i \in \{0,1\}$ for all $i \in \{1,2,\dots,2k\}$ with

$$x_i = \begin{cases} 1 & \text{if the } i\text{th isolated edge is a matching edge}\,, \\ 0 & \text{else}\,. \end{cases}$$

These strings contain exactly k zeros and k ones. This leads to $\binom{2k}{k}$ equivalence classes. With the help of Stirling's formula (1.2) and the inequality (1.3) of Chapter 1 we get

$$\binom{2k}{k} > \frac{1}{2} \cdot \frac{4^k}{\sqrt{\pi k}} = \frac{1}{2} \cdot \frac{4^{\frac{n}{6}}}{\sqrt{\pi \cdot \frac{n}{6}}} = \frac{1}{2} \cdot \frac{2^{\frac{n}{3}}}{\sqrt{\frac{\pi}{6}} \cdot \sqrt{n}}$$
$$= \sqrt{\frac{3}{2\pi}} \cdot \frac{2^{\frac{n}{3}}}{\sqrt{n}}\,,$$

for any $k = \frac{n}{6}$. We claimed that there exists an algorithm $A_<$ that uses less than

$$\log\left(\sqrt{\frac{3}{2\pi}} \cdot \frac{2^{\frac{n}{3}}}{\sqrt{n}}\right) = \frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{2\pi}}\right) < \log\binom{2k}{k}$$

advice bits. Therefore, there are two equivalence classes which get the same advice string. This means that the algorithm makes the same decisions on the first $\frac{n}{3}$ isolated edges on both instance classes. But since these equivalence classes are different, there is a first isolated edge e such that, w.l.o.g., e is a matching edge in the first instance but a non-matching edge in the second instance. Therefore, the matching chosen by $A_<$ will not be optimal in one of these instances which is a contradiction to the assumption.

Therefore, there is no online algorithm which can find an optimal matching in all online path instances with less than $\frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{2\pi}}\right)$ advice bits. □

The lower bound for online cycles $C^{\prec} = (C_n, \prec)$ can be derived similarly. The main difference is that both types of cycles, with n even or n odd, do not have a unique solution. Therefore, we have to pay a little bit more attention to the argumentation in the proof.

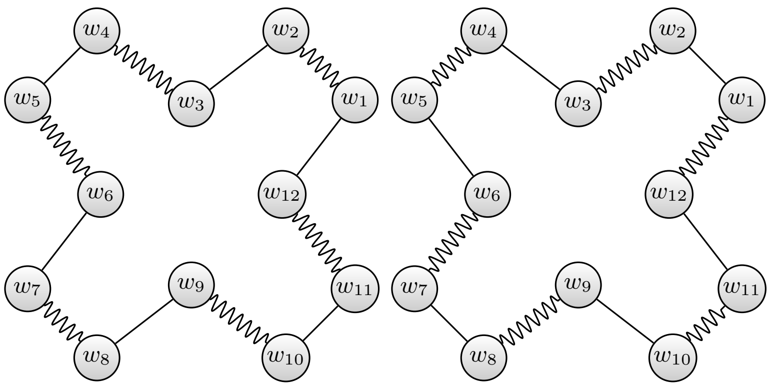


Figure 3.18. The two solutions of an online cycle instance with $n = 12$ ($k = 2$) for the lower bound on advice bits used by every algorithm solving the online matching problem on an online cycle.

Theorem 3.21. *Any deterministic online algorithm for the problem of finding a maximum matching in an online cycle instance $C^{\prec} \in \mathcal{C}_n$ needs to read at least*

$$\left\lfloor \frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{8\pi}}\right) \right\rfloor$$

advice bits in order to be optimal.

Proof. The instance class is defined similarly to the one in the proof of Theorem 3.20. Again, we assume that $n = 6k$. The adversary shows to the algorithm first the $\frac{n}{3} = 2k$ isolated edges $\{\{w_{3i+1}, w_{3i+2}\} \mid i \in \{0, 1, 2, \ldots, 2k-1\}\}$ in an arbitrary order and then the $2k$ connecting vertices $\{w_{3i} \mid i \in \{1, 2, \ldots, 2k\}\}$ which are depicted on an inner circle in Figure 3.18.
Note that every instance has exactly two optimal solutions. If the string $x_1x_2 \ldots x_{2k}$ with $x_i \in \{0, 1\}$ for all $i \in \{1, 2, \ldots, 2k\}$ with

$$x_i = \begin{cases} 1 & \text{if the } i\text{th isolated edge is a matching edge}, \\ 0 & \text{else} \end{cases}$$

is an optimal solution, also the string $\overline{x}_1\overline{x}_2\overline{x}_3 \ldots \overline{x}_{2k}$ with

$$\overline{x}_i = \begin{cases} 1 & \text{if } x_i = 0, \\ 0 & \text{if } x_i = 1 \end{cases}$$

is an optimal solution for the same instance. Therefore, there are $\frac{\binom{2k}{k}}{2}$ different equivalence classes. Adjusting the calculation of the proof of Theorem 3.20, we get

$$\frac{\binom{2k}{k}}{2} = \sqrt{\frac{3}{8\pi}} \cdot \frac{2^{\frac{n}{3}}}{\sqrt{n}}.$$

and therefore, with the same argumentation as in the proof of Theorem 3.20, we get the claimed lower bound of

$$\frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{8\pi}}\right) < \log\left(\frac{\binom{2k}{k}}{2}\right)$$

concluding the proof □

3.3 Lower Bound for Optimality in Trees

In order to determine the amount of advice bits needed solving the online matching problem on trees optimally, we will first show a lower bound on the number of advice bits that any online algorithm needs to read in order to find a maximum matching in trees. The described lower bound will have the special shape of combs. This is a special case of so-called caterpillars [66].

Definition 3.22 (Caterpillar, Comb). A *caterpillar* is a tree in which there exists a path (called the *spine*) that contains every vertex of degree two or more. A *comb* is a caterpillar in which all vertices on the spine have vertex degree 3, except the first and the last vertex which have degree 2.

See Figure 3.19 for an example of a caterpillar and a comb.

Theorem 3.23. *Any deterministic online algorithm for the problem of finding a maximum matching in an online comb instance on n vertices needs to read at least*

$$\frac{n}{2} - 1$$

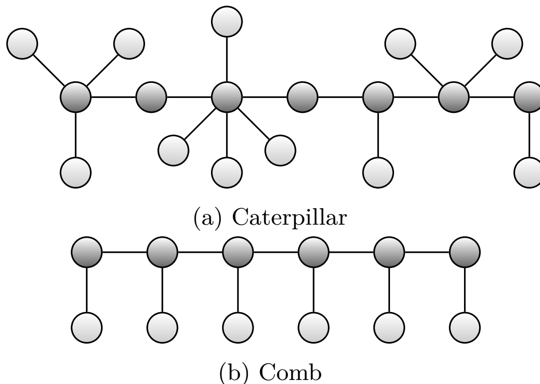
advice bits in order to be optimal.

Proof. Assume that there is an algorithm $A_<$ that needs less than $\frac{n}{2} - 1$ advice bits to be optimal for any online comb instance on n vertices. We need to describe a class of online instances in which each two of the instances need a different advice string. We show that this class consists of $2^{\frac{n}{2}-1}$ instances. This is a contradiction to the assumption and this provides us with the claimed lower bound.
Let $n = 2k + 2$ with $k \in \mathbb{N}$ be the number of vertices in the final comb. They are built according to the pattern described in Figure 3.20. Note that these instances all have a unique optimal solution. All the instances start with the vertex $v_1 = w_1$.

Then, for the vertex v_2 there are 2 possibilities: Either $v_2 = w_2^1$ or $v_2 = w_2^2$. According to the decision for v_2, we assign the remaining vertex of the vertex set $\{w_2^1, w_2^2\}$ to v_3. In general, for every $i \in \{1, 2, \ldots, k\}$, the vertex v_{2i} is either w_{i+1}^1 or w_{i+1}^2, and v_{2i+1} is the remaining of these two vertices.
So, the adversary has the chance to choose, in every time step $2i$ for all $i \in \{1, 2, \ldots, k\}$, either a matching or a non-matching edge to show to the algorithm. Therefore, there are

$$2^k = 2^{\frac{n}{2}-1}$$

many different instances.
It remains to show that no two of these online instances can get the same advice string in order to be optimal. These two instances have a prefix with undistinguishable information and differ, w.l.o.g., in time step $2i$ for the first time. W.l.o.g., one of the instances, say I_1, has $v_{2i} = w_{i+1}^1$ and $v_{2i+1} = w_{i+1}^2$ and the other instance I_2 has $v_{2i} = w_{i+1}^2$ and $v_{2i+1} = w_{i+1}^1$ (see Figure 3.21).



(a) Caterpillar

(b) Comb

Figure 3.19. Example of a caterpillar and a comb. The vertices on the spine are marked darker.

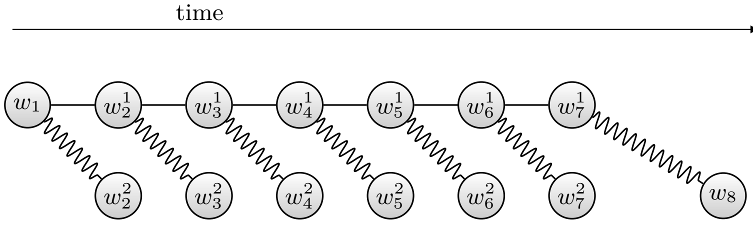


Figure 3.20. Pattern of the online combs for the proof of the lower bound on advice bits used by every algorithm solving the online matching problem on combs. In this example, we have $n = 14$ and $k = 6$. In time steps 2, 4, 6, 8, and 10, the adversary has the choice to reveal either an edge that should be a matching edge or a non-matching edge.

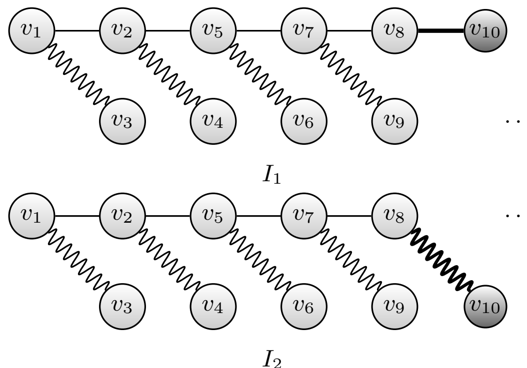


Figure 3.21. Two online instances that differ in time step 10 for the first time.

Since for I_1 and I_2 the same information is provided until time step $2i$, the algorithm has to decide for both instances I_1 and I_2 in the same way in time step $2i$. Therefore one of the two chosen matchings will not be optimal. Therefore, any algorithm has to use two different advice strings for each two instances in our class. This means that every algorithm needs to read at least $\frac{n}{2} - 1$ advice bits to be optimal. □

One special subclass of trees we will consider in the next section is the class of trees where we restrict the vertex degrees of the inner vertices. All inner vertices will have exactly the vertex degree 3. The instance class used in the proof of Theorem 3.23 does not satisfy this property. Therefore, we modify the instances slightly, but we have to accept a lower bound that is weaker by two advice bits.

Corollary 3.24. *Any deterministic online algorithm for the problem of finding a maximum matching in a tree on n vertices with vertex degrees 1 and 3 needs to read at least*

$$\frac{n}{2} - 3$$

advice bits in order to be optimal.

Proof. The only modification on the instance class of the proof of Theorem 3.23 is that we have to append to the leftmost vertex w_1 a vertex w^3_{k+2} with two neighbors w^1_{k+3} and w^2_{k+3} and to the rightmost vertex w^1_{k+1} of the spine the concluding vertex w^2_{k+2} for a $k \in \mathbb{N}$ (see Figure 3.22). The adversary shows the additional vertices w^3_{k+2}, w^1_{k+3}, w^2_{k+3}, and w^2_{k+2} in this order after the set of vertices described in Theorem 3.23 (actually, an arbitrary order would be sufficient, but it is easier to fix one). These additional vertices do not provide any additional choice to the algorithm but they assure that the resulting online graph is an online $\{1,3\}$-tree. In the new instance class, we have $n = 2k + 6$ with $k \in \mathbb{N}$ vertices in the final tree. Since the prefix of these instances is constructed in the same way as the one of Theorem 3.23 until time step $2k + 2$, again, we have 2^k different instances for

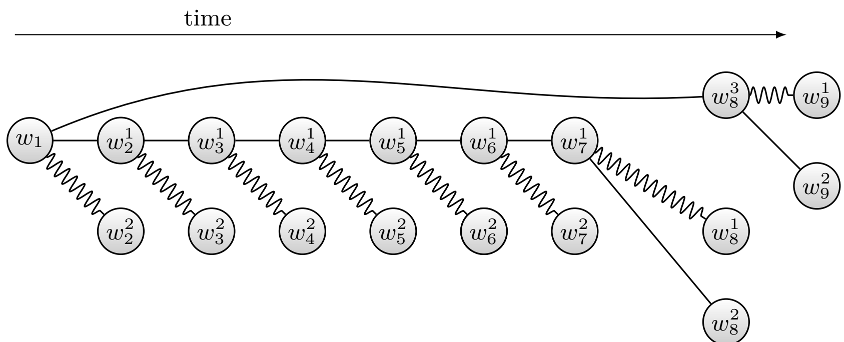


Figure 3.22. Pattern of the modified trees for the proof of the lower bound on advice bits used by every algorithm solving the online matching problem on $\{1,3\}$-trees. Here we have $n = 18$ and $k = 6$.

which each two have to receive a different advice string as shown in the proof of Theorem 3.23. Inserting $k = \frac{n-6}{2}$ to this term, we see that we have $2^{\frac{n-6}{2}} = 2^{\frac{n}{2}-3}$ different instances leading to $\frac{n}{2} - 3$ advice bits that are necessary to be optimal. □

3.4 Upper Bounds for Optimality in Trees

After giving a straight-forward upper bound on general trees, we restrict our attention to special subclasses of trees.

Theorem 3.25. *There is an online algorithm which reads at most $n - 1$ advice bits to solve the online online matching problem on trees T_n with n vertices.*

Proof. The algorithm A_{tree} asks for every new edge one bit of advice which indicates to the algorithm if the edge should be a matching or a non-matching edge. Since a tree T_n contains exactly $n - 1$ edges, the algorithm A_{tree} has to ask for maximally $n - 1$ bits. □

Trees With Restricted Vertex Degrees

In the following, we consider special graph classes with restricted vertex degrees.

Definition 3.26 (S-Tree). A tree T with vertex degrees from the set $S \subseteq \mathbb{N}$ is called an *S-tree.*

According to this definition, a path is a $\{1,2\}$-tree. As a next graph class, we will consider $\{1,3\}$-trees. This means that all inner vertices of the trees have exactly degree 3. Therefore, these trees are similar to binary trees. Let us first observe some special properties of (offline) $\{1,3\}$-trees.

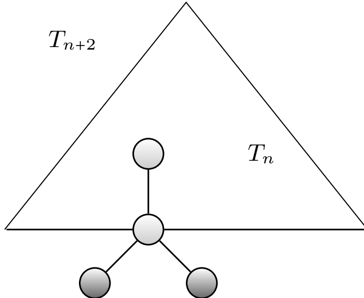


Figure 3.23. To construct T_{n+2}, we attach two vertices to one vertex of degree 1 in T_n.

Lemma 3.27. *The number of vertices in $\{1,3\}$-trees with more than one vertex is even.*

Proof. For a tree $T = (V, E)$ we have

$$\sum_{v \in V(T)} \deg(v) = 2 \cdot |E(T)|,$$

i. e., the sum over all vertex degrees is even. Because all the vertex degrees are odd, only an even number of summands can lead to an even number. Therefore the number of vertices is even. □

To count the number of leaves in $\{1,3\}$-trees, we need to know how all $\{1,3\}$-trees can be constructed.

Lemma 3.28. *Every $\{1,3\}$-tree T_n on $n = 2k$ vertices, for $k \in \mathbb{N}$, $k > 1$, can be constructed recursively by the following two rules:*

1. *For $k = 1$, T_2 is a single edge.*
2. *Let $n = 2k$ for a $k \in \mathbb{N}$, $k > 1$. We construct a $\{1,3\}$-tree T_{n+2} by attaching two vertices to one vertex of degree 1 in some $\{1,3\}$-tree T_n (see Figure 3.23).*

Proof. We will prove this lemma by contradiction. Assume that there exists a smallest $\{1,3\}$-tree T_{2k+2} for some $k \in \mathbb{N}$ that cannot be constructed following these two rules. Then k has to be larger than 0 since a $\{1,3\}$-tree on 2 vertices always has to follow rule 1. The deviation from the construction rules has to have happened in the last step since we deal with the smallest $\{1,3\}$-tree which does not follow these rules. Therefore, the construction of T_{2k+2} from T_{2k} happened in a different way than suggested in rule 2 and applying rule 2 on any T_{2k} cannot result in T_{2k+2}.
Rule 2 could have been applied if there would exist two leaves in T_{2k+2} which would be attached to the same vertex as shown in Figure 3.23. Since this rule was not

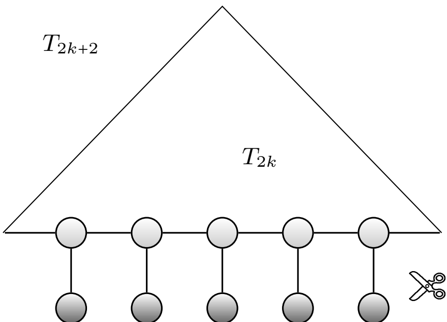


Figure 3.24. No two leaves in T_{2k+2} are attached to the same inner vertex.

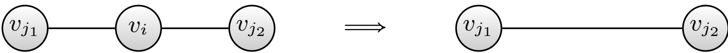


Figure 3.25. All vertices v_i of degree 2 in T' are removed in order to get a regular graph T'' with degree 3.

applicable, T_{2k+2} contains only leaves that are pairwise adjacent to different inner vertices (see Figure 3.24). Since T_{2k+2} is a $\{1,3\}$-tree, removing all leaves (i. e., all vertices of degree 1) results in a tree T' with vertices of degrees 2 and 3. Note that the vertices that are adjacent to the removed leaves all have degree 3 and therefore they do not become leaves after removing their leaves. We want to show that T' contains a cycle which is a contradiction to T_{2k+2} being a tree.
Therefore, we transform T' to a graph T'' of containing only vertices of degree 3 by removing all vertices v_i of degrees two and adding edges v_{j_1} and v_{j_2} to T'' if and only if $\{v_{j_1}, v_i\} \in E(T')$ and $\{v_{j_2}, v_i\} \in E(T')$ (see Figure 3.25). Note that, due to the above construction, if there is a cycle in T'', there is also one in T' and therefore in T_{2k+2}. Since T'' is a 3-regular graph, it has to contain a cycle. Hence, our assumption was wrong, and every graph T_{2k+2} can be constructed following rules 1 and 2 of the lemma. □

In the proof, we used the folklore that every 3-regular (cubic) graph contains a cycle[1].

Lemma 3.29. *Any $\{1,3\}$-tree T_n with n vertices contains exactly $\frac{n}{2}+1$ leaves.*

Proof. This can be proven by induction over the number of vertices. Because of Lemma 3.27, we know that the number of vertices in a $\{1,3\}$-tree T_n is even, i. e., $n = 2k$ for some $k \in \mathbb{N}$.
The only tree with two vertices, i. e., for $k = 1$, is an isolated edge which contains two leaves. Because $\frac{n}{2} + 1 = 2$, the base case is proven.

[1] This holds because the depth-first search starting from an arbitrary vertex v will lead back to v at the latest when all other vertices have been visited. Therefore, this graph has to contain a cycle.

Assume that any $\{1,3\}$-tree T_{2k} on $n = 2k$ vertices contains exactly $\frac{n}{2}+1 = k+1$ leaves. Now, let us look at the case $k+1$, i. e., the number of vertices is $n = 2(k+1) = 2k+2$. Because of Lemma 3.28, any $\{1,3\}$-tree T_{2k+2} on $2k+2$ vertices can be constructed from some $\{1,3\}$-tree T_{2k} on $2k$ vertices by attaching two new vertices to some leaf of T_{2k}.
With the described construction, T_{2k+2} gets two new leaves, but one old leaf disappears. Therefore T_{2k+2} has one leaf more than T_{2k}. This implies that T_{2k+2} contains $(k+1)+1 = k+2 = \frac{n}{2}+2 = \frac{n+1}{2}+1$ leaves concluding the proof. □

A Special Maximum Matching

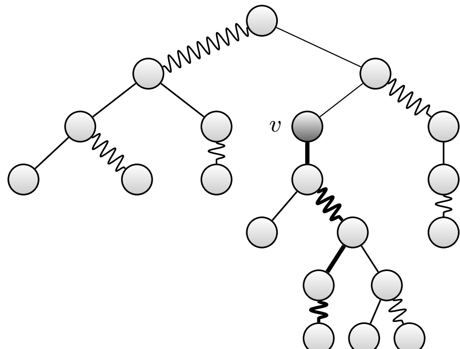
Sometimes it is easier to try to find a maximum matching with a special property. In Lemma 3.8, we showed that in paths, there is always a maximum matching that matches all inner vertices. Also for trees a matching which matches all inner vertices can be easier to find.

Lemma 3.30. *There exists always a maximum matching for a general tree T that matches all inner vertices.*

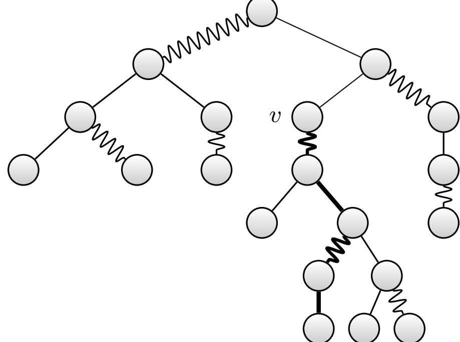
Proof. Suppose that, for a maximum matching M, there exist inner vertices that are not incident to a matching edge. First, consider one of the unmatched inner vertices v with the smallest distance to a leaf. Since no two neighboring edges can be both matching edges and v is an unmatched vertex that is nearest to this leaf, there must exist an alternating path between v and a leaf (see Figure 3.26). Since M is a maximum matching, this alternating path ends with a matching edge. If this edge would not be a matching edge, we would have an augmenting path which is a contradiction to the fact that M is a maximum matching.
Now, we can switch the matching and non-matching edges on this path such that v will be matched in the end.
Then, we can proceed in the same way with a next unmatched inner vertex until all inner vertices are matched. Obviously, this procedure terminates in finitely many steps because we do not create new unmatched inner vertices when switching matching and non-matching edges on the alternating paths. □

One First Algorithm for Finding a Maximum Matching in {1,3}-Trees

Now, we are ready to look at two algorithms proving upper bounds on the number of advice bits needed to find a maximum matching in an online $\{1,3\}$-tree. We will explain both algorithms, estimate their advice complexity and show their limitations. Both algorithms will look for a matching that matches all inner vertices.
The first algorithm is similar to Algorithm 3.4 for finding a maximum matching on paths. The algorithm reads an advice bit for every edge if necessary. The



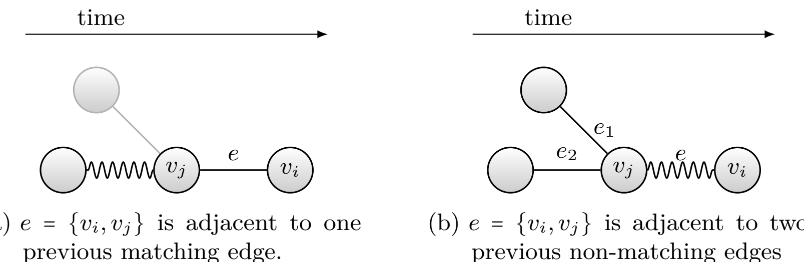
(a) The marked vertex is unmatched and the marked path is an alternating path.



(b) All inner vertices are matched by this matching.

Figure 3.26. The transformation of a matching that does not match an inner vertex to one which matches all inner vertices.

algorithm does not have to use any advice bit for a new edge e if it is adjacent to a previous matching edge or to two non-matching edges (see Figure 3.27) because the algorithm seeks a matching that matches all inner vertices. In the first case, e will be a non-matching edge and in the second case, e has to be a matching edge. To describe the algorithm in more detail, we need an order on the edges. Recall, that we define this order in two steps. First, we order the edges with respect to the appearance of the vertices v_i in the online graph $G^{\prec}$ given by $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$. We denote by E_i the set of edges that appear in time step i by adding the vertex v_i, i. e., $E_i = \{\{v_i, v_j\} \in E \mid j < i\}$ for all $i \in \{1, 2, \ldots, n\}$. Within a set E_i, $|E_i| > 1$, the order of edges is given by the order, in which two previous adjacent vertices v_j, v_k $(j, k < i)$ appear (see Definition 3.9 for more details).
Now, we are ready to give the details of the first algorithm for finding a maximum matching in a $\{1, 3\}$-tree which are described in Algorithm 3.5 (see also Figure 3.27).



(a) $e = \{v_i, v_j\}$ is adjacent to one previous matching edge.

(b) $e = \{v_i, v_j\}$ is adjacent to two previous non-matching edges

Figure 3.27. The two cases where the Algorithm 3.5 does not need to read an advice bit.

Algorithm 3.5 Maximum Matching on $\{1, 3\}$-trees

INPUT: $T^{\prec} \in \mathcal{T}_n$ with $T^{\prec}$ a $\{1, 3\}$-tree, for some $n \in \mathbb{N}$

```
 1: M = ∅
 2: for i = 1 to n do
 3:     if E_i = ∅ then
 4:         M ← M
 5:     else
 6:         for all e ∈ E_i do
 7:             if (a) e is adjacent to a matching edge then
 8:                 M ← M
 9:             else if (b) e is adjacent to e_1, e_2 ∈ E_i ∖ M with e_1, e_2 ≺ e then
10:                 M ← M ∪ {e}
11:             else
12:                 read an advice bit σ to decide whether e is a matching edge:
13:                 if σ = 1 then
14:                     M ← M ∪ {e}
15:                 else
16:                     M ← M
17:     output M_i = M
```

OUTPUT: $M = (e_{i_1}, e_{i_2}, \ldots, e_{i_m}) = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

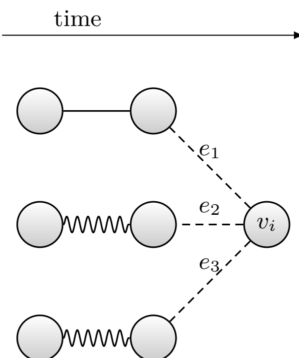


Figure 3.28. If the algorithm looks at all 3 edges at once, it can see that e_1 should be a matching edge.

In the ith run of the loop, Algorithm 3.5 looks at every edge $e \in E_i$ one by one according to their order. But sometimes it could save an advice bit by looking at the edges of E_i at once because there could be some useful information for the other edges. In the case shown in Figure 3.28, the algorithm would ask an advice bit for the edge e_1. But, because e_2 and e_3 have to be non-matching edges and the algorithm constructs a matching which matches all inner vertices, e_1 has to be the matching edge. Therefore, we could modify the algorithm by taking this case into account. We decided not to do so because this case does not appear in any worst-case instance and therefore cannot be used to improve the estimation of the upper bound.

Recall that the lower bound on the number of advice bits for an algorithm finding a maximum matching in a $\{1,3\}$-tree is $\frac{n}{2} - 3$ as we have proven in Corollary 3.24. Our goal is to find a matching upper bound for $\{1,3\}$-trees. We show that this algorithm will not satisfy our needs. First, we estimate the amount of advice used by the algorithm and then, we will give a class of online $\{1,3\}$-trees for which the algorithm uses a lot of advice bits.

A local view on the work of Algorithm 3.5 will help us to estimate the upper bound.

Lemma 3.31. *Let $T^{\prec}$ with $T = (V, E)$ and $V = \{v_1, v_2, \ldots, v_n\}$ be an online $\{1,3\}$-tree. For every vertex $v_i \in V$ of degree 3, Algorithm 3.5 reads an advice bit for at most two of the three incident edges.*

Proof. The edges incident to the vertex v_i appear either in time step i or later. There is an order on these three edges. Say, the edges e_1 and e_2 are already present and the edge e appears as the last incident edge to v_i. Then, we encounter one of the two cases in Figure 3.27. Either one of e_1 and e_2 is a matching edge and therefore e is non-matching or neither e_1 nor e_2 are matching and hence e is

matching since the algorithms seeks a matching which matches all inner vertices. Therefore, the third incident edge e never will ask for an advice bit. □

Lemma 3.31 helps us to estimate the number of advice bits that Algorithm 3.5 needs to read in order to find a maximum matching in an arbitrary $\{1,3\}$-tree.

Theorem 3.32. *Algorithm 3.5 uses at most*

$$\left\lceil \frac{3}{4}n - \frac{1}{2} \right\rceil$$

advice bits to find a maximum matching in an online $\{1,3\}$-tree $T^{\prec} \in \mathcal{T}_n$ on n vertices.

Proof. Algorithm 3.5 asks for every edge either for one bit of advice or for no bit of advice. To make the calculation easier, we copy, for every edge $e = \{v_i, v_j\}$, this bit onto the two incident vertices v_i and v_j. Because of Lemma 3.31, we know that every vertex of degree 3 is labeled with at most two advice bits. Furthermore, every leaf will have at most 1 advice bit in the worst case.
Now, we sum up all the advice bits of the n vertices. Note that we count every bit twice, since we copied the advice bit of every edge to the two incident vertices. In Lemma 3.29, we showed that there are exactly $\frac{n}{2} + 1$ leaves in a $\{1,3\}$-tree. Therefore, Algorithm 3.5 uses at most

$$\frac{\left(\frac{n}{2}+1\right)\cdot 1 + \left(\frac{n}{2}-1\right)\cdot 2}{2} = \frac{\frac{3n}{2}-1}{2} = \frac{3}{4}n - \frac{1}{2}$$

advice bits to find a maximum matching in any online $\{1,3\}$-tree of size n. □

In a next step, we show that Algorithm 3.5 cannot reach our desired bound of $\frac{n}{2} - 1$ advice bits for a $\{1,3\}$-tree of size n. Therefore, we need an instance class of $\{1,3\}$-trees where the algorithm always has to use a certain amount of advice bits to find a maximum matching in these graphs.

Theorem 3.33. *Algorithm 3.5 reads at least $\lfloor \frac{7}{12}n - \frac{1}{2} \rfloor = \lfloor 0.58\overline{3}n - 0.5 \rfloor$ advice bits in order to find a maximum matching in every $\{1,3\}$-tree of size n.*

Proof. To show the desired lower bound on the algorithm, we need to describe an instance class such that, for infinitely many n, there exists an instance of size n for which the algorithm reads at least $\lfloor \frac{7}{12}n - \frac{1}{2} \rfloor$ bits in order to be optimal.
These instances will repeatedly use the two subgraphs $C_1^{\prec}$ and $C_2^{\prec}$ described in Figure 3.29. The order of the vertices in the components $C_1^{\prec}$ and $C_2^{\prec}$ is given according to the index of the vertices. In both components, the algorithm will ask for an advice bit for the edges $\{v_1, v_2\}$, $\{v_1, v_3\}$ and $\{v_2, v_4\}$, and additionally one bit is required for the edge $\{v_3, v_6\}$ in $C_2^{\prec}$, so three advice bits are needed in component $C_1^{\prec}$ and four in component $C_2^{\prec}$. It does not matter whether the edge

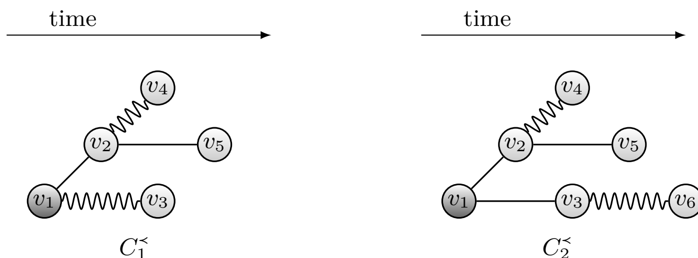


Figure 3.29. The two components $C_1^{\prec}$ and $C_2^{\prec}$ that are repeatedly used in the worst-case instances for Algorithm 3.5. The marked vertices will be the connecting vertices.

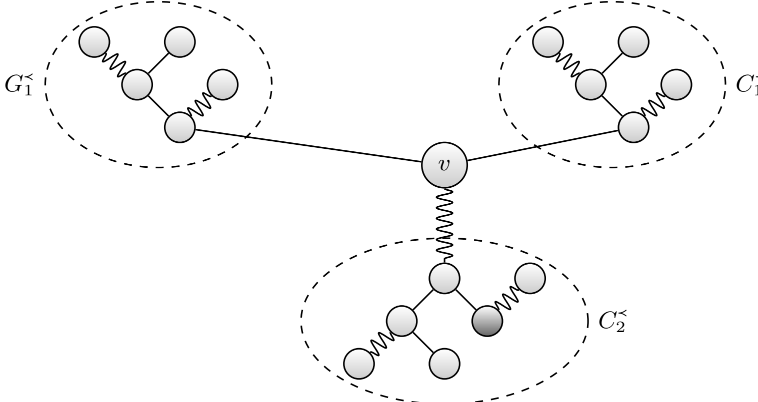


Figure 3.30. The graph $G_2^{\prec}$ is built by attaching a copy of $C_1^{\prec}$ and one of $C_2^{\prec}$ with a new vertex v to $G_1^{\prec}$. The marked vertex will be the connecting vertex for further components.

$\{v_2, v_4\}$ or the edge $\{v_2, v_5\}$ is a matching edge, but, as we will see later, all other edges have to be matching or non-matching edges as shown in Figure 3.29.
Now we are ready to build the instances step by step. Let the first subgraph $G_1^{\prec}$ be the component $C_1^{\prec}$. Note that the algorithm asks for 3 advice bits in $G_1^{\prec}$. Then, in every following stage k, an independently built copy of $C_1^{\prec}$ and one of $C_2^{\prec}$ will be attached to the existing subgraph $G_k^{\prec}$ by a vertex v adjacent to the marked vertices as shown in Figure 3.30 for the graph $G_2^{\prec}$. Therefore, the algorithm will not ask for any advice bit for the three new edges incident to v. Hence, with the additional 7 advice bits, the algorithm asks in total for 10 advice bits in $G_2^{\prec}$. The new connecting vertex is the vertex v_3 of the component $C_2^{\prec}$, which is the marked vertex in Figure 3.30.
To conclude the construction of the graphs $G_k^{\prec}$ to the final graphs $\tilde{G}_k^{\prec}$, for all $k \in \mathbb{N}$, we need to add an additional vertex adjacent to the last marked vertex in $G_k^{\prec}$ in order to build a $\{1,3\}$-tree. Otherwise, the last marked vertex would have degree 2. An example of a final graph $\tilde{G}_3^{\prec}$ is shown in Figure 3.31.

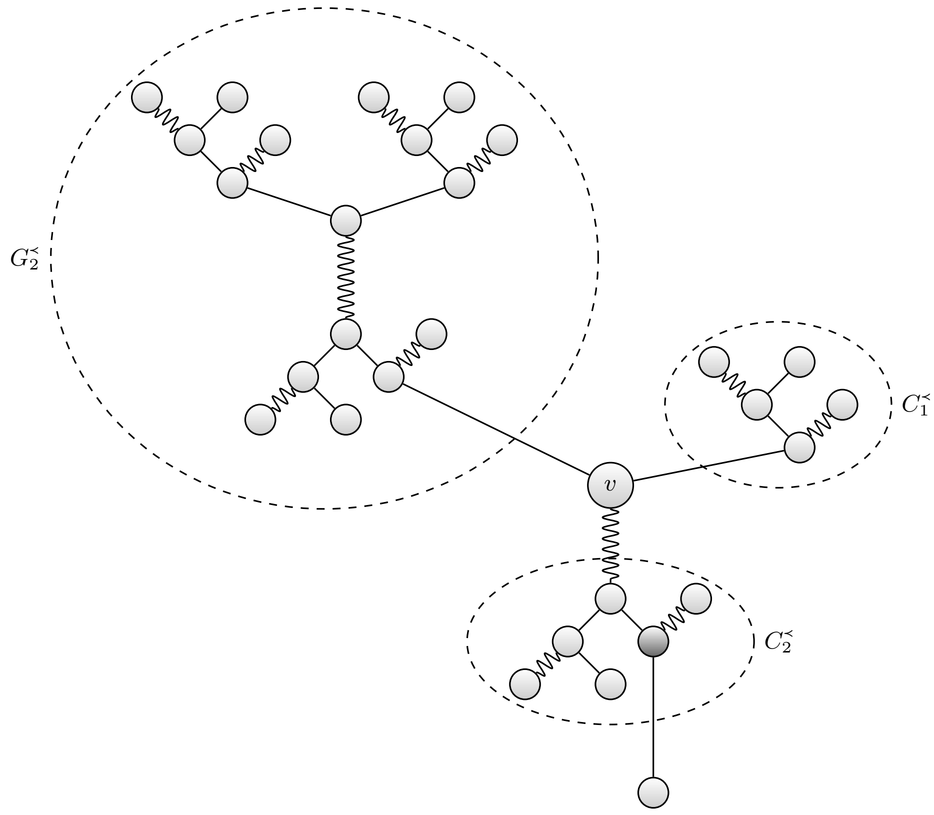


Figure 3.31. An example of a final graph $\tilde{G}_k^{\prec}$ for $k = 3$.

The number n of vertices in stage k, in the final graph $\tilde{G}_k^{\prec}$, is

$$\begin{aligned} n = V(\tilde{G}_k^{\prec}) &= V(G_1^{\prec}) + (k-1) \cdot \big(V(C_1^{\prec}) + V(C_2^{\prec}) + 1\big) + 1 \\ &= 5 + (k-1) \cdot (5 + 6 + 1) + 1 = 6 + (k-1) \cdot 12 \\ &= 12k - 6\,. \end{aligned}$$

The number of advice bits needed by the algorithm in order to find a maximum matching in $\tilde{G}_k^{\prec}$ is

$$\begin{aligned} 3 + (k-1) \cdot (3+4) = 3 + (k-1) \cdot 7 &= 7k - 4 \\ &= 7 \cdot \frac{n+6}{12} - 4 = \frac{7n+42}{12} - \frac{48}{12} \\ &= \frac{7}{12} n - \frac{1}{2} \end{aligned}$$

using the fact that $k = \frac{n+6}{12}$.

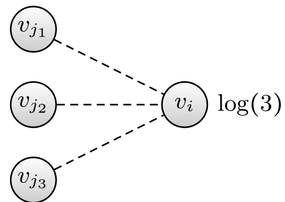
It remains to show that there is no maximum matching for which the algorithm would need less bits to find it. These instances are constructed such that all maximum matchings differ only in the choice whether the edge $\{v_2, v_4\}$ or the edge $\{v_2, v_5\}$ in each component $C_1^{\prec}$ and $C_2^{\prec}$ is a matching edge and whether the edge $\{v_3, v_6\}$ of the last constructed component $C_2^{\prec}$ or the last concluding edge is a matching edge. No matter which of these edges is chosen as matching edge, the algorithm has to ask one advice bit for the first edge of each of these pairs of edges. The matching of the inner edges is the same in every maximum matching. Therefore, for each of these maximum matchings, the algorithm has to consider the same number of advice bits. □

An Improved Algorithm for Finding a Maximum Matching in {1,3}-Trees

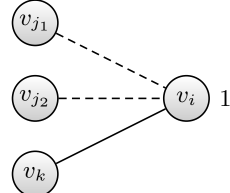
We now know the range of advice bits used by Algorithm 3.5 in order to find a maximum matching in any $\{1,3\}$-tree. This algorithm asks advice bits for every edge, if needed. But in some cases this may be too much. For example, for a vertex v_i of degree 3 in the ith run of the loop for a $\{1,3\}$-tree, Algorithm 3.5 asks two advice bits for the three incident edges in the worst case. But, because only one of these incident edges can be a matching edge, this is a one-out-of-three choice and therefore we would only need $\log(3)$ advice bits to encode the choice of the matching edge from the three incident edges to v_i. If there already exists a non-matching edge $\{v_i, v_k\}$ incident to v_i, for a $k < i$, even one bit of advice suffices to decide which one of the two new edges, $\{v_i, v_{j_1}\}$ or $\{v_i, v_{j_2}\}$, should be a matching edge (see Figure 3.32). The second modification does not improve the previous algorithm with respect to the number of advice bits used, but the advice bit is used slightly different. The first algorithm uses this bit to decide whether the second edge incident to v_i is a matching edge and the second algorithm decides with the help of this bit whether the second or the third edge is a matching edge. Since both algorithms are looking for an algorithm that matches all inner vertices, these two possibilities are the same.

So, the main modification that we undertake to find a better algorithm is, that we ask $\log(3)$ advice bits instead of 1 advice bit for an isolated edge $\{v_i, v_j\}$. These $\log(3)$ advice bits encode the one-out-of-three decision for the three edges incident to v_i if the next adjacent edge is an edge $\{v_i, v_k\}$ for a $k > i$ (see Figure 3.33). Note that if v_j gets additional adjacent vertices in a later time step, the algorithm has to ask an advice bit for the one-out-of-two decision for these two new edges.

Another new feature of the new algorithm is that the advice bits asked by it will always be assigned to a vertex and not to an edge as in the previous algorithm. This results in the problem that we have to define for which vertices the new algorithm should ask for advice in time step i. This will depend on the neighborhood of the new vertex v_i in the ith run of the loop.



(a) One-out-of-three decision for the three new edges incident to v_i.



(b) One-out-of-two decision for the two new incident edges of v_i.

Figure 3.32. The two possible decisions that have to be made in order to find the matching edge. $\{v_k, v_i\}$ for $k < i$ is a non-matching edge.

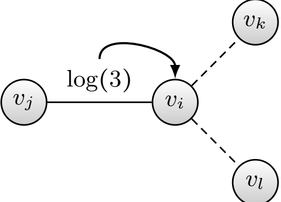


Figure 3.33. If the algorithm asks log(3) advice bits for an edge $\{v_i, v_j\}$ in time step i, these bits are mapped, w.l.o.g., to the vertex v_i with the next incident edge in the online presentation of the tree for $i, j < k$ and $i, j < l$.

Before describing the details of the new algorithm, we have to take care about what it means to read log(3) advice bits because a fraction of advice bits cannot be read. We take three possibilities into account:

1. One possibility would be to round the log(3) advice bits up to 2 advice bits, but then we would not save anything with respect to Algorithm 3.5.

2. The second possibility is to use the more elaborate idea of [58] to ask the oracle what we should make in the case of a one-out-of-three decision.

   Lemma 3.34 (Seibert et al. [58]). *Reading several one-out-of-three decisions from a bit string can be done with* $\frac{46}{29} < 1.5863$ *advice bits on average.* □

   In the proof of Lemma 3.34 it is described that, when the first one-out-of-three decision is necessary, the algorithm reads 46 bits from the advice string. This is far too much, but these 46 bits encode 29 one-out-of-three decisions. So the algorithm always asks 29 one-out-of-three decisions at the time and keeps them in its memory. This means that in the worst case, the algorihm has an overhead of 44 advice bits in the end which it cannot use. These small blocks of advice bits that encode one-out-of three decisions are good for instances on which not many one-out-of-three decisions have to be made.

3. If the algorithm has to make many one-out-of-three decisions, it would be better to ask the oracle in advance for the number k of one-out-of-three decisions that are necessary on the online $\{1,3\}$-tree. This costs the algorithm $\mathcal{O}(\log(k))$ advice bits. But then, the oracle can encode all of these decisions in one advice block in order not to have a big overflow of advice bits:

   Lemma 3.35. *When k is known to the algorithm, encoding k one-out-of-three decisions costs $\lfloor k\log(3)\rfloor + 1$ advice bits.*

   Proof. Let k be the number of one-out-of-three decisions that an algorithm has to make on an online $\{1,3\}$-tree. Hence, the algorithm has to choose one out of the 3^k possible sequences of k one-out-of-three decisions. We want to know how many advice bits are necessary to encode these 3^k sequences. So, we need to find a length l of the advice string such that

$$3^k \leq 2^l$$

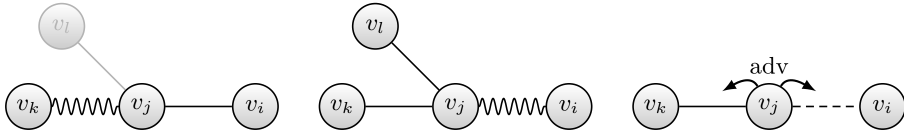
   holds. Applying the logarithm with basis two on both sides leads to $k\log(3) \leq l$ and therefore to $l = \lfloor k\log(3)\rfloor + 1$. □

   We will use this third possibility in our calculation.

To give the details of an improved algorithm, we need to distinguish some special edges which do not need any advice because it is clear from their neighborhood what decision has to be made for this edge. Note that our new algorithm will also match all inner vertices as the previous one. There are two possibilities where the decision for an edge $\{v_i, v_j\}$ in a time step i is already made because of the previous neighborhood:

- Either the adjacent edges give some information about the new edge e,
- or the algorithm already asked advice for the vertex v_j in an earlier time step.

In both cases, the algorithm does not need any additional advice to decide optimally on the edge e.



(a) v_j is incident to a matching edge.
(b) v_j is incident to two non-matching edges.
(c) The algorithm already read advice bits for v_j.

Figure 3.34. The edge $\{v_i, v_j\}$ is in the above cases saturated in time step i for $j, k, l < i$.

Definition 3.36 (Saturated Edge). Let $T^{\prec}$ be an online tree. In time step i, a new edge $e = \{v_i, v_j\}$, for a $j < i$, is called a *saturated edge* if (see Figure 3.34)

a) either v_j is incident to a previous matching edge $\{v_j, v_k\}$ for a $k < i$,

b) or v_j is incident to two previous non-matching edges $\{v_j, v_k\}$ and $\{v_j, v_l\}$ for $k, l < i$,

c) or the algorithm already read advice bits for the vertex v_j.

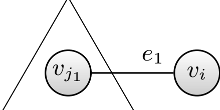
We say in cases (a) and (b) that e is *saturated by the edge* $\{v_j, v_k\}$, or *by the edges* $\{v_j, v_k\}$ and $\{v_j, v_l\}$ respectively, and in case (c), e is *saturated by the vertex* v_j.

Now we want to show that the algorithm does not need any additional advice for deciding if a saturated edge e is a matching edge or not.

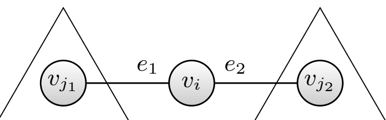
Lemma 3.37. *The algorithm can make an optimal decision on saturated edges.*

Proof. In the case of adjacency to a matching edge, $e = \{v_i, v_j\}$ is non-matching for sure. In case b), the new edge e has to be a matching edge since our algorithm looks for a matching that matches all inner vertices. In the last case, the algorithm already asked advice bits for e in an earlier time step and therefore the decision has been already made. □

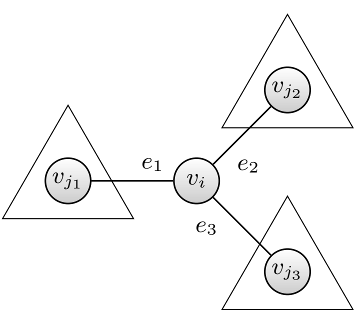
Note that the algorithm can exploit more benefits: If a non-saturated edge e is incident to exactly one previous saturated edge, a one-out-of-two decision is sufficient to make a decision on e as shown in Figure 3.32 (b). If e is adjacent to exactly two previous saturated edges, the decision of e being a matching or a non-matching edge is given implicitely by the two other edges.
To describe the details of the algorithm, we have to distinguish three cases with respect to the neighborhood a new vertex v_i encounters in time step i. The vertex v_i has either 1, 2 or 3 neighbors in time step i as described in Figure 3.35. Note that a new vertex v_i can never be connected to two vertices in the same subtree because



v_i is adjacent to one subtree.



v_i is adjacent to two subtrees.



v_i is adjacent to three subtrees.

Figure 3.35. The three possibilities for the neighborhood of a vertex v_i in time step i for $j_1, j_2, j_3 < i$.

this would invoke a cycle. The subtrees to which v_i is adjacent in time step i can either contain more than one vertex or just be a single vertex.
The new algorithm can ask for advice in every time step i and assigns the advice bits to a vertex which not always has to be the new vertex. The decision if an advice bit should be read is based on the neighborhood of v_i the algorithm encounters in step i. All the details are formalized in Algorithm 3.6.
Now we want to estimate the number of advice bits Algorithm 3.6 uses in order to find a maximum matching in an arbitrary $\{1,3\}$-tree. Unfortunately, we do not reach the desired bound of $\frac{n}{2} - 3$ but we come closer to it as with Algorithm 3.5.

Theorem 3.38. *Algorithm 3.6 needs to ask at most*

$$\frac{\log(3)}{3} n + \mathcal{O}(\log(n)) = 0.52832n + \mathcal{O}(\log(n))$$

advice bits in order to find a maximum matching in an online $\{1,3\}$-tree $T^{\prec} \in \mathcal{T}_n$.

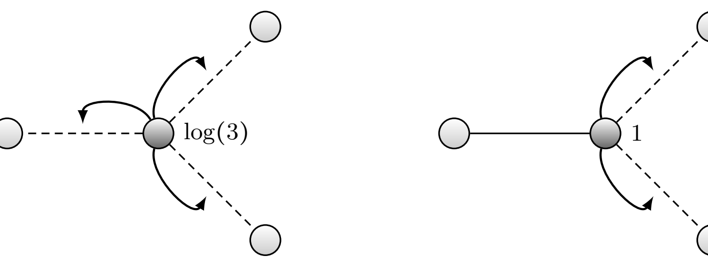
Proof. We relate the advice asked by Algorithm 3.6 always to the vertices. There are three possibilities for the vertices: Either a vertex is labeled with advice $\log(3)$, with advice 1, or with advice 0 if no decision is to be made for this vertex, respectively its incident edges.
To count the number of advice bits Algorithm 3.6 needs, we have to take care of the vertices with a positive amount of assigned advice bits. We will distribute these advice bits to the edges that are saturated by this vertex. Note that, due to the

Algorithm 3.6 Improved Maximum Matching on $\{1,3\}$-trees

INPUT: $T^{\prec} \in \mathcal{T}_n$ with $T^{\prec}$ an online $\{1,3\}$-tree, for some $n \in \mathbb{N}$

1: $M = \varnothing$
2: **for** $i = 1$ to n **do**
3:     **if** $E_i = \varnothing$ **then**
4:         $M \leftarrow M$
5:     **else if** $E_i = \{e_1\}$ **then**
6:         **if** $e_1 = \{v_i, v_j\}$ is isolated **then**
7:             make a one-out-of-three-decision accounting for the incident vertex receiving the next incident edge and update M according to the decision
8:         **else if** e_1 is saturated **then**
9:             make the decision according to the neighboring edges or the advice bits of v_j and update M accordingly
10:         **else**
11:             make a one-out-of-two-decision for vertex v_i, update M accordingly
12:     **else if** $E_i = \{e_1, e_2\}$ **then**
13:         **if** e_1 and e_2 are saturated **then**
14:             make the decision according to the neighboring edges or the advice bits of v_{j_1} and v_{j_2} and update M accordingly
15:         **else if** only one of e_1 and e_2 is saturated **then**
16:             make a one-out-of-two-decision for vertex v_i, update M accordingly
17:         **else**
18:             make a one-out-of-three-decision for vertex v_i, update M accordingly
19:     **else if** $E_i = \{e_1, e_2, e_3\}$ **then**
20:         **if** at least 2 of the incident edges are saturated **then**
21:             the decision is already made, update M according to the decision
22:         **else if** only e_1 is saturated **then**
23:             make a one-out-of-two-decision for vertex v_i, update M accordingly
24:         **else**
25:             make a one-out-of-three-decision for vertex v_i, update M accordingly
26:     **output** $M_i = M$

OUTPUT: $M = (e_{i_1}, e_{i_2}, \ldots, e_{i_m}) = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$



(a) A log(3)-vertex saturates 3 incident edges.

(b) A 1-vertex saturates 2 incident edges.

Figure 3.36. The connection between saturated edges and their assigned vertices (marked darker).

construction of the algorithm, once an edge is saturated by a vertex it will not be saturated again by another vertex. So, the mapping of a an edge that is saturated by a vertex to a vertex is unique. Note that this mapping is not defined for every edge. Some of the edges are not saturated by any vertex but by the neighboring edges. But we are only interested in the edges which are saturated by vertices with advice larger than 0 since we want to count the number of advice bits used. The edges which are saturated by other edges and not by previous read advice bits do not account nothing to this sum.
A vertex that is labeled with log(3) always saturates exactly 3 edges. So, the algorithm asks $\frac{\log(3)}{3}$ advice bits in average for these saturated edges. Similarly, a vertex labeled with 1 saturates exactly two vertices implying an average advice bit request of $\frac{1}{2}$ per saturated edge (see Figure 3.36). Therefore, the maximum amount of advice that one edge needs in average is $\frac{\log(3)}{3}$. Because of Lemma 3.35, we know that there is a possibility to encode the one-out-of-three decisions corresponding to the label log(3) that is approximately log(3) in average but the algorithm has to read $\mathcal{O}(\log(n))$ bits in advance to get the number of one-out-of-three decisions. Therefore, Algorithm 3.6 reads at most

$$\frac{\log(3)}{3}(n-1) + \mathcal{O}(\log(n)) \leq \frac{\log(3)}{3}n + \mathcal{O}(\log(n))$$

advice bits for a $\{1,3\}$-tree on n vertices. □

This upper bound comes closer to the lower bound of $\frac{n}{2} - 3$ for online $\{1,3\}$-trees that are connected in every time step. An example of such an instance is shown in Figure 3.37. The vertex labels correspond to the number of advice bits Algorithm 3.6 uses while finding a maximum matching. More precisely, log(3) indicates that the algorithm makes a one-out-of-three decision, 1 stands for a one-out-of-two decision and 0 means that all decisions are already made.

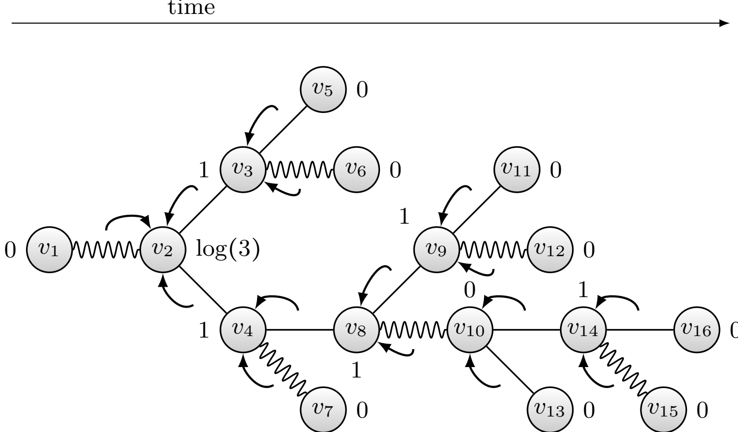


Figure 3.37. A $\{1,3\}$-tree that is connected in every time step of the online presentation. The labels indicate the number of advice bits, the algorithm reads in order to get an optimal maximum matching. The arrows illustrate the mapping of edges to the vertices by which they are saturated.

Lemma 3.39. *In a $\{1,3\}$-tree which is connected in every time step of the online presentation, Algorithm 3.6 has to make exactly one one-out-of-three decision.*

Proof. An online $\{1,3\}$-tree $T^{\prec}$ which is connected in every time step can have an edge set $|E_i| > 1$ only either in time step 2 ($T^{\prec}$ starts with an isolated edge), time step 3 ($T^{\prec}$ starts with two isolated edges) or time step 4 ($T^{\prec}$ starts with three isolated edges). After that, all new vertices lead to exactly one new edge since a new vertex can never be connected to two vertices of the same subtree. Hence, $|E_i| = 1$ holds for all $i > 4$. Therefore, Algorithm 3.6 only makes one one-out-of-three decision in the beginning of the online presentation. □

Therefore, the number of advice bits used in average per edge in $\{1,3\}$-trees that are connected in every time step is in most of the cases $\frac{1}{2}$ and only for three edges $\frac{\log(3)}{3}$ ($\frac{2}{3}$ respectively), i. e., for one vertex $\log(3)$ (2 respectively).

Corollary 3.40. *Algorithm 3.6 uses at most*

$$\frac{1}{2}n$$

advice bits to find a maximum matching in an online $\{1,3\}$-tree $T^{\prec} \in \mathcal{T}_n$ that is connected in every time step of the online presentation.

Proof. Because of Lemma 3.39, we know that the algorithm asks $\log(3)$ advice bits for only one vertex. For this one-out-of-three decision, the algorithm needs to read

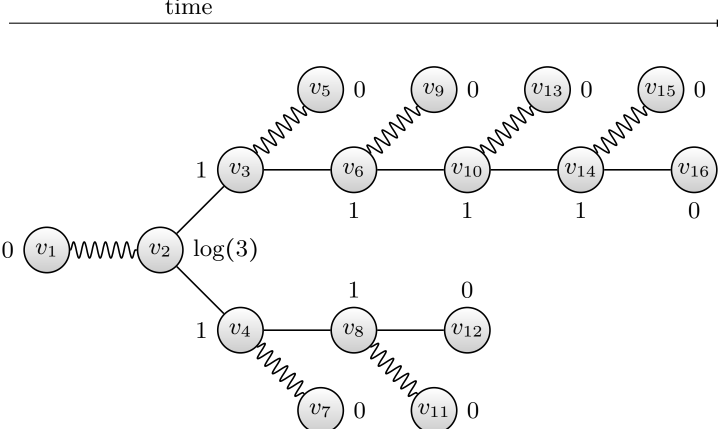


Figure 3.38. An online worst-case $\{1,3\}$-tree that is connected in every time step of the online presentation.

2 advice bits. The other vertices need at most 1 advice bit. In the worst case, the algorithm has to ask one advice bit for every inner vertex. In total, we have

$$(n-1)\cdot\frac{1}{2}+0.5=\frac{1}{2}n$$

advice bits since the algorithm asks for all of the edges in average $\frac{1}{2}$ advice bits up to three edges that need 2 advice bits in total what explains the 0.5 on the left side of the above equation. □

Note that the worst case can only happen if this online $\{1,3\}$-tree is a special comb similar to the one treated in Corollary 3.24 (compare Figure 3.38 with Figure 3.22). This has to be a worst-case instance since Algorithm 3.6 never reads advice bits for the leaves due to the construction of the algorithm.
Assume that Algorithm 3.6 reads k advice bits on such a worst-case instance. Then, this instance contains $n=4+2k$ vertices. Therefore, the algorithm reads in total

$$2+k=2+\frac{n-4}{2}=\frac{1}{2}n+2-2=\frac{1}{2}n$$

advice bits in the worst case.
Now it is time for lower bounds. In general, Algorithm 3.6 cannot guarantee asking less than approximately $\frac{\log(3)}{3}n$ advice bits on an online $\{1,3\}$-tree on n vertices. Hence, Theorem 3.38 gives us the best upper bound that Algorithm 3.6 can produce.

Theorem 3.41. *Algorithm 3.6 needs to read at least* $\lfloor\frac{\log(3)}{3}n-\frac{\log(3)}{3}\rfloor$ *advice bits in order to find a maximum matching in every* $\{1,3\}$*-tree of size* n*.*

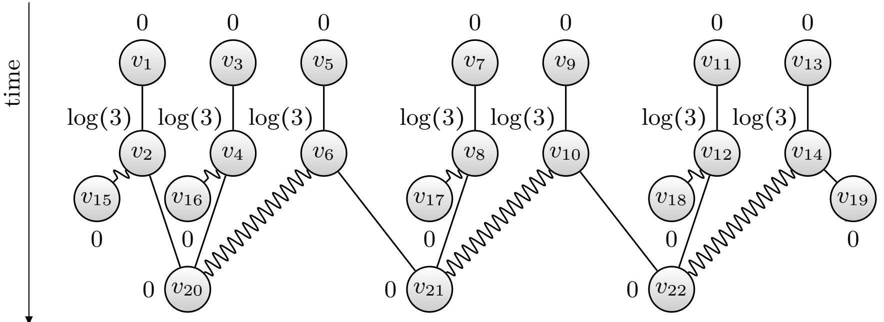


Figure 3.39. An online $\{1,3\}$-tree on $n = 22$ vertices for the lower bound of Algorithm 3.6 with a possible maximum matching.

Proof. Let $n = 3k + 1$ for some $k \in \{3, 5, 7, \ldots\}$. The instance class for the lower bound on the algorithm starts in the first phase with k isolated edges. Then, $\frac{k+1}{2}$ of these edges get an adjacent edge. In a last phase, repeatedly three of these subtrees are connected in order to get a $\{1,3\}$-tree as shown in Figure 3.39.
Since all the advice bits are asked on the first k isolated edges, it does not depend on the chosen maximum matching how much advice the algorithm needs to read. Therefore, Algorithm 3.6 needs

$$\log(3) \cdot k = \log(3) \cdot \frac{n-1}{3} = \frac{\log(3)}{3} n - \frac{\log(3)}{3}$$

advice bits on these instances in order to be optimal. □

3.5 Optimality in General and Bipartite Graphs

A natural generalization of paths and trees are bipartite graphs. In this section, we want to discuss a straight-forward upper bound that also holds for general graphs, and a lower bound on the number of advice bits any algorithm needs to read in order to find a maximum matching in bipartite graphs.

Upper Bound for General Graphs

A vertex in a general online graph on n vertices has at most $n-1$ neighbors, i. e., has vertex degree of at most $n-1$. For every vertex, at most one incident edge can be a matching edge. Therefore, if an algorithm asks, for every of the n vertices, which of the neighboring edges should be a matching edge, it finds a maximum matching. Algorithm 3.7 uses this strategy. Note that the algorithm needs to get to know the number n of vertices first, since otherwise it does not know how many neighbors one vertex can have at most.

Algorithm 3.7 Maximum Matching on General Graphs

INPUT: general graph $G^{\prec} \in \mathcal{G}_n$, for some $n \in \mathbb{N}$

1: $M = \varnothing$
2: read the number n of vertices from the advice tape
3: **for** $i = 1$ to $n-1$ **do**
4: read $\log(n)$ advice bits to decide which of the potentially $n-1$ incident edges is a matching edge and assign this edge, when it appears, to the matching and update M accordingly
5: assign an edge greedily to the matching for the last vertex v_n and update M accordingly

OUTPUT: $M = \{e_{i_1}, e_{i_2}, \dots, e_{i_m}\} = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

Theorem 3.42. *Algorithm 3.7 reads at most*

$$n\lceil\log(n)\rceil + 2\lceil\log(\lceil\log(n)\rceil)\rceil + 1$$

advice bits in order to find a maximum matching in a general online graph $G_n^{\prec}$ on n vertices.

Proof. Algorithm 3.7 finds a maximum matching in a general graph since the $\lceil\log(n)\rceil$ advice bits for a vertex v_i, for any $i \in \{1, 2, \dots, n\}$, indicate to the algorithm, which of the at most $n-1$ incident edges (if any) is a matching edge. For the last vertex v_n, the decision can be taken greedily: if there is a still unmatched neighbor v_j of v_n, the edge $\{v_j, v_n\}$ can be assigned to the matching.
But first, in line 2, Algorithm 3.7 needs to read the number n of vertices since otherwise the algorithm would not know how much $\lceil\log(n)\rceil$ advice bits are. This number n appears on the advice tape encode in a self-delimiting way as described in [45]. Therefore, the algorithm reads $\lceil\log(n)\rceil + 2\lceil\log(\lceil\log(n)\rceil)\rceil$ advice bits in a first step.
In total, Algorithm 3.7 needs to read at most

$$(n-1)\lceil\log(n)\rceil + \lceil\log(n)\rceil + 2\lceil\log(\lceil\log(n)\rceil)\rceil \le n\lceil\log(n)\rceil + 2\lceil\log(\lceil\log(n)\rceil)\rceil + 1$$

advice bits in order to be optimal. □

Approximately the same upper bound can also be reached with an algorithm that asks the oracle in advance, which of the potentially $\binom{n}{2} = \frac{n(n-1)}{2}$ edges should be assigned to the matching. Since a graph on n vertices can contain at most $\frac{n}{2}$ matching edges, the indices of the matching edges can be encoded with at most

$$\frac{n}{2} \cdot \left\lceil \log\left(\frac{n(n-1)}{2}\right)\right\rceil$$

advice bits. With a lower order term, we can encode the length n, and therefore this algorithm uses approximately the same number of advice bits as Algorithm 3.7.

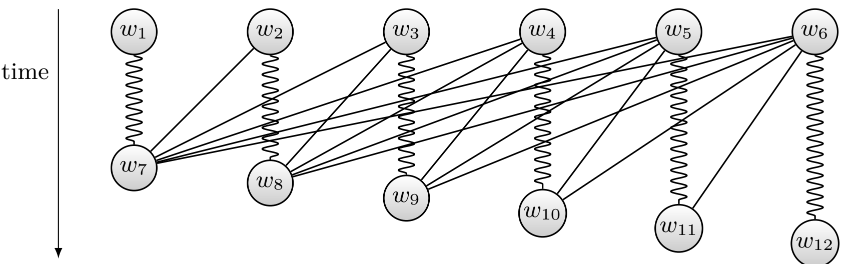


Figure 3.40. Pattern of the online bipartite graphs used in the proof of the lower bound on the number of advice bits used by every algorithm solving the online matching problem in bipartite graphs. Here, we have $n = 12$. The adversary shows first a permutation of the vertices $w_1, w_2, \ldots, w_6$. Then, the vertices $w_7, w_8, \ldots, w_{12}$ are given in this order.

Lower Bound for Bipartite Graphs

Now, we will show a matching lower bound which already holds for bipartite graphs. This lower bound has the same order $\Theta(n \log(n))$ as the upper bound for general graphs.

Theorem 3.43. *Any deterministic online algorithm for the problem of finding a maximum matching in an online bipartite graph instance needs to read at least*

$$\log\left(\left(\frac{n}{2}\right)!\right)$$

advice bits in order to find a maximum matching.

Proof. In order to show that no algorithm exists that needs less than $\log\left(\left(\frac{n}{2}\right)!\right)$ advice bits to be optimal, we describe a class of online instances in which each two of the instances need a different advice string. Since the cardinality of this class is $\left(\frac{n}{2}\right)!$, the algorithm cannot cope with less than $\log\left(\left(\frac{n}{2}\right)!\right)$ advice bits.
Let $n = 2k$ with $k \in \mathbb{N}$ be the number of vertices in the instance class, containing two shores of size k, $S_1 = \{w_1, w_2, \ldots, w_k\}$ and $S_2 = \{w_k, w_{k+1}, \ldots, w_{2k}\}$ (see Figure 3.40 for an example).
The edge set E is defined as

$$E = \{\{w_i, w_j\} \mid j \le i + k \text{ with } 1 \le i \le k \text{ and } k + 1 \le j \le 2k\},$$

leading to a unique perfect matching in these types of bipartite graphs since the only possibility for all vertices w_{2i} being matched is that they are matched by the edges $\{w_i, w_{2i}\}$, for $1 \le i \le k$.
The online presentation of one of these bipartite graphs starts with a permutation of isolated vertices $w_1, w_2, \ldots, w_k$. Then, the adversary presents the vertices $w_{k+1}, w_{k+2}, \ldots, w_{2k}$ in this order. In every time step i, for any $k + 1 \le i \le 2k$, the algorithm encounters the same prefix but it has to choose one of the $2k - i + 1$ edges

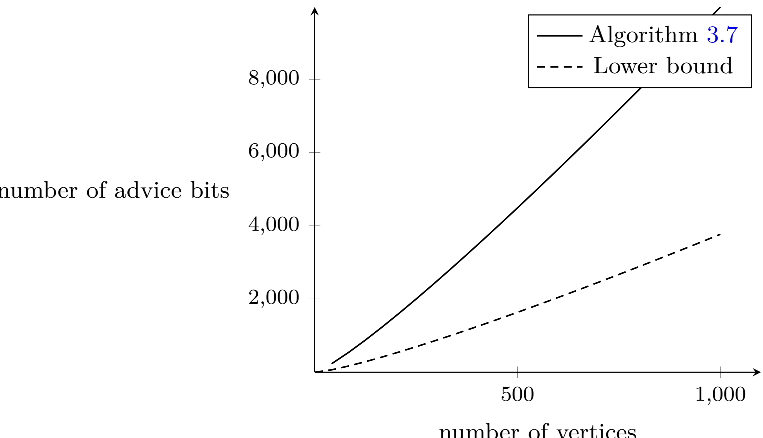


Figure 3.41. The lower bound and the upper bound on online algorithms with advice achieving an optimal solution in bipartite graphs.

incident to w_i to be a matching edge, and only one of these possibilities is correct. Therefore, the algorithm has to distinguish

$$\prod_{i=k+1}^{2k} (2k - i + 1) = \prod_{i=1}^{k} i = \left(\frac{n}{2}\right)!$$

different possible online representations of this bipartite graph, leading to

$$\log\left(\left(\frac{n}{2}\right)!\right)$$

advice bits that every online algorithm with advice needs to read at least in order to be optimal. With the help of Stirling's formula (see Equation (1.2) of Chapter 1), we see that

$$\log\left(\left(\frac{n}{2}\right)!\right) \geq \log\left(\sqrt{2\pi \cdot \frac{n}{2}} \cdot \left(\frac{\frac{n}{2}}{\mathrm{e}}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\left(\frac{n}{2\mathrm{e}}\right) + \frac{1}{2}\log(n) + \frac{\log(\pi)}{2} \in \Theta(n\log(n)),$$

concluding the proof. □

In Figure 3.41 the lower bound and the upper bound on optimality in bipartite graphs are compared.

P_k-free Bipartite Graphs

It is folklore that, in a complete bipartite graph, i. e., in a graph that contains all edges between the two shores (see Figure 3.42), there is an optimal deterministic online algorithm that solves the online matching problem on these types of graphs. We give the easy proof here for the sake of completeness.

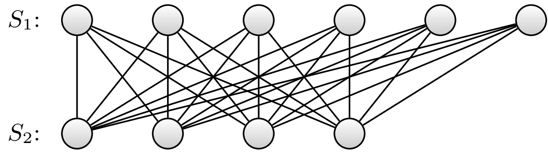


Figure 3.42. A complete bipartite graph on 10 vertices. One shore contains 6 vertices and the other one 4 vertices.

Lemma 3.44. *There exists a deterministic online algorithm for finding a maximum matching in an online complete bipartite graph.*

Proof. Let $B = (V, E)$ be a complete bipartite graph with the shores S_1 and S_2 such that $S_1 \cup S_2 = V$ and $|S_1| = m$ and $|S_2| = n$ and $m \geq n$. A maximum matching in a complete bipartite graph matches all vertices of the smaller shore. For each vertex on the smaller shore S_2, the algorithm can choose any of the incident edges to be a matching edge. This blocks exactly one vertex in S_1 from being part of a further matching edge. But, since $m \geq n$, there still remain sufficiently many vertices in S_1 that can be part of a matching edge. □

In this section, we want to discuss some results on bipartite graphs that do not contain certain types of paths as induced subgraphs.

Definition 3.45 (P_k-Free Bipartite Graph). A bipartite graph not containing a path P_k on k vertices, for some $k \in \mathbb{N}$, as an induced subgraph is called a *P_k-free bipartite graph.*

In the case of $k = 4$, a deterministic online algorithm solves the online matching problem optimally since, in [28], the authors showed that every P_4-free bipartite graph is a complete bipartite graph.

Lemma 3.46 (Fomin [28]). *Every connected bipartite P_4-free graph is a complete bipartite graph.* □

Corollary 3.47. *There is a deterministic online algorithm for finding a maximum matching in an online P_4-free bipartite graph.*

Proof. Due Lemma 3.46, every P_4-free bipartite graph is a complete bipartite graph, and therefore Lemma 3.44 states that there is an algorithm for solving the online matching problem on P_4-free bipartite graphs. □

Now, we want to show that the lower bound on the number of advice bits used by any online algorithm solving the online matching problem in bipartite graphs also holds for P_5-free bipartite graphs.

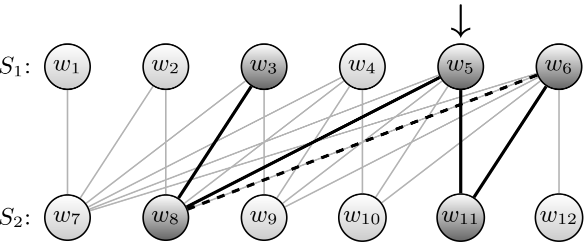


Figure 3.43. A path on 5 vertices, say $P_4 = (w_3, w_8, w_5, w_{11}, w_6)$, cannot be an induced subpath because there is an edge connecting the two vertices w_6 and w_8.

Theorem 3.48. *Any deterministic online algorithm for the online matching problem on online P_5-free bipartite graphs needs to read at least*

$$\log\left(\left(\frac{n}{2}\right)!\right)$$

advice bits in order to be optimal.

Proof. We show that the instances of the instance class described in the proof of Theorem 3.43 are P_5-free and therefore the lower bound for general bipartite graphs carries over to P_5-free bipartite graphs.
Assume that one of these instances on $n = 2k$ vertices contains an induced P_5. Let w_i be the middle vertex of this path and, w.l.o.g., $w_i \in S_1$ and therefore, $i \in \{1, 2, \ldots, k\}$. Because of the construction, both neighbors w_{j_1} and w_{j_2} of w_i in the path, lie left or below of w_i, i. e., $j_1 < j_2 \leq i + k$ (see Figure 3.43 for an example). The next neighboring vertex of w_{j_2} in the path is an arbitrary vertex w_{j_3} lying on the opposite shore above or right of w_{j_2}, i. e., $j_2 \leq j_3$. Due to the construction, there is also an edge $\{w_{j_1}, w_{j_3}\}$ disproving the assumption that the graph contains an induced P_5. Hence, the graph is P_5-free, concluding the proof. □

Since, for any $k \geq 6$, the class of all P_k-free graphs contains all P_5-free graphs, we know that this lower bound also holds for all P_k-free bipartite graphs with $k \geq 5$.

Theorem 3.49. *Any deterministic online algorithm for the online matching problem on online P_k-free bipartite graphs, for any $k \geq 5$, needs to read at least*

$$\log\left(\left(\frac{n}{2}\right)!\right)$$

advice bits in order to be optimal.

Therefore, excluding a P_k from a bipartite graph, for any $k \geq 5$, leads to the same lower bound for the online matching problem as in general bipartite graphs.

Graphs With Small Diameter

The graphs from the class described in the proof of Theorem 3.43 have the special property that every vertex can be reached from every other vertex via a short path. This property can be measured by the diameter of a graph.

Definition 3.50 (Diameter). The *diameter* of a graph is the maximum distance of two arbitrary vertices. The distance of two vertices u and v is measured by the length of the shortest u-v-path, i. e., the number of edges on this path.

We show that the instances used to show the lower bound on general bipartite graphs have diameter 3, implying that bipartite graphs with small diameter are not easier to solve than general bipartite graphs.

Theorem 3.51. *Any deterministic online algorithm for the online matching problem on online bipartite graphs with diameter 3 needs to read at least*

$$\log\left(\left(\frac{n}{2}\right)!\right)$$

advice bits in order to be optimal.

Proof. Showing that the instances of the graph class described in the proof of Theorem 3.43 have diameter 3, we can prove that the general lower bound for bipartite graphs also holds for online bipartite graphs of diameter 3.
There are three cases for two vertices w_i and w_j:

(a) Either they are both on the same shore, say, w.l.o.g., in S_1, and $i < j$, or

(b) $w_i \in S_1$ and $w_j \in S_2$ with $i \geq j - k$, or

(c) $w_i \in S_1$ and $w_j \in S_2$ with $i < j - k$.

In the first case, a path (w_i, w_{k+i}, w_j) is a path of length 2 from w_i to w_j. In the second case, the edge $\{w_i, w_j\}$ is present and therefore they have distance 1. And in the last case, the path $(w_i, w_{i+k}, w_{j-k}, w_k)$ has length 3 (see Figure 3.44 for an example). Therefore, every pair of vertices is connected with a path of length smaller than four, implying a diameter of 3 in these graphs. □

3.6 Tradeoffs in Paths

Until now, we know bounds on how much advice is needed to be optimal. We want to analyze what happens in the case when the algorithm has less than this number of advice bits on disposal.
In this section, we return to the problem of online maximum matching on paths. We want to discuss some tradeoffs between the amount of advice bits algorithms

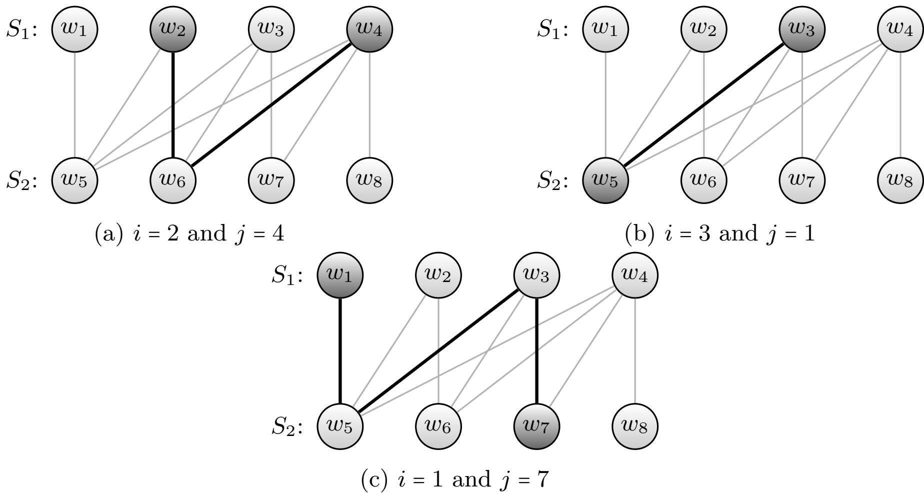


Figure 3.44. The three cases in the proof of Theorem 3.51 show that these graphs all have diameter 3.

are allowed to read and the competitive ratio these algorithms with advice can reach. We already know that any algorithm needs to read at least

$$\left\lfloor \frac{1}{3}n - \frac{1}{2}\log(n) + \log\left(\sqrt{\frac{3}{2\pi}}\right)\right\rfloor$$

advice bits for solving the online matching problem optimally (see Theorem 3.20). Algorithm 3.4 almost matches this lower bound using $\lceil \frac{n}{3} \rceil$ advice bits (see Theorem 3.16).
We showed in Section 3.1 that all online algorithms solving the online matching problems on paths without advice have a competitive ratio of at least $\frac{3}{2} - \varepsilon$ (see Theorem 3.14). A greedy algorithm provides us with a matching upper bound of $\frac{3}{2}$ on the competitive ratio proven in Theorem 3.11.

A Lower Bound for One Advice Bit

First, we want to show that, for one bit of advice, we can prove the same lower bound as for deterministic algorithms on paths.

Theorem 3.52. *No online algorithm using one bit of advice for finding a maximum matching in a path on n vertices with $n \geq \frac{3(\alpha+1)}{\varepsilon} - 2$ can be better than $\left(\frac{3}{2} - \varepsilon\right)$-competitive (with additive constant α), for arbitrary non-negative constants α and ε.*

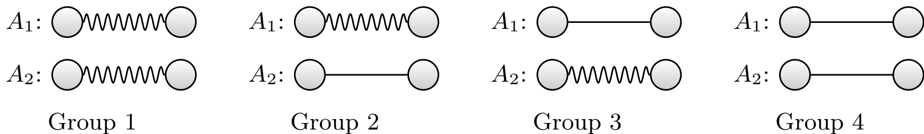


Figure 3.45. The four possibilities how a pair of algorithms A_1 and A_2 can decide about an isolated edge.

Proof. One bit of advice can be used to decide which of two given deterministic algorithms should be used in order to find a maximum matching in paths on n vertices. This makes the work of the adversary more involved since he has to provide a class of lower bound instances for all pairs of algorithms. To cover all these pairs, we distinguish the algorithms with respect to the behaviour on the isolated edges for a given online path instance P_n. Let A_1 and A_2 be two arbitrary online algorithms. There are four possibilities how these two algorithms can treat an isolated edge e (see Figure 3.45):

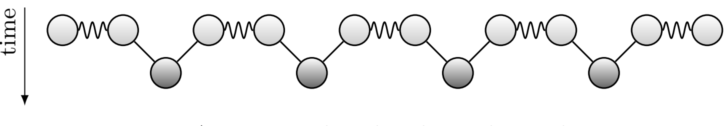
Group 1: Both, A_1 and A_2 assign e to the matching.

Group 2: Only A_1 takes e into the matching.

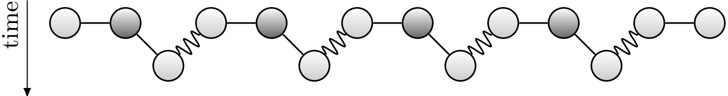
Group 3: Only A_2 takes e into the matching.

Group 4: Neither A_1 nor A_2 allocate e to the matching.

This gives us a classification of the isolated edges. Note that the adversary fights against two arbitrary algorithms. Therefore, some of these groups of edges can be empty. The algorithm has to handle all subsets of these four groups.
Recall that, due to Lemma 3.13, a matching M with k unmatched vertices has $\lfloor \frac{k}{2} \rfloor$ matching-edges less than a maximum matching on this instance. Therefore, it suffices if we count the minimum number of unmatched vertices on the instances of these special instance class in order to calculate a lower bound on the competitive ratio for algorithms reading one bit of advice.
The construction of the class of lower bound instances is similar to the one in the proof of Theorem 3.14. Let $n = 6k + 4$ be the number of vertices in the online path P_n for a $k \in \mathbb{N}$. The adversary starts with $\frac{n-1}{3}$ isolated edges which belong to one of the above described edge groups. In order to reach a high number of unmatched vertices, the adversary arranges the isolated edges group by group in order to build four subpaths $P^{(i)}$, for $i \in \{1, 2, 3, 4\}$, connecting the isolated edges inside a group by single vertices. Note that, in each group, the neighboring isolated edges are either both matching or both non-matching edges. Therefore, if the algorithms act best possible, any of the subpaths $P^{(i)}$ built by any of the two algorithms is of one of the two forms as shown in Figure 3.46.
In both types of subpaths containing i isolated edges, the adversary forces the algorithm to leave at least $i - 1$ vertices unmatched. Therefore, we can guarantee

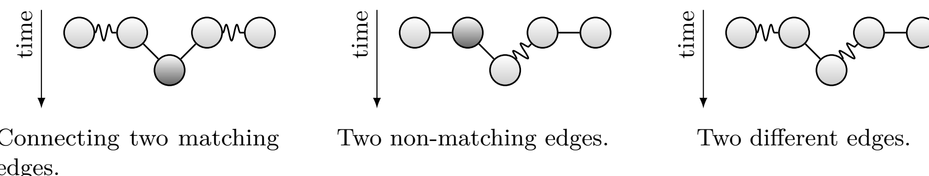


Type 1: A group with isolated matching edges.



Type 2: A group with isolated non-matching edges

Figure 3.46. The algorithm chooses, inside each group, one of the two patterns if it acts best possible connecting the isolated edges to a path. The marked vertices are the unmatched vertices within each group (ignoring the end vertices for type 2 subpaths). Note that we can assume w.l.o.g. that the algorithm always assigns the right edge to the matching when connecting the isolated edges in a subpath of type 2.



Connecting two matching edges. Two non-matching edges. Two different edges.

Figure 3.47. Connecting two isolated edges. The adversary can only force the algorithm to leave a vertex unmatched when connecting two edges of the same type.

$i-1$ unmatched vertices in each group. In type 2 subpaths, we have possibly further unmatched vertices at the left and the right end. But we will count these possibly unmatched vertices in a second step, when the adversary connects the subpaths such that the two algorithms are forced to leave many vertices unmatched in between the at most four groups. Note that connecting two edges that are both matching or both non-matching edges, forces the algorithm to leave one vertex unmatched. On the other hand, connecting two different edges enables the algorithm to construct at least locally a good matching (see Figure 3.47).
The end edges of possibly four subpaths are given in Figure 3.45. Therefore, these pairs of edges will symbolize the whole subpaths in the following figures. We will do a case distinction with respect to the subset of the groups that arise in the solution of a given pair of algorithms:

Groups 1, 2, 3, 4 If all the edge groups are present in a solution, we arrange the subpaths in the order 2, 1, 3, 4, as shown in Figure 3.48.

Both algorithms A_1 and A_2, have to leave 3 vertices unmatched. Since in a subpath on i vertices, both algorithms leave at least $i-1$ vertices unmatched,

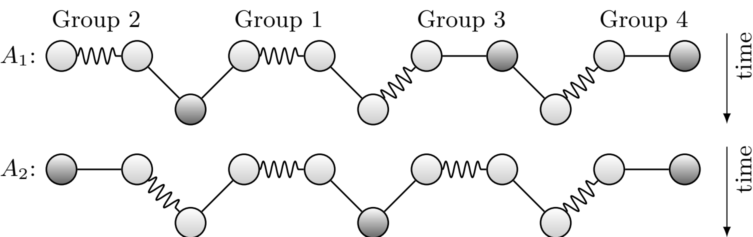


Figure 3.48. All of the edge groups are present in the solution.

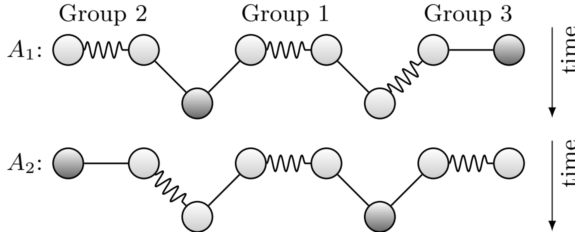


Figure 3.49. The edge groups 1, 2 and 3 are present in the solution.

we have $\frac{n-1}{3} - 4$ unmatched vertices within the four subpaths. Therefore, overall, the adversary can force both algorithms to leave at least

$$\frac{n-1}{3} - 4 + 3 = \frac{n-1}{3} - 1 = 2k$$

vertices unmatched, leading to a loss of k matching edges as in the proof of Theorem 3.14.

If we can show the same for all other subgroups, we are done, since we get the same lower bound as in the deterministic case.

Groups 1, 2, 3 When no edges from group 4 appear in the solution, the adversary arranges the remaining groups as shown in Figure 3.49.

Within the three subpaths, there are at least $\frac{n-1}{3} - 3$ unmatched vertices and together with the two additional unmatched vertices between the paths, we have again $\frac{n-1}{3} - 1$ unmatched vertices.

Groups 2, 3, 4 In this case, the adversary provides the order of Figure 3.50, leading to at least $\frac{n-1}{3} - 1$ unmatched vertices.

Groups 1, 2, 4 or groups 1, 3, 4 In these two equivalent cases, the order of Figure 3.51 leads to $\frac{n-1}{3} - 1$ unmatched vertices.

Groups 1, 4 or groups 2, 4 or groups 3, 4 If only two edge groups appear in the solution, there are $\frac{n-1}{3} - 2$ unmatched vertices within the two subpaths.

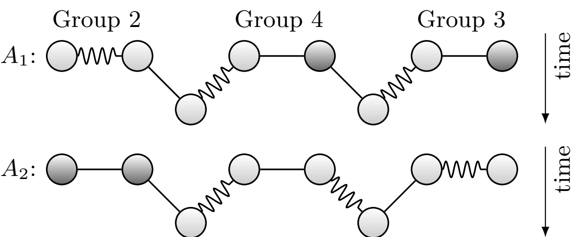


Figure 3.50. The edge groups 2, 3 and 4 are present in the solution.

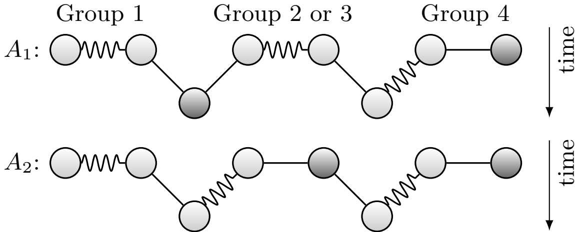


Figure 3.51. The edge groups 1 and 4 and either group 2 or group 3 are present in the solution.

Therefore, one unmatched vertex between the two subpaths or at one end would be enough to reach the bound of $\frac{n-1}{3} - 1$ unmatched vertices in total. We see in Figure 3.52 that this is given for all pairs of groups containing group 4, since the adversary can force the algorithm to leave at least the right end vertex unmatched.

Groups 1, 2 or groups 1, 3 Also in these two cases, the algorithm can force the algorithm to leave $\frac{n-1}{3} - 1$ vertices unmatched (see Figure 3.53).

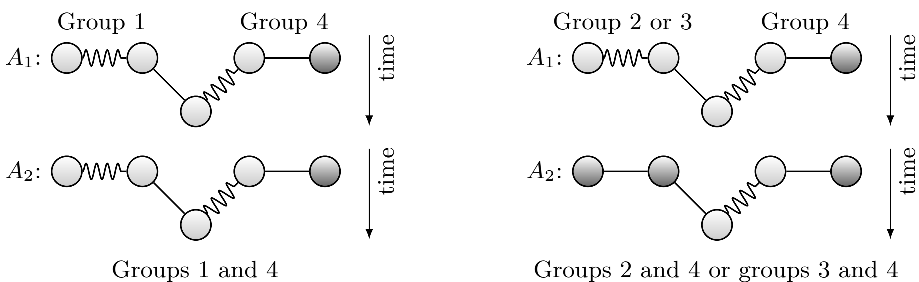


Figure 3.52. All pairs of edge groups containing group 4 in the solution.

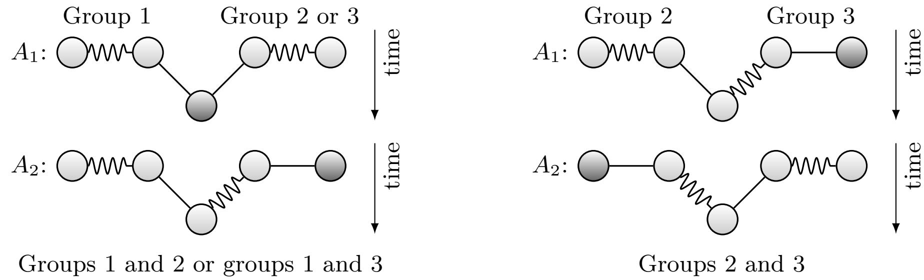


Figure 3.53. All remaining pairs of two edge groups.

Groups 2 and 3 Also in this case, one of the end vertices will stay unmatched (see Figure 3.53) and therefore the adversary can guarantee a bound of $\frac{n-1}{3}$ unmatched vertices.

Group 1, 2, 3 or 4 If only one group of edges is present in the solution, an algorithm leaves at least $\frac{n-1}{3}$ vertices unmatched within this path.

Summarizing, in all possible subsets of isolated edges, the solution has to contain at least $\frac{n-1}{3}$ unmatched vertices. In the proof of Theorem 3.14, we showed that this leads to a competitive ratio of $\frac{3}{2} - \varepsilon$ in paths on n vertices with $n \geq \frac{3(\alpha+1)}{\varepsilon} - 2$, for arbitrary non-negative constants α and ε. □

Note that Theorem 3.52 shows the same lower bound for the competitive ratio as the one for deterministic online algorithms. Therefore, Theorem 3.11 with its greedy Algorithm 3.3 implies an almost matching upper bound of $\frac{3}{2}$ on the competitive ratio for online algorithms reading one advice bit in order to solve the online matching problem on paths.
It remains open to analyze how much advice bits can help an algorithm to improve over any deterministic online algorithm.

Not Enough Advice Bits to Be Optimal

Recall that we showed in Lemma 3.8 that, for every path P_n, there is exactly one maximum matching that matches all inner vertices. Using this result, we could show in the proof of Theorem 3.16 that, if an algorithm gets advice for every isolated edge and for isolated paths on 3 vertices (that arise from connecting the new vertex v_i with two isolated vertices in time step i), then the non-isolated edges can be treated greedily in order to get a maximum matching. The advice bit was used in the first case to decide if the isolated edge is a matching edge or not. In the second case, the advice bit indicates which of the two edges is a matching edge. Because at most $\lceil \frac{n}{3} \rceil$ edges can appear isolated, the algorithm reads at most $\lceil \frac{n}{3} \rceil$ advice bits. In the following, we will discuss algorithms that have less than

$\lceil \frac{n}{3} \rceil$ advice bits on disposal. Obviously, such algorithms cannot always find an optimal solution.
We want to use the same idea as described before for an upper bound on the tradeoff of advice bits and the competitive ratio of the algorithm. If the algorithm gets less advice bits, it may happen that the decision on some of the isolated edges or paths on 3 vertices is wrong. The question is, by how much the solution is worse than in a maximum matching. For the sake of convenience, we will fix for every instance one maximum matching M^*, which matches all inner vertices. We will compare the solution of the algorithm always with M^* although also other maximum matchings may exist.
We need to know what happens with the non-isolated edges between two given neighboring isolated edges when treating them greedily. We observe that, given a decision on the isolated edges, a greedy strategy applied on the non-isolated edges yields the best possible matching on every subpath between two nearest previous isolated edges. Note that the following results are, for the sake of convenience, only formulated for isolated edges but they are also valid for isolated paths on 3 vertices.

Lemma 3.53. *Assume that the decision on two neighboring isolated edges e_1 and e_2 is made. Then the greedy algorithm, which assigns every possible non-isolated edge to the matching, leaves at most one inner vertex on the subpath containing e_1 and e_2 as end edges unmatched.*

Proof. There are 3 possibilities how the isolated edges e_1 and e_2 can be set:

1. e_1 and e_2 are both matching edges,
2. e_1 and e_2 are both non-matching edges,
3. e_1 is a matching edge and e_2 not (the converse case is equivalent).

For each of these cases, the number of edges between e_1 and e_2 can be either even or odd.
Since e_1 and e_2 are neighboring isolated edges, all the edges in between are either appended to the subpath containing e_1 or to the subpath containing e_2. This always happens without leaving any vertex unmatched, since the algorithm acts greedily. At some point, both subpaths will be connected with a vertex v_i. In this time step i, at most one of the vertices v_i or its neighbors can remain unmatched because a greedy algorithm admits at most one unmatched vertex when connecting two subpaths.
Table 3.1 gives an overview of the unmatched vertices in all the cases with an odd or an even number of vertices between two isolated edges e_1 and e_2. □

A consequence of Lemma 3.53 is that, if one of two neighboring isolated edges is treated wrongly by the algorithm, there remains one vertex unmatched on the subpath connecting them.

Table 3.1. Number of unmatched vertices between two given neighboring isolated edges e_1 and e_2 treating the non-isolated edges greedily. The column with the parity indicates if there are an even or an odd number of edges between e_1 and e_2.

Case	Parity Edges	Unmatched Vertices	Example
1	even	1	
1	odd	0	
2	even	1	
2	odd	0	
3	even	0	
3	odd	1	

Corollary 3.54. *Assume that the algorithm does not decide right on one of two neighboring isolated edges e_1 or e_2 with respect to a fixed maximum matching M^* which matches all inner vertices. Then, there remains one inner vertex unmatched on the subpath connecting those two edges.*

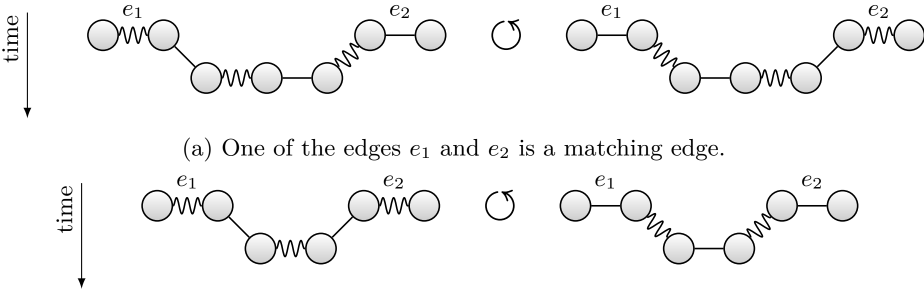
Proof. Let e_1 and e_2 be two neighboring isolated edges. If there are an odd number of edges between e_1 and e_2 in the complete online graph, an algorithm which matches all inner vertices, either has to assign both edges e_1 and e_2 to the matching or leave both unmatched in order to solve the problem optimally. Therefore, a wrong decision would be to treat e_1 and e_2 differently leading to an unmatched vertex on the subpath containing e_1 and e_2 as end vertices due to Table 3.1.
If e_1 and e_2 have an even number of edges in between, an algorithm would have to choose one of the two edges to be a matching edge and leave the other edge unmatched in order to be optimal on this subpath. Due to Table 3.1, we see that a wrong decision on one of the edges e_1 and e_2 leads again to an unmatched vertex between e_1 and e_2, concluding the proof. □

The good news is that, if both neighboring isolated edges e_1 and e_2 are treated wrongly, the mistake on the path containing e_1 and e_2 as end vertices is canceled

out and the greedy algorithm connects this two edges without leaving any inner vertex on the connecting subpath unmatched.

Corollary 3.55. *If the algorithm decides for both neighboring isolated edges e_1 and e_2 wrongly with respect to a fixed maximum matching M^* that matches all inner vertices, the greedy algorithm connects these two edges without leaving any inner vertex on the connecting subpath unmatched.*

Proof. For a fixed maximum matching, flipping the matched and the unmatched edges on a certain subpath containing e_1 and e_2 as end edges leads to a matching that matches all inner vertices on this subpath (see Figure 3.54). □



(a) One of the edges e_1 and e_2 is a matching edge.

(b) Either both edges e_1 and e_2 are matching edges or both not.

Figure 3.54. Flipping all matching and non-matching edges on a path containing e_1 and e_2 as end vertices yields again a matching that matches all inner vertices.

The local matching of Corollary 3.55 on a subpath containing e_1 and e_2 as end edges does not have to be part of the fixed maximum matching M^*, but there will be no unmatched vertex on this subpath.
Corollary 3.54 and Corollary 3.55 show that an algorithm that acts greedily on non-isolated edges causes an unmatched vertex on a subpath of non-isolated edges only in the case that exactly one isolated end edge was set wrong with respect to a fixed matching M^* that matches all inner vertices (see Figure 3.55 for an example).

Therefore, a wrong decision on an isolated edge can cause at most two unmatched vertices when the algorithm uses a greedy strategy on the non-isolated edges.
The worst case for i isolated edges, for some $i \in \{0, 1, 2, \ldots, \lceil \frac{n}{3} \rceil\}$, happens if an algorithm decides wrongly on $\lceil \frac{i}{2} \rceil$ of the i isolated edges, $i \in \{0, 1, 2, \ldots, \lceil \frac{n}{3} \rceil\}$, and the adversary places always an isolated edge with a right decision next to one with a wrong decision, beginning with a wrong decision as the leftmost isolated edge. We get the following pattern on the isolated edges:

$$✗ ✓ ✗ ✓ ✗ \ldots ✓ ✗ ✓ ✗ ✓ ✗$$

time

(a) A fixed maximum matching M^*.

time

(b) The maximum matching M found by an algorithm.

Figure 3.55. An example of a fixed maximum matching M^* and a matching M for which some of the isolated edges are set different by the algorithm.

The leftmost and the rightmost subpath containing each only one isolated edge as an end edge might not contain any unmatched vertex. But every other subpath containing neighboring isolated edges as end edges, contains one unmatched vertex in this example. Therefore, the algorithm leaves at most $i+1$ vertices unmatched if it acts greedily on the non-isolated edges.

Corollary 3.56. *An algorithm that acts greedily on non-isolated edges leaves at most $i+1$ vertices unmatched in an online path instance containing i isolated edges, for some $i \in \{0,1,2,\ldots,\lceil \frac{n}{3} \rceil\}$. This bound is reached for $\lceil \frac{i}{2} \rceil$ wrongly set isolated edges.* □

So, the worst case occurs for $i = \lceil \frac{n}{3} \rceil$ leading, due to Lemma 3.13, to

$$\left\lfloor \frac{i+1}{2} \right\rfloor = \left\lfloor \frac{\lceil \frac{n}{3} \rceil + 1}{2} \right\rfloor$$

matching edges less with respect to a fixed maximum matching M^* that matches all inner vertices. Recall that a maximum matching in a path of n vertices contains $\lfloor \frac{n}{2} \rfloor$ matching edges. Hence, an algorithm as described above, finds a matching containing at least

$$\left\lfloor \frac{n}{2} \right\rfloor - \left\lfloor \frac{\lceil \frac{n}{3} \rceil + 1}{2} \right\rfloor \geq \frac{n}{2} - \frac{1}{2} - \frac{\lceil \frac{n}{3} \rceil + 1}{2} \geq \frac{n}{2} - \frac{1}{2} - \frac{\frac{n}{3} + \frac{2}{3} + 1}{2} \geq \frac{n}{3} - \frac{4}{3}$$

matching edges in an online path containing n vertices and i isolated edges. This is a similar result as we have already seen in Lemma 3.10, with the difference that this result speaks about *all* algorithms that treat the non-isolated edges greedily. In Lemma 3.10, we only proved that an algorithm that acts greedily on all edges reaches a slightly higher bound.

Corollary 3.57. *Every deterministic algorithm that acts greedily on non-isolated edges, in order to find a matching in an online path om n vertices, chooses a matching of size at least $\lceil \frac{n}{3} - \frac{4}{3} \rceil$.* □

Now, we are ready to look at algorithms with advice that use less than $\lfloor \frac{n}{3} \rfloor$ advice bits, which would be necessary in order to find a maximum matching in every online path on n vertices. Recall again Algorithm 3.4 which finds an optimal solution. This algorithm reads an advice bit for every isolated edge and for every isolated path on 3 vertices. In all other cases, it acts greedily. In order to have an algorithm using only $\lceil \frac{n}{3} \rceil - k$ advice bits, we modify this algorithm to Algorithm 3.8 such that this new algorithm acts the same as the previous one until all allowed advice bits are exhausted. Then, the algorithm follows a greedy strategy on all (possibly also isolated) edges. To implement this strategy, the algorithm has to know the length n of the input. This can be communicated in a self-delimiting way by using additional $\lceil \log(n) \rceil + 2\lceil \log(\lceil \log(n) \rceil) \rceil$ advice bits (see Lemma 1.25).
Note that there are at most k isolated edges or isolated paths on 3 vertices left in the remainder which has to be set without advice. In the worst case, all of these remaining isolated subpaths are set wrong.

Theorem 3.58. *Algorithm 3.8 finds a matching on* $\max\left(\left\lceil \frac{n}{3} - \frac{4}{3} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor - k\right)$ *edges in a path on n vertices using at most* $\lceil \frac{n}{3} \rceil + \lceil \log(n) \rceil + 2\lceil \log(\lceil \log(n) \rceil) \rceil - k$ *advice bits for some* $k \in \left\{0, 1, \ldots, \lceil \frac{n}{3} \rceil\right\}$.

Proof. Due to Corollary 3.57, no matter how the isolated edges are set, every algorithm that matches non-isolated edges greedily, i. e., including Algorithm 3.8, matches at least $\lceil \frac{n}{3} - \frac{4}{3} \rceil$ edges.
But for small k, i. e., for

$$k \leq \left\lceil \frac{\lceil \frac{n}{3} \rceil}{2} \right\rceil,$$

the algorithm can reach a better bound, as we will show now. The algorithm first reads the input length n from the advice tape. From this, it knows how many advice bits it can use for the isolated edges. Since Algorithm 3.8 first uses all advice bits, at most k isolated edges are left for which the algorithm has no advice left. Therefore, it greedily assigns all these isolated edges to the matching. In the worst case, all these edges which were set without advice are wrong. Since, k is smaller than approximately half of the possible isolated edges, Corollary 3.56 states that, in the worst case, the adversary can arrange the edges on which the wrong decision was made such that they have two neighboring isolated edges with a right decision. Hence, every mistake leads to two unmatched vertices and therefore to a matching edge less with respect to a fixed maximum matching M^* that matches all inner vertices.
Figure 3.56 visualizes the tradeoff between the number of advice bits read and the number of matching edges that Algorithm 3.8 can reach on every online path on n vertices. □

In Theorem 3.58, we fixed a maximum matching M^* for every online path and compared the work of Algorithm 3.8 with respect to M^*. Paths with an odd number of vertices contain $\frac{n+1}{2}$ different maximum matchings (see Lemma 3.7)

Algorithm 3.8 Matching on Paths with $\lceil \frac{n}{3} \rceil - k$ advice bits

INPUT: $P^{\prec} \in \mathcal{P}_n$, for some $n \in \mathbb{N}$

```
 1: M = ∅
 2: read the number n of vertices from the advice tape
 3: adv = ⌈n/3⌉ − k
 4: for i = 1 to n do
 5:     while adv > 0 do
 6:         if (a) v_i has exactly one edge e to a previously isolated vertex then
 7:             read an advice bit σ to decide whether e is a matching edge:
 8:             if σ = 1 then
 9:                 M ← M ∪ {e}
10:             else
11:                 M ← M
12:             adv ← adv − 1
13:         else if (b) v_i has two edges e_1 and e_2 to prev. isolated vertices then
14:             use an advice bit σ to decide whether e_1 or e_2 is a matching edge:
15:             if σ = 0 then
16:                 M ← M ∪ {e_1}
17:             else
18:                 M ← M ∪ {e_2}
19:             adv ← adv − 1
20:         else if (c) v_i is connected to some non-isolated and unmatched vertex
            by an edge e then
21:             M ← M ∪ {e}
22:         else if (d) v_i has an edge e_1 to a matched vertex and an edge e_2 to an
            isolated vertex then
23:             M ← M ∪ {e_2}
24:         else
25:             M ← M
26:         output M_i = M
27:     # Advice bits are all used up:
28:     if (a) then
29:         M ← M
30:     else if (b) then
31:         M ← M ∪ {e_1}
32:     else if (c) then
33:         M ← M ∪ {e}
34:     else if (d) then
35:         M ← M ∪ {e_2}
36:     else
37:         M ← M
38:     output M_i = M
```

OUTPUT: $M = (e_{i_1}, e_{i_2}, \dots, e_{i_m}) = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

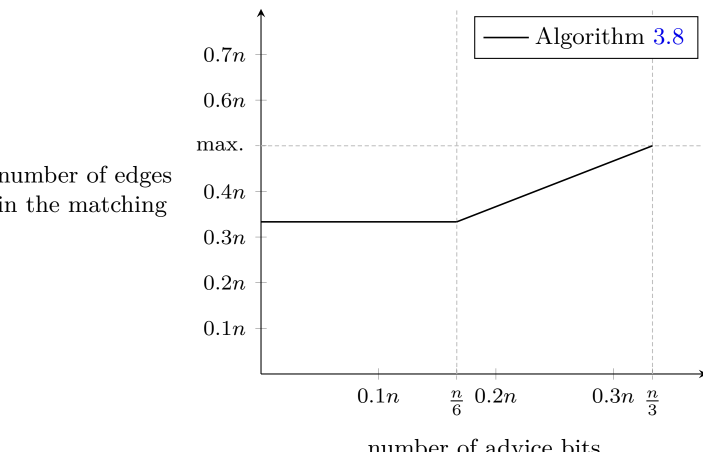


Figure 3.56. Minimum number of edges found by Algorithm 3.8 in an arbitrary online path on n edges with respect to the number of advice bits that are on disposal.

admitting that the algorithm can afford one unmatched inner vertex. But allowing all maximum matchings does not improve the algorithm since every wrongly set isolated edge forces two unmatched vertices.

3.7 Tradeoffs in Paths and Trees via the String Guessing Problem

We already showed that, given the decision on the isolated edges, an algorithm solves the online matching problem in paths optimally. Therefore, we used in Section 3.6 the straight-forward idea that the algorithm asks for every isolated edge one advice bit in order to set this edge correctly. Algorithm 3.8 asks k advice bits less than would be necessary to find a maximum matching. It uses up all advice bits for the first isolated edges and decides on the remaining isolated edges greedily as well as on all non-isolated edges. Algorithm 3.8 finds a matching containing $\max\left(\left\lceil \frac{n}{3} - \frac{4}{3} \right\rceil, \left\lfloor \frac{n}{2} \right\rfloor - k\right)$ edges.
In this section, we want to modify this algorithm in order to give an improved upper bound. We will use an algorithm for the string guessing problem to find a correct setting for the isolated edges. This algorithm will reach a better competitive ratio for a fixed number of advice bits.
In the string guessing problem, the algorithm has to guess a bit string

$$b = b_1 b_2 b_3 \dots b_n$$

of a given length n, for some $b_i \in \{0, 1\}$ for all $i \in \{1, 2, \dots, n\}$. An online algorithm solves the online instance bit by bit, starting with b_1. All requests contain the

question what the next bit is, denoted by $?_i$ for the ith request. In the first request, the algorithm also receives the length n of the bit string. And as a last, concluding request, the bit string b itself is given in order to distinguish the instances of the same length n. The goal of the online algorithm is to minimize the number of wrongly guessed bits. This measure can be described by the Hamming distance, denoted by Ham, of the string b and the guessed bit string. The following two definitions are a special case of the definitions in [8, 9].

Definition 3.59 (String Guessing with Unknown History).

Input: A sequence of requests $B = ((n, ?_1), ?_2, ?_3, \dots, ?_n, b)$ for a bit string $b = b_1 b_2 b_3 \dots b_n$ and some $n \in \mathbb{N}$

Output: Guessed bits: $(\beta_1, \beta_2, \dots, \beta_n, \lambda)$ such that $\beta_i \in \{0, 1\}$ for all $i \in \{1, 2, \dots, n\}$

Cost: Number of wrongly guessed bits: $\text{Ham}(b_1 b_2 \dots b_n, \beta_1 \beta_2 \dots \beta_n)$

Goal: Minimum

In this online optimization problem, it might be useful if the algorithm would know the correct answers to the already set part of the online instance. In the setting of the so-called string guessing with known history, the algorithm receives, with every new request, the correct answer for the preceding request.

Definition 3.60 (String Guessing with Known History).

Input: A sequence of requests

$$B = ((n, ?_1), (b_1, ?_2), (b_1 b_2, ?_3), \dots, (b_1 b_2 \dots b_{n-1}, ?_n))$$

for a bit string $b = b_1 b_2 b_3 \dots b_n$ and some $n \in \mathbb{N}$

Output: Guessed bits: $(\beta_1, \beta_2, \dots, \beta_n, \lambda)$ such that $\beta_i \in \{0, 1\}$ for all $i \in \{1, 2, \dots, n\}$

Cost: Number of wrongly guessed bits: $\text{Ham}(b_1 b_2 \dots b_n, \beta_1 \beta_2 \dots \beta_n)$

Goal: Minimum

An Upper Bound for the Online Matching Problem on Paths

The new algorithm will use an algorithm for the string guessing problem with known history as a subroutine to compute advice for the isolated edges. Therefore, we need to reduce the online matching problem to the string guessing problem with known history.

Figure 3.57. This online path instance on 16 vertices contains the isolated edges e_1, e_2, e_3, and e_4. The maximum matching corresponds to the bit string $b = b_1b_2b_3b_4b_5b_6 = 100100$ in the bit string guessing problem with known history. b_1 to b_4 correspond to the decision on the edges e_1 to e_4, and b_5 and b_6 are non-relevant bits that occur since this online path does not contain the maximum possible number of isolated edges, which would be 6.

Lemma 3.61. *There is an algorithm with advice that solves the online matching problem with help of the string guessing problem with known history.*

Proof. Let $P^{\prec} \in \mathcal{P}_n$ be an online path instance for the online matching problem and let M^* be a fixed maximum matching that matches all inner vertices. Algorithm 3.9 works on the non-isolated edges in the same way as Algorithm 3.8, namely greedily. But the treatment on the isolated edges is different: Algorithm 3.9 computes the decision on the isolated edges with an algorithm solving the string guessing problem with known history.
Recall that an online path instance on n vertices contains at most $\lceil \frac{n}{3} \rceil$ isolated edges. Let

$$e_1, e_2, \ldots, e_k, \text{ for some } k \in \left\{1, 2, \ldots, \left\lceil \frac{n}{3} \right\rceil \right\},$$

be these isolated edges.
Now, we transform the decision on these isolated edges to the problem of finding a bit string $b = b_1b_2 \ldots b_{\lceil \frac{n}{3} \rceil}$ with

$$b_i = \begin{cases} 1, & \text{if } e_i \text{ is an edge in } M^*, \\ 0, & \text{if } e_i \text{ is not present in } M^*, \end{cases}$$

for all $i \in \{1, 2, \ldots, k\}$ and $b_i = 0$ for all non-relevant bits $b_{k+1}, \ldots, b_{\lceil \frac{n}{3} \rceil}$. See Figure 3.57 for an example. Algorithm 3.9 describes the algorithm using string guessing with known history to set the isolated edges in detail. □

Now, we want to analyze Algorithm 3.9 in order to give an upper bound on the tradeoff for the online matching problem. For this, we need an online algorithm described in [8, 9] for solving the string guessing problem with known history.

Lemma 3.62 (Böckenhauer et al. [8]). *Let b be a bit string of length m. There is an online algorithm reading at most*

$$\left\lceil (1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha))\, m + \frac{3}{2}\log(m) + \frac{1}{2} + \log(\ln(2)) \right\rceil$$

advice bits in order to guess αm bits of b correctly, for some $\frac{1}{2} \le \alpha < 1$.

Algorithm 3.9 Matching on Paths Using a Bit String

INPUT: $P^{\prec} \in \mathcal{P}_n$, for some $n \in \mathbb{N}$

1: $M = \varnothing$
2: Calculate $b = b_1 b_2 \ldots b_{\lceil \frac{n}{3} \rceil}$ by an algorithm for the string guessing problem with known history using the amount of advice specified in Theorem 3.63 to achieve the desired number of correct guesses
3: **for** $i = 1$ to n **do**
4:     **if** (a) v_i has exactly one edge e to a previously isolated vertex **then**
5:         read the next bit in b to decide whether e is a matching edge:
6:         **if** $\sigma = 1$ **then**
7:             $M \leftarrow M \cup \{e\}$
8:         **else**
9:             $M \leftarrow M$
10:     **else if** (b) v_i has two edges e_1 and e_2 to prev. isolated vertices **then**
11:         use the next bit in b to decide whether e_1 or e_2 is a matching edge:
12:         **if** $\sigma = 0$ **then**
13:             $M \leftarrow M \cup \{e_1\}$
14:         **else**
15:             $M \leftarrow M \cup \{e_2\}$
16:     **else if** (c) v_i is connected to some non-isolated and unmatched vertex by an edge e **then**
17:         $M \leftarrow M \cup \{e\}$
18:     **else if** (d) v_i has an edge e_1 to a matched vertex and an edge e_2 to an isolated vertex **then**
19:         $M \leftarrow M \cup \{e_2\}$
20:     **else**
21:         $M \leftarrow M$
22:     **output** $M_i = M$

OUTPUT: $M = (e_{i_1}, e_{i_2}, \ldots, e_{i_m}) = \bigcup_{i=1}^{n} M_i \subseteq E$, for some $m \in \mathbb{N}$

Applying this result to Algorithm 3.9 leads to an upper bound on the number of advice bits the algorithm needs to achieve a desired size of the matching.

Theorem 3.63. *Algorithm 3.9 finds a matching on*

$$\left\lfloor \frac{n}{2} \right\rfloor - (1-\alpha) \left\lceil \frac{n}{3} \right\rceil$$

edges in a path on n vertices using at most

$$\left\lceil (1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha)) \left\lceil \frac{n}{3} \right\rceil + \frac{3}{2}\log(\left\lceil \frac{n}{3} \right\rceil) + \frac{1}{2} + \log(\ln(2)) \right\rceil$$

advice bits for some $\frac{1}{2} \leq \alpha < 1$.

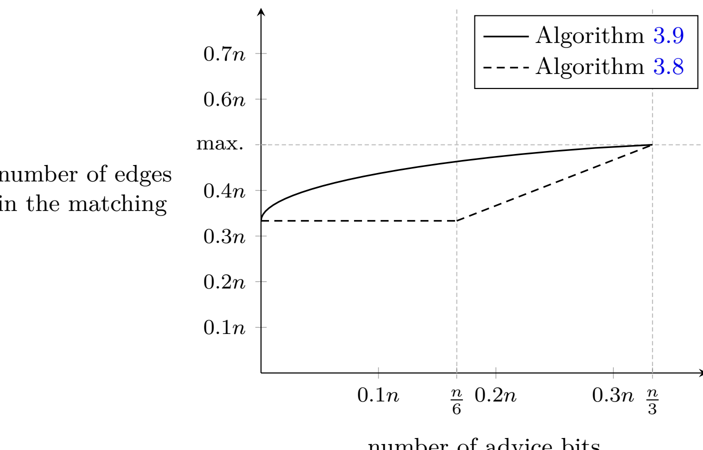


Figure 3.58. Minimum number of edges found by Algorithm 3.9 in an arbitrary online path on n edges with respect to the number of advice bits that are on disposal.

Proof. Guessing αm bits correctly means guessing $(1-\alpha)m$ bits wrongly, implying $(1-\alpha)m$ wrongly set isolated edges in the online path on n vertices with $m = \lceil \frac{n}{3} \rceil$ with respect to a fixed maximum matching M^* that matches all inner vertices. We showed in the proof of Theorem 3.58 that every wrongly set isolated edge leads to exactly two unmatched vertices if the adversary can arrange the isolated edges such that every correctly set edge lies between two wrongly set edges. This implies directly that every wrongly set isolated edge leads to exactly one edge less in the matching computed by the algorithm, with respect to the fixed maximum matching M^*. To arrange the isolated edges in this way is possible as long as we have less than half of the isolated edges set wrong, i. e., if $\alpha \geq \frac{1}{2}$ holds.

Therefore, we can use Lemma 3.62 and and the fact that a path on n vertices has a matching containing $\lfloor \frac{n}{2} \rfloor$ matching edges to show that Algorithm 3.9 finds a matching of size

$$\left\lfloor \frac{n}{2} \right\rfloor - (1-\alpha)m = \left\lfloor \frac{n}{2} \right\rfloor - (1-\alpha)\left\lceil \frac{n}{3} \right\rceil$$

for some $\frac{1}{2} \leq \alpha < 1$, using

$$\left\lceil (1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha))\left\lceil \frac{n}{3} \right\rceil + \frac{3}{2}\log\left(\left\lceil \frac{n}{3} \right\rceil\right) + \frac{1}{2} + \log(\ln(2)) \right\rceil$$

advice bits. This function is visualized in Figure 3.58. □

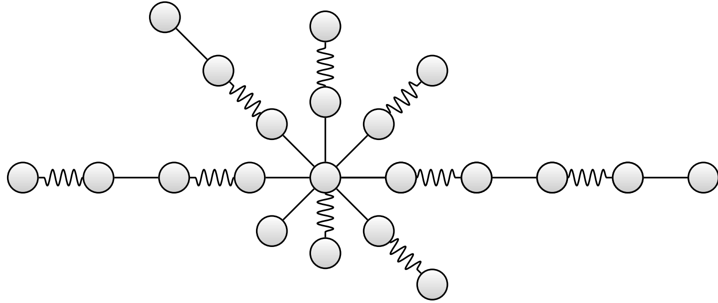


Figure 3.59. Example of a matching in a spider graph.

A Lower Bound on the Online Matching Problem in Trees

To show a lower bound on the tradeoff for the online matching problem on trees, we will use spider graphs (see Definition 2.23). Recall that a spider graph is a special tree where all vertices except for one have degree at most 2.
As shown in Lemma 3.7, a path on n vertices contains a matching of size $\lfloor \frac{n}{2} \rfloor$. In the case of n even, the matching is unique whereas there are $\frac{n+1}{2}$ maximum matchings for a path with n odd. In a spider graph, the maximum matching depends on how the vertices are distributed to legs of even and odd lengths (see Figure 3.59). Since at most one matching edge is incident to the center, the legs of the spider graph can be seen as independent paths. Every leg of an even number k of vertices contributes $\frac{k}{2}$ edges to the matching and every leg of an odd number l of vertices, except of one, contributes exactly $\frac{l-1}{2}$ edges to the matching. The leg which is adjacent to the matching edge incident to the center contains, including the center edge, $\frac{l+1}{2}$ edges. As shown in Figure 2.15, we have an unmatched vertex in every path of an odd number of vertices except in one. Hence, a spider graph with i legs containing an odd number of vertices has a maximum matching of size

$$\frac{n}{2} - \frac{i-1}{2} = \frac{n-i+1}{2} .$$

To give a lower bound on the number of advice bits used in order to achieve a certain fixed competitive ratio on finding a maximum matching in trees, we reduce the string guessing problem with unknown history to the online matching problem in special spider graphs.

Lemma 3.64. *There exists an advice-preserving reduction from the string guessing problem with unknown history to the online matching problem on spider graphs with legs of 4 vertices, except of one leg consisting of exactly one vertex.*

Proof. Let $b = b_1 b_2 b_3 \dots b_m$ be the bit string that should be guessed from some input $B = ((m, ?_1), ?_2, ?_3, \dots, ?_m, b)$ for the string guessing problem with unknown history, for some $m \in \mathbb{N}$. We transform the problem of guessing this bit string b

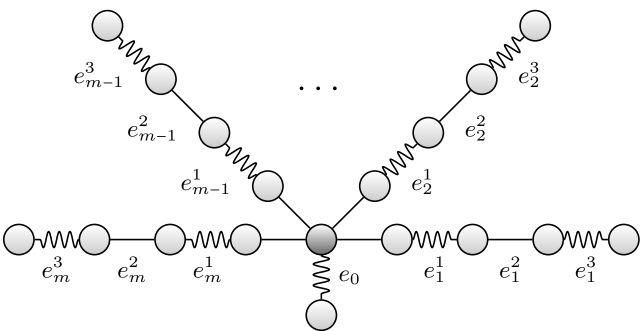


Figure 3.60. A spider graph containing exactly one maximum matching, used for the advice-preserving reduction.

to the problem of finding a maximum matching in a special online spider graph on $n = 4m + 2$ vertices as shown in Figure 3.60. This spider graph has exactly one maximum matching because of the leg containing 1 vertex. Since this spider graph contains $m - 1$ legs of length 4 and one of length 1, the size of the maximum matching is

$$2m + 1 = 2 \cdot \left(\frac{n-2}{4}\right) + 1 = \frac{n}{2} \,.$$

In the online presentation of this spider graph, $S_n^{\prec}$ with $n = 4m + 2$, the adversary shows the isolated edges first, according to the bit string b. The ith edge e_i in the online presentation is defined as

$$e_i = \begin{cases} e_i^1 & \text{if } b_i = 1\,, \\ e_i^2 & \text{if } b_i = 0\,. \end{cases}$$

Now, we have to show how a solution to the online matching problem on such a spider graph $S_n^{\prec}$ transforms to a solution for the string guessing problem with unknown history on b. If e_i is supposed to be a matching edge in $S_n^{\prec}$, then the according bit b_i is set to 1, otherwise, i. e., if e_i is not a matching edge, $b_i = 0$, in other words,

$$b_i = \begin{cases} 1 & \text{if } e_i \text{ is a matching edge}\,, \\ 0 & \text{else}\,. \end{cases}$$

We want to show that every wrongly set edge in $S_n^{\prec}$ leads to exactly one matching edge less with respect to the maximum matching, and therefore a wrongly set bit in b corresponds directly to one lost matching edge in $S^{\prec}$. If e_i^1 would wrongly be set as non-matching, only one of the edges e_i^2 or e_i^3 can be a matching edge, leading to a matching edge less on the leg containing e_i^1. Assigning wrongly e_i to be a matching edge, prevents both, e_i^1 and e_i^2, to be matching edges, and since the edge next to the center also cannot be a matching edge because of the edge e_0, we

again lose one matching edge with respect to the unique maximum matching on the corresponding leg.

Summarizing, if an algorithm for the online matching problem computes a maximum matching on $\leq \frac{n}{2} - k$ edges, then there is also an algorithm for string guessing admitting k mistakes and using the same amount of advice. Therefore, if an algorithm for solving the string guessing problem needs at least $f_\alpha(m)$ advice bits to guarantee guessing αm bits correctly, for some $\frac{1}{2} \leq \alpha < 1$, then every algorithm for the online matching problem on this special spider graphs needs at least $f_\alpha\left(\frac{n-2}{4}\right)$ advice bits to guarantee a matching on

$$\frac{n}{2} - (1-\alpha)m = \frac{n}{2} - (1-\alpha) \cdot \frac{n-2}{4}$$

edges. □

Since a lower bound on the string guessing problem with known history directly implies a lower bound on the string guessing problem with unknown history, we can use the lower bound of [8] to give a lower bound for the online matching problem on trees.

Lemma 3.65 (Böckenhauer et al. [8]). *Let b be a bit string of length m. Every deterministic online algorithm for the string guessing problem with known history that can guarantee to be correct for more than αm bits, for some $\frac{1}{2} \leq \alpha < 1$, needs to read at least*

$$(1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha))\, m$$

advice bits. □

This leads to a lower bound on the number of advice bits needed by every online algorithm solving the online matching problem on trees with a fixed competitive ratio.

Theorem 3.66. *Every deterministic online algorithm that guarantees a matching of size*

$$\frac{n}{2} - (1-\alpha) \cdot \frac{n-2}{4},$$

for an $\frac{1}{2} \leq \alpha < 1$, needs to read at least

$$(1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha)) \cdot \frac{n-2}{4}$$

advice bits.

Proof. Due to Lemma 3.64, a matching edge less than the unique maximum matching in the special spider graphs described in the proof implies directly a mistake in the string guessing problem. Therefore the lower bound on the number of

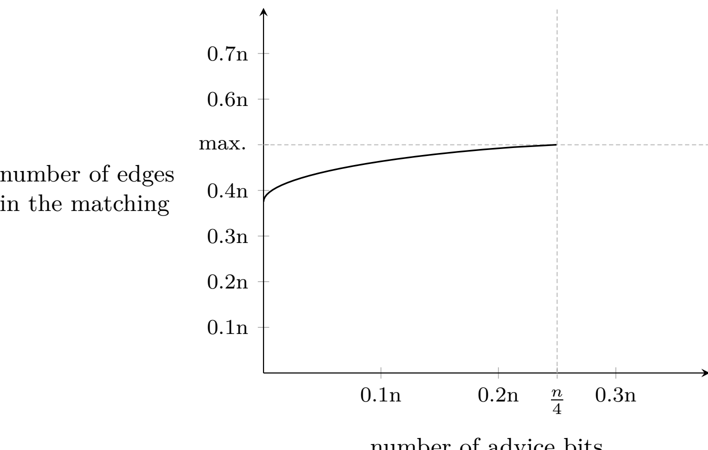


Figure 3.61. The lower bound from Theorem 3.66 on the number of advice bits every online algorithm solving the online matching problem on trees needs to read in order to find a matching of fixed size.

correct guesses in the string guessing problem with known history directly transfers to the online matching problem on these special trees. Hence, Lemma 3.65 leads to

$$f_\alpha\left(\frac{n-2}{4}\right) = (1 + (1-\alpha)\log(1-\alpha) + \alpha\log(\alpha)) \cdot \frac{n-2}{4}$$

advice bits that every online algorithm has to read in order to get a maximum matching of size $\frac{n}{2} - (1-\alpha)\cdot\frac{n-2}{4}$. This lower bound is depicted in Figure 3.61. □

3.8 Conclusion

We considered the online matching problem mainly on bipartite graphs and some of its subclasses. For general graphs, we constructed a deterministic online algorithm with advice which solves the problem optimally using $\mathcal{O}(n\log(n))$. We complemented this result with a matching lower bound that holds already for P_5-free bipartite graphs with diameter 3.
For paths and cycles, we showed an almost tight upper and lower bound of approximately $\frac{n}{3}$ advice bits for optimality. We also investigated the case of $\{1,3\}$-trees, where we showed a lower bound of approximately $\frac{n}{2}$ and an upper bound of approximately $\frac{\log(3)}{3}n$. It would be interesting to extend these results to more general trees, starting with $\{1,k\}$-trees and general degree-bounded trees.
For paths, we also investigated the tradeoff of the number of advice bits and the competitive ratio using a reduction from string guessing. Moreover, we showed that one single advice bit does not help at all. It is open which ratio can be reached

with two or a few more advice bits. Also the tradeoff for other graph classes could be of interest.

Approximability of Splitting-SAT in 2-CNF Horn Formulas

Many problems arising from practical applications can be formulated using Boolean formulas in conjunctive normal form (CNF). Usually, the variables of the formula model some parameters of the problem, and the constraints of the problem are modeled by the clauses of the formula. The goal is to find out a valid parameter setting by computing a satisfiable assignment for the corresponding formula. But often the modeling of the practical situation is very complex, leading to some contradictory constraints in the model, and the corresponding formula turns out to be unsatisfiable. In this case, one often tries to find a maximum set of constraints that can be simultaneously satisfied. This leads to the well-known MAX-SAT problem (see, e. g., [3] for an overview of the known results for MAX-SAT). Another source of mistakes that might arise when modeling a real-world problem as a Boolean formula is a too coarse-grained choice of parameters, i. e., variables. If two different parameters are erroneously modeled by the same variable, this might also lead to an unsatisfiable formula. In other words, an unsatisfiable formula might contain one or more variables that should be *split* into two variables in order to make the formula satisfiable. The *minimum splitting SAT problem* formalizes this approach, the input is an (unsatisfiable) CNF-formula and the goal is to find a minimum number of variables that have to be split into two to make the resulting formula satisfiable.

The splitting operation has not only been considered on formulas. For example, it arises in the context of vertex splitting in phylogenetic tree construction [55]. Splitting vertices in a graph was also considered for making a graph Hamiltonian [62]. To the best of our knowledge, splitting variables in a Boolean formula was introduced by Steinová [62], who showed that the minimum splitting SAT problem is $\mathcal{APX}$-

hard even for formulas in 2-CNF, i. e., when restricted to formulas in which each clause contains at most two literals.
In a 2-CNF formula, we can have the following five types of clauses:

P1: Positive 1-clauses (x) consisting of one positive literal,

N1: negative 1-clauses $(\overline{x})$ consisting of one negative literal,

M2: mixed 2-clauses $(x \vee \overline{y})$ consisting of one positive and one negative literal,

N2: negative 2-clauses $(\overline{x} \vee \overline{y})$ consisting of two negative literals, and

P2: positive 2-clauses $(x \vee y)$ consisting of two positive literals.

A 2-CNF formula without positive 2-clauses is called a 2-CNF *Horn formula*. We will restrict our attention to 2-CNF Horn formulas in the first part of this chapter. We analyze which combinations of clause types make the minimum splitting SAT problem hard to approximate. An overview of the results is given in Figure 4.1. Observe that lower bounds carry over upwards and upper bounds downwards in the lattice of subsets. In particular, we show that the minimum splitting SAT problem remains exactly as hard to approximate as the vertex cover problem, when restricted to the special case of Horn formulas consisting of clauses of type P1 and N2 only. On the other hand, even when allowing additional clauses of type N1, the problem can be approximated exactly as good as the vertex cover problem, it becomes polynomially solvable when restricted to Horn formulas consisting of clauses of type P1, N1, and M2.
Another way to look at the splitting SAT problem is to ask for the maximum number of variables that can be assigned a truth value without evaluating any clause to 0, i. e., for the maximum number of variables that can be left unsplit. This is called the *maximum assignment SAT problem*. Obviously, the optimal solutions for minimum splitting SAT and maximum assignment SAT coincide, but we show that the approximability of the two problems essentially differs. The maximum assignment SAT problem on 2-CNF Horn formulas with clauses of type P1 and N2 turns out to be as hard to approximate as the maximum independent set problem, and, on arbitrary 2-CNF Horn formulas, it can be approximated as good as the maximum independent set problem. An overview of the results on the maximum assignment SAT problem is shown in Figure 4.2.
We complement our results with an approximation algorithm for the maximum assignment SAT problem on E2-CNF formulas, i. e., formulas containing only clauses of the types M2, P2, and N2. The approximation hardness of this problem was shown by Steinová [62].
This chapter is organized as follows: In Section 4.1, we fix our notation. Sections 4.2 and 4.3 are devoted to the analysis of minimum splitting SAT and maximum assignment SAT in Horn formulas, respectively. In Section 4.4, we discuss the case of E2-CNF formulas, and we conclude this chapter in Section 4.5.

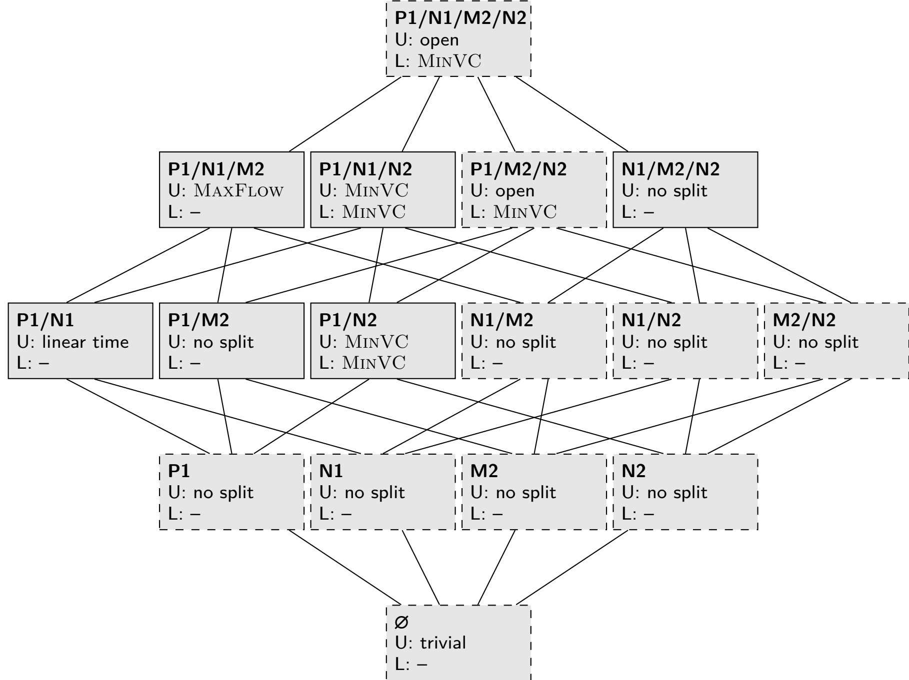


Figure 4.1. Upper and lower bounds on the approximability of the minimum splitting SAT problem on 2-CNF Horn formulas for each set of allowed clause types

4.1 Basic Definitions

We start with formally defining the minimum splitting SAT problem and the maximum assignment SAT problem. We follow the definitions from [62].

Definition 4.1 (Splitting). Let Φ be a Boolean formula over the variable set X and let $y, z \notin X$ be two new variables. We say that a variable $x \in X$ is *split* if each occurrence of x in Φ is replaced by either y or z and each occurrence of $\overline{x}$ is replaced by either $\overline{y}$ or $\overline{z}$. This operation is called a *splitting* of x. We call a set $X' \subseteq X$ such that splitting all variables from X' yields a satisfiable formula a *feasible splitting set* (or *splitting set* for short).

Note that, when splitting a variable x into the two new variables y and z, we can replace all occurrences of the literal x by y and all occurrences of the literal $\overline{x}$ by $\overline{z}$. Thus, the resulting formula is satisfiable if and only if the formula resulting from removing all clauses containing the variable x is satisfiable. Hence, we can think of a splitting operation as the removal of the split variable (together with all clauses it appears in) from the formula. Furthermore, note that the result of

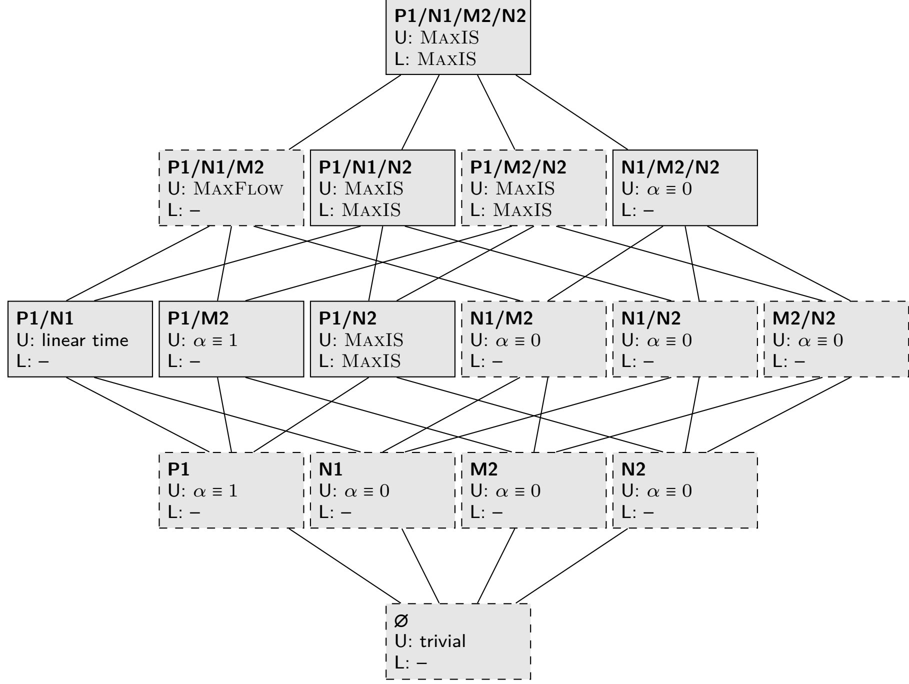


Figure 4.2. Upper and lower bounds on the approximability of the maximum assignment SAT problem on 2-CNF Horn formulas for each set of allowed clause types

splitting multiple variables is independent from the order of applying the splitting operations.

Definition 4.2 (Minimum Splitting SAT Problem). The *minimum splitting SAT problem*, MinSplit-SAT for short, is the following minimization problem:

Input: A Boolean formula in CNF over the variable set $X = \{x_1, \dots, x_n\}$.

Set of feasible solutions: A feasible splitting set $X' \subseteq X$.

Cost: Number of split variables.

Goal: Minimum.

If the input is restricted to k-CNF formulas, we call the resulting subproblem MinSplit-k-SAT.

In this chapter, we will consider the following subclass of 2-CNF formulas.

Definition 4.3 (Horn Formulas). For a formula in 2-CNF and a set S of clause types from $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}, \mathrm{P2}\}$ as defined above, we say that Φ is in S-2-CNF, if it contains only clauses of types from S. The $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}\}$-2-CNF formulas are called *Horn formulas.*

Definition 4.4 (MinSplit-S-2-SAT). When we restrict, for some

$$S \subseteq \{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}, \mathrm{P2}\},$$

the input to S-2-CNF formulas, the resulting subproblem is denoted as MinSplit-S-2-SAT. The problem MinSplit-$\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}, \mathrm{N2}\}$-2-SAT is referred to as MinSplit-Horn-2-SAT and MinSplit-$\{\mathrm{P2}, \mathrm{M2}, \mathrm{N2}\}$-2-SAT is called MinSplit-E2-SAT.

In the following, we will only deal with MinSplit-2-SAT and its subproblems and we will assume without loss of generality that the input always is an unsatisfiable formula. Since 2-SAT, i.e., checking the satisfiability of a 2-CNF formula, is solvable in polynomial time, this is no severe restriction. This implies that the cost of any feasible solution is always strictly greater than zero, which allows us to consider the approximation ratio of an algorithm for MinSplit-2-SAT as the quotient of the cost of the computed solution and the optimal cost.

Definition 4.5 (Maximum Assignment SAT Problem). The *maximum assignment SAT problem*, MaxAssign-SAT for short, is the following maximization problem:

Input: A Boolean formula in CNF over the variable set $X = \{x_1, \ldots, x_n\}$.

Set of feasible solutions: A subset $X' \subseteq X$ such that there exists a partial assignment $\alpha : X' \to \{0, 1\}$ such that no clause is evaluated to 0 under this partial assignment.

Cost: Size of the subset.

Goal: Maximum.

For the restrictions to special types of clauses, we use analogous notations as for the respective MinSplit-SAT variants.

Observation 4.6. *Let Φ be a formula in CNF over the set $X = \{x_1, \ldots, x_n\}$ of variables. A set $X' \subseteq X$ is an optimal* MinSplit-SAT *solution for Φ if and only if $X - X'$ is an optimal* MaxAssign-SAT *solution for Φ.*

Proof. Let X' be a feasible solution for MinSplit-SAT on Φ. Then all clauses are either satisfied by splitting one of the variables from X' or can be satisfied by some

partial assignment α for the variables from $X - X'$ (otherwise another variable would need to be split). Thus, $X - X'$ is a feasible MaxAssign-SAT solution for Φ since all clauses not satisfied by α contain an unassigned variable from X' and thus are not evaluated to 0.
On the other hand, let $X - X'$ be a feasible MaxAssign-SAT solution for Φ. Then there exists a partial assignment α to the variables from $X - X'$, such that no clause evaluates to 0 under α. Thus, each clause is either satisfied by α or contains an unassigned variable from X'. Thus, splitting all variables from X' makes the formula satisfiable. □

Several of our results are based on reductions from the minimum vertex cover problem and the maximum independent set problem. The *minimum vertex cover problem*, MinVC for short, is the following minimization problem: Given an undirected graph $G = (V, E)$, find a minimum-size vertex cover of G, i. e., a minimum-size subset $C \subseteq V$ such that $e \cap C \neq \varnothing$ for all $e \in E$. The *maximum independent set problem*, MaxIS for short, is the following maximization problem: Given an undirected graph $G = (V, E)$, find a maximum-size independent set of G, i. e., a maximum-size subset $I \subseteq V$ such that $\{x, y\} \notin E$ for all $x, y \in I$ with $x \neq y$.
MinVC is known to be approximable within a factor of $2 - \frac{\log(\log(|V|))}{2\log(|V|)}$ [52], but it is $\mathcal{APX}$-hard [56] and not approximable within a factor of $2 - \varepsilon$, for any constant $\varepsilon > 0$, if the Unique Games Conjecture holds [42]. The MaxIS is approximable within $O\left(\frac{|V|}{(\log(|V|))^2}\right)$ [13], but not approximable within $|V|^{1-\varepsilon}$, for any $\varepsilon > 0$, unless $\mathcal{P} = \mathcal{NP}$ [35].

4.2 Splitting in 2-CNF Horn Formulas

In this section, we deal with the approximability of MinSplit-S-2-SAT, for all possible subsets $S \subseteq \{\text{P1}, \text{N1}, \text{M2}, \text{N2}\}$.
If all allowed clause types contain a positive literal or all contain a negative literal, setting all variables to 1 or 0, respectively, satisfies the formula and no splitting is needed.

Observation 4.7. *For $S = \{\text{P1}, \text{M2}\}$ or $S = \{\text{N1}, \text{M2}, \text{N2}\}$ or any subset thereof,* MinSplit-S-2-SAT *is solvable in constant time.* □

Similarly, as already observed in [62], for any formula consisting of 1-clauses only, MinSplit-S-2-SAT is easily solvable. This immediately leads to the following observation.

Observation 4.8. *For $S = \{\text{P1}, \text{N1}\}$,* MinSplit-$S$-2-SAT *is solvable in linear time.* □

Next, we prove that MinSplit-$\{\text{P1}, \text{N1}, \text{M2}\}$-2-SAT can be solved by computing the maximum flow in a given network. For this, we first define a representation of

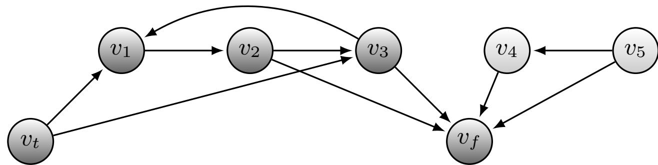


Figure 4.3. The graph representation of the {P1, N1, M2}-2-CNF Horn formula $\Phi = (x_1) \wedge (\overline{x_2}) \wedge (x_3) \wedge (\overline{x_3}) \wedge (\overline{v_4}) \wedge (\overline{v_5}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_3}) \wedge (v_4 \vee \overline{v_5})$

formulas of this type by directed graphs. This graph representation is a special case of the representation in [23].

Definition 4.9 (Graph Representation of a Horn Formula). Given a {P1, N1, M2}-2-CNF Horn formula $\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ over the variable set $X = \{x_1, x_2, \ldots, x_n\}$, $G_\Phi = (V_\Phi, E_\Phi)$ is a digraph with vertex set $V_\Phi = \{v_i \mid x_i \in X\} \cup \{v_t\} \cup \{v_f\}$ and arc set

$$E_\Phi = \{(v_t, v_j) \mid (x_j) \text{ in } \Phi\} \cup \{(v_j, v_f) \mid (\overline{x_j}) \text{ in } \Phi\} \cup \{(v_i, v_j) \mid (\overline{x_i} \vee x_j) \text{ in } \Phi\}.$$

The arcs from v_t to vertices $v_i \in \{v_1, v_2, \ldots, v_n\}$ represent a satisfying assignment $\alpha(x_i) = 1$ for all clauses of type (x_i). The connecting arcs in the vertex set $\{v_1, v_2, \ldots, v_n\}$ indicate that, for an arc (v_i, v_j), $\alpha(x_i) = 0$ or $\alpha(x_j) = 1$ has to hold in order to satisfy the clauses of type $(\overline{x_i} \vee x_j)$. An arc from a vertex $v_i \in \{v_1, v_2, \ldots, v_n\}$ to v_f represents a satisfying assignment $\alpha(x_i) = 0$ for the corresponding clause $(\overline{x_i})$. We see in Figure 4.3 that only the shaded vertices can be reached by a path from v_t. We call such a vertex a v_t*-pebbled* vertex or *pebbled vertex* for short.
This graph construction and the idea of a pebbling were used in a more general way by Dowling and Gallier [23]. They proved the following theorem.

Theorem 4.10 (Downing and Gallier [23]). *A Horn formula is satisfiable if and only if there is no directed path from* v_t *to* v_f*.* □

Furthermore, they made a statement about the assignment in the case of a satisfiable formula.

Theorem 4.11 (Downing and Gallier [23]). *Let* Φ *be a satisfiable Horn formula with variable set* $X = \{x_1, x_2, \ldots, x_n\}$*. The assignment* $\alpha(x_i) = 1$ *if and only if* v_i *is pebbled and* $\alpha(x_i) = 0$ *otherwise, is a satisfying assignment.* □

This means that we get a satisfying assignment if we set all variables corresponding to pebbled vertices to 1 and all other vertices to 0 in the case with no directed path from v_t to v_f.

Corollary 4.12. *Only pebbled vertices are candidates to split.*

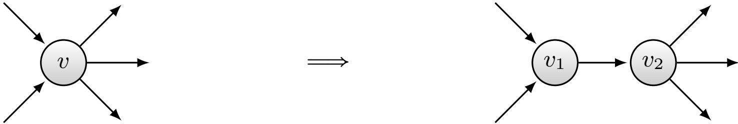


Figure 4.4. In the transformation of an instance G for finding a minimum $s-t$ vertex cut into an instance G' for finding a minimum $s-t$ arc cut, every vertex v is replaced by two vertices v_1 and v_2 and the arcs are adjusted as shown above.

Proof. There is no directed path from v_t to a non-pebbled vertex v_i and therefore no 1-clause (x_i). Furthermore, for all variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ that correspond to vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ for which a directed path from $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_i exists, there are no 1-clauses $(x_{i_1}), (x_{i_2}), \ldots, (x_{i_k})$. Therefore, all variables that correspond to non-pebbled vertices can be set to 0 so that all clauses containing those vertices are satisfied by this assignment. □

If we remove all non-pebbled vertices from the graph, it remains connected with a directed path from v_t to v_f. To make the corresponding formula satisfiable, we have to delete some of the remaining vertices in order to disconnect v_t from v_f.

Lemma 4.13. *A splitting set of size k in a* $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$*-2-CNF Horn formula corresponds to a $v_t - v_f$ vertex cut of size k in the corresponding graph.*

Proof. After removing the split vertices from the formula Φ, the corresponding graph G_Φ has no directed path from v_t to v_f due to Theorem 4.10. Hence, the removed vertices form a $v_t - v_f$ vertex cut.
Conversely, removing the variables corresponding to a $v_t - v_f$ vertex cut in G_Φ from the formula Φ makes it satisfiable due to Theorem 4.10. □

The problem of finding a minimum $s-t$ vertex cut in a graph G equals the problem finding a minimum $s-t$ arc cut in a graph G' where we replace every vertex v of G, except the source s and the sink t, by two vertices v_1 and v_2 and an arc (v_1, v_2) in G'. The ingoing arcs of v are connected to v_1 in G' and the outgoing arcs of v are outgoing arcs of v_2 (see Figure 4.4). Additionally, the new arcs in G' get capacity 1 and the old ones capacity $2n+1$ such that in a minimum arc cut the new arcs will be chosen.
According to the well-known maxflow-mincut theorem [18], the problem of finding a minimum $s-t$ arc cut in a graph G equals the problem of finding a maximum value of a $s-t$ flow in G.

Corollary 4.14. *The problem of finding a splitting in a* $\{\mathrm{P1}, \mathrm{N1}, \mathrm{M2}\}$*-2-CNF Horn formula equals the problem of finding a maximum flow in a graph G'_Φ.*

Since the graph G'_Φ and its capacities are of polynomial size with respect to the formula size, the discussion above immediately yields the following theorem.

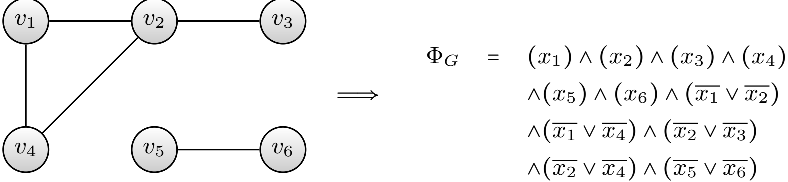


Figure 4.5. An example of the construction used in the proof of Theorem 4.16

Theorem 4.15. MinSplit-{P1, N1, M2}-2-SAT *is polynomial-time solvable.* □

Next, we show that MinSplit-{P1, N2}-2-SAT is as hard to approximate as MinVC. This result immediately implies that all remaining subcases of Horn formulas are also as hard to approximate as MinVC since they are generalizations of MinSplit-{P1, N2}-2-SAT.

Theorem 4.16. MinVC $\leq_{\mathrm{AP}}$ MinSplit-{P1, N2}-2-SAT.

Proof. We present an AP-reduction from MinVC to MinSplit-{P1, N2}-2-SAT. For this, we give a polynomial-time function transforming any MinVC instance G into a MinSplit-{P1, N2}-2-SAT instance Φ_G such that any α-approximate feasible solution for Φ_G can be transformed in polynomial time into a feasible solution for G achieving the same approximation ratio. For more details on the general concept of AP-reductions, see [3, 37].
Let G be a MinVC instance, where $G = (V, E)$ is an undirected graph. Let $V = \{v_1, \ldots, v_n\}$. We construct a formula Φ_G from G as follows: Φ_G contains a positive 1-clause (x_i) for each non-isolated vertex $v_i \in V$ and a negative 2-clause $(\overline{x_i} \vee \overline{x_j})$ for each edge $\{v_i, v_j\} \in E$. An example of this construction is shown in Figure 4.5.
We now show that every vertex cover in G corresponds to a feasible set of split variables of the same size in Φ_G and vice versa.
Let $C = \{v_{i_1}, \ldots, v_{i_k}\}$ be a vertex cover of G of size k. We consider the corresponding variable set $X_C = \{x_{i_1}, \ldots, x_{i_k}\}$ in Φ_G. Following the construction, since C is a vertex cover, every 2-clause in Φ_G contains at least one variable from X_C. Thus, splitting the variables from X_C removes all 2-clauses and the remaining formula consists of positive 1-clauses only and hence is obviously satisfiable.
Let, on the other hand, $X = \{x_{i_1}, \ldots, x_{i_k}\}$ be a set of variables whose splitting makes Φ_G satisfiable. Since there exists a positive 1-clause for each variable in Φ_G, every partial assignment setting any variable to 0 violates at least one of these 1-clauses. Thus, every variable that remains unsplit has to be assigned the value 1. This means that every 2-clause in Φ_G has to contain at least one variable from X. We consider the corresponding set $C_X = \{v_{i_1}, \ldots, v_{i_k}\}$ of vertices in G. Due to the construction, C_X is a vertex cover of G of size k.
Summing up, there is a one-to-one correspondence between vertex covers for G and feasible solutions for Φ_G of the same size proving our claim. □

We conclude this section with showing that MinSplit-{P1, N1, N2}-2-SAT can be approximated as good as MinVC. The upper bound on the approximability of MinSplit-{P1, M2, N2}-2-SAT and MinSplit-Horn-2-SAT remains open. For these cases, we only know about an $O\left(\frac{n}{\log(n)}\right)$-approximative algorithm due to Mömke that was mentioned in [62].

Theorem 4.17. *Any polynomial-time α-approximation algorithm for* MinVC *can be used to approximate* MinSplit-{P1, N1, N2}-2-SAT *within a factor of α in polynomial time.*

Proof. We first preprocess the {P1, N1, N2}-CNF formula Φ in order to receive a {P1, N2}-CNF formula Φ'. We first remove all clauses containing variables x_i with (x_i) and $(\overline{x_i})$ in Φ and add those variables to the splitting set. All remaining variables x_i with $(\overline{x_i})$ in Φ can be set to 0 such that no clause is violated and all clauses containing the variable x_i are satisfied. Thus, we remove all clauses containing those variables x_i. After that, we set all variables x_i occurring only positively or only negatively in Φ to the value 1 or 0, respectively. The remaining formula Φ' contains only clauses of type P1 and N2 because of the construction, and every variable occurs in a positive 1-clause.
Now, we present a reduction from MinSplit-{P1, N2}-2-SAT to MinVC. Let Φ' be a {P1, N2}-CNF formula with variable set $X = \{x_1, \dots, x_n\}$. Then, $G_{\Phi'}$ is a graph with vertex set $V_{\Phi'} = \{v_i \mid (x_i) \text{ in } \Phi'\}$ and edge set $E_{\Phi'} = \{\{v_i, v_j\} \mid (\overline{x_i} \vee \overline{x_j}) \text{ in } \Phi'\}$. Note that, since every possible positive 1-clause is present in Φ', this is exactly the reverse of the construction used in the proof of Theorem 4.16, where we have already proven the one-to-one correspondence between feasible splitting sets for Φ' and vertex covers for $G_{\Phi'}$. This proves our claim. □

4.3 Maximum Assignment in 2-CNF Horn Formulas

In this section, we deal with the approximability of MaxAssign-Horn-2-SAT and its subproblems. According to Observation 4.6, every polynomial-time algorithm for minimum splitting immediately yields a polynomial-time algorithm for maximum assignment. Hence, the results of Observation 4.7 and Theorem 4.15 directly carry over to MaxAssign-Horn-2-SAT.

Observation 4.18. *For S = {P1, M2} or S = {P1, N1} or S = {N1, M2, N2} or any subset thereof,* MaxAssign-S-2-SAT *is solvable in linear time.* □

Theorem 4.19. MaxAssign-{P1, N1, M2}-2-SAT *is polynomial-time solvable.* □

It is well known that, if C is a vertex cover of size k in a graph $G = (V, E)$ with $|V| = n$, then $V - C$ is an independent set of size $n - k$ in G. This strong correspondence between MinVC and MaxIS resembles the correspondence between

minimum splitting and maximum assignment. Thus, we can use similar ideas as in the previous section to prove that MaxAssign-{P1, N2}-2-SAT is as hard to approximate as MaxIS and that MaxAssign-Horn-2-SAT can be approximated using MaxIS algorithms.

Theorem 4.20. *Unless $\mathcal{P} = \mathcal{NP}$,* MaxAssign-{P1, N2}-2-SAT *cannot be better approximated than* MaxIS*.*

Proof. We use the same reduction as in the proof of Theorem 4.16 to transform a given MaxIS instance G into a MaxAssign-{P1, N2}-2-SAT instance Φ_G. Following the discussion about the relation of MinVC and MaxIS above, it is easy to see that every independent set in G corresponds to a set of variables in Φ_G of the same size which can be assigned the truth value 1 without generating an unsatisfied clause. □

Theorem 4.21. *Any polynomial-time $f(n)$-approximation algorithm for* MaxIS *can be used to approximate* MaxAssign-Horn-2-SAT *within $f(2 \cdot n)$ in polynomial time, where n denotes the number of vertices or variables, respectively.*

Proof. Let Φ be an input instance for MaxAssign-Horn-2-SAT. We start with a preprocessing of Φ. If there exists a variable x that occurs both in a positive and a negative 1-clause, then x obviously cannot be assigned any truth value, thus, we delete all clauses containing x from the formula. If, for some variable x and some literal l, there exist two clauses $(\overline{x})$ and $(\overline{x} \vee l)$, we can remove the clause $(\overline{x} \vee l)$. This is due to the fact that any partial assignment that does not contradict $(\overline{x})$ also does not contradict $(\overline{x} \vee l)$. Analogously, for any two clauses (x) and $(x \vee \overline{y})$, we can remove the clause $(x \vee \overline{y})$. Finally, if some variable appears only positively or only negatively in the formula, we can assign the respective value to it and shrink the formula accordingly.

For the remainder of the proof, let $\Phi = C_1 \wedge \ldots \wedge C_m$ denote a 2-CNF Horn formula on the variable set $X = \{x_1, \ldots, x_n\}$ that cannot be further shrunk by the preprocessing as described above. We construct a graph G_Φ from Φ as follows. For each variable x_i that appears in a positive 1-clause, we add a vertex v_i, for each variable x_i that appears in a negative 1-clause, we add a vertex $\overline{v_i}$, and for each variable x_k only appearing in 2-clauses of Φ, we add two vertices v_k and $\overline{v_k}$ and an edge between them. Additionally, we add an edge $\{\overline{v_i}, v_j\}$ for each clause $(x_i \vee \overline{x_j})$ and an edge $\{v_i, v_j\}$ for each clause $(\overline{x_i} \vee \overline{x_j})$. Note that, due to the preprocessing, this construction is well-defined. An example of the construction is shown in Figure 4.6.

We now show that any independent set in G_Φ directly translates into a set of variables in Φ that can be assigned without raising a contradiction. Let $V' \subseteq V_\Phi$ be an independent set in G_Φ. By the construction, at most one of the vertices v_i and $\overline{v_i}$ can be part of V', for every $1 \le i \le n$. Every variable x_i corresponding to a vertex $v_i \in V'$ can be set to the value 1 and every variable x_i corresponding to a vertex $\overline{v_i} \in V'$ can be set to the value 0. Since V' is an independent set, this

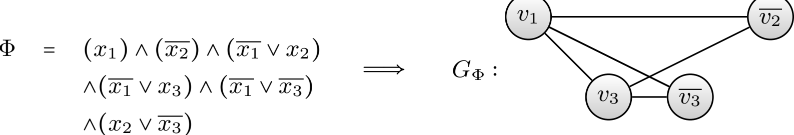


Figure 4.6. An example of the construction in the proof of Theorem 4.21.

leaves at least one endpoint of each edge in G_Φ unassigned, thus the corresponding 2-clause does not cause a contradiction. On the other hand, any partial assignment not causing any conflict in Φ directly translates into an independent set in G_Φ of the same size. Since the graph G_Φ has at most $2n$ vertices, we lose at most a factor of 2 on the approximation ratio. □

Corollary 4.22. *Since* MaxIS *can be approximated with at least a linear function, any polynomial-time $f(n)$-approximation algorithm for* MaxIS *can be used to approximate* MaxAssign-Horn-2-SAT *within $2 \cdot f(n)$ in polynomial time.*

4.4 Maximum Assignment in Exact-2-CNF Formulas

In this section, we deal with the case of E2-CNF formulas, i. e., formulas containing only clauses of types M2, P2, and N2. The approximation hardness of this problem was implicitly shown by Steinová [62], in her proof of the approximation hardness of the general MaxAssign-2-SAT, she constructs formulas consisting of 2-clauses only.

Theorem 4.23 (Steinová, 2012). *There exists an AP-reduction from* MaxIS *on undirected hypergraphs to* MaxAssign-E2-SAT. □

We complement this result by the following upper bound.

Theorem 4.24. MaxAssign-E2-SAT $\leq_{\text{AP}}$ MaxIS.

Proof. To prove AP-reducability of MaxAssign-E2-SAT to MaxIS, we need the following polynomial-time function that transforms an E2-CNF formula Φ with variable set $X = \{x_1, x_2, \ldots, x_n\}$ into a graph instance G_Φ for MaxIS with vertex set $V_\Phi = \{v_i^0, v_i^1 \mid x_i \in X\}$ and edge set $E_\Phi = \{\{v_i^0, v_j^0\} \mid (x_i \vee x_j) \text{ in } \Phi\} \cup \{\{v_i^0, v_j^1\} \mid (x_i \vee \overline{x_j}) \text{ in } \Phi\} \cup \{\{v_i^1, v_j^1\} \mid (\overline{x_i} \vee \overline{x_j}) \text{ in } \Phi\} \cup \{\{v_i^0, v_i^1\} \mid 1 \leq i \leq n\}$. In other words, every variable x_i gives rise to two vertices v_i^0 an v_i^1 representing the assignment 0 or 1. Those two vertices are connected by an edge and there is also an edge for every assignment restriction given by the clauses (see Figure 4.7). Obviously, this transformation can be implemented in polynomial time.

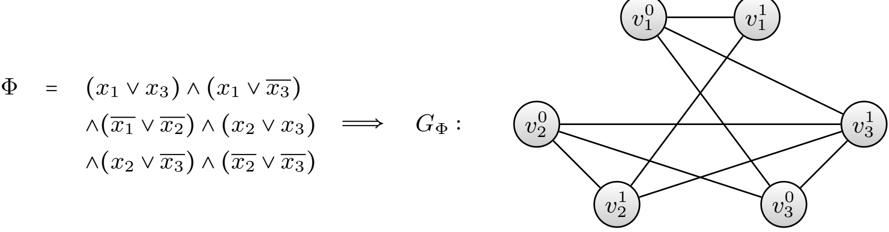


Figure 4.7. An example of the construction in the proof of Theorem 4.24.

We show that every feasible set of variables in Φ with a partial assignment not violating any clause corresponds to an independent set in G_Φ of the same size. Let $X' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \subseteq X$ be a subset of variables and let $\alpha\colon X' \to \{0,1\}$ be a partial assignment such that no clause is evaluated to 0. We show that, for a variable $x_i \in X'$ and an assignment $\alpha(x_i) = b$ for some $b \in \{0,1\}$, the corresponding vertex v_i^b is part of an independent set $I_{X'} \subseteq V_\Phi$ in G_Φ. No two endpoints of edges of type $\{v_i^0, v_i^1\}$ will be part of $I_{X'}$ since a variable can be set either to 1 or to 0, but not to both values. Moreover, no two endpoints of the remaining edges will be both part of $I_{X'}$ since then the corresponding assignment would cause an empty clause. Therefore, $I_{X'}$ is an independent set in G_Φ.
Conversely, let $I \subseteq V_\Phi$ be an independent set in G_Φ. For every $v_i^b \in I$, let $\alpha(x_i) = b$ define the partial assignment for the vertex set $S_I \subseteq X$ in Φ. This assignment does not violate any clause since never both endpoints of an edge will be part of I and therefore, never both literals of a clause will be set to 0.
Hence, we have a one-to-one correspondence between a partial assignment in Φ not causing empty clauses and an independent set in G_Φ of the same size. □

4.5 Conclusion

We have explored the approximability of the minimum splitting problem and the maximum assignment problem in various special cases of 2-CNF formulas, including Horn formulas and E2-CNF formulas. The main open problem is to close the gap between the trivial upper bound and the lower bound for general 2-CNF Horn formulas and for those excluding negative 1-clauses in the splitting case. It would also be interesting to extend the results to other classes of non-Horn 2-CNF formulas besides the E2-CNF formulas.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. 59

[2] Helmut Alt, Norbert Blum, Kurt Mehlhorn, and Markus Paul. Computing a maximum cardinality matching in a bipartite graph in time $o(n^{1.5}\sqrt{m/logn})$. *Information Processing Letters*, 37(4):237–240, 1991. 58

[3] Giorgio Ausiello, Pierluigi Crescenzi, Georgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation*. Springer-Verlag, 1999. 2, 131, 139

[4] Kfir S. Barhum, Hans-Joachim Böckenhauer, Michal Forišek, Heidi Gebauer, Juraj Hromkovič, Sacha Krug, Jasmin Smula, and Björn Steffen. On the power of advice and randomization for the disjoint path allocation problem. In *Proc. of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2014)*, volume 8327 of *Lecture Notes in Computer Science*, pages 89–101. Springer-Verlag, 2014. 22

[5] Dwight R. Bean. Effective coloration. *The Journal of Symbolic Logic*, 41(2):469–480, Association for Symbolic Logic, 1976. 28, 31

[6] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, and Lucia Keller. Online coloring of bipartite graphs with and without advice. In *Proc. of the 18th Annual International Computing and Combinatorics Conference (COCOON 2012)*, volume 7434 of *Lecture Notes in Computer Science*, pages 519–530. Springer-Verlag, 2012. 22, 47

[7] Maria Paola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, Sacha Krug, and Björn Steffen. On the advice complexity of the online L(2,1)-coloring problem on paths and cycles. In *Proc. of the 19th International Conference on Computing and Combinatorics (COCOON 2013)*, volume 7936 of *Lecture Notes in Computer Science*, pages 53–64. Springer-Verlag, 2013. 22

[8] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Sacha Krug, Jasmin Smula, and Andreas Sprock. The string guessing problem as a method to prove lower bounds on the advice complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:162, 2012. 121, 122, 127

[9] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Sacha Krug, Jasmin Smula, and Andreas Sprock. The string guessing problem as a method to prove lower bounds on the advice complexity. In *Proc. of the 19th International Conference on Computing and Combinatorics (COCOON 2013)*, volume 7936 of *Lecture Notes in Computer Science*, pages 493–505. Springer-Verlag, 2013. 22, 121, 122

[10] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královič, and Peter Rossmanith. On the advice complexity of the knapsack problem. In *Proc. of the 10th Latin American Symposium on Theoretical Informatics (LATIN 2012)*, volume 7256 of *Lecture Notes in Computer Science*, pages 61–72. Springer-Verlag, 2012. 22

[11] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, and Richard Královič. On the advice complexity of the k-server problem. In *Proc. of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011)*, volume 6755 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag, 2011. 22

[12] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Královič, Richard Královič, and Tobias Mömke. On the advice complexity of online problems. In *Proc. of the 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer-Verlag, 2009. 2, 20, 22

[13] Ravi Boppana and Magnús Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT Numerical Mathematics*, 32:180–196, 1992. 136

[14] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998. 14, 15, 18, 22

[15] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. Online bin packing with advice. *CoRR*, abs/1212.4016, 2012. 22

[16] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. 1

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158. Association for Computing Machinery, 1971. 8

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. 138

[19] Stefan Dobrev, Rastislav Královič, and Richard Královič. Independent set with advice: The impact of graph knowledge. In *Proc. of the 10th International Workshop on Approximation and Online Algorithms (WAOA 2012)*, volume 7846 of *Lecture Notes in Computer Science*, pages 2–15. Springer-Verlag, 2012. 22

[20] Stefan Dobrev, Rastislav Královič, and Euripides Markou. Online graph exploration with advice. In *Proc. of the 19th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2012)*, volume 7355 of *Lecture Notes in Computer Science*, pages 267–278. Springer-Verlag, 2012. 22

[21] Stefan Dobrev, Rastislav Královič, and Dana Pardubská. How much information about the future is needed? In *Proc. of the 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, volume 4910 of *Lecture Notes in Computer Science*, pages 247–258. Springer-Verlag, 2008. 2, 20

[22] Vladimir Andreevich Dobrushkin. *Methods in Algorithmic Analysis*. Chapman and Hall/CRC Press, 2010. 5

[23] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984. 137

[24] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. 58

[25] Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. In *Proc. of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, volume 5555 of *Lecture Notes in Computer Science*, pages 427–438. Springer-Verlag, 2009. 20, 22

[26] Lene M. Favrholdt and Martin Vatshelle. Online greedy matching from a new perspective. available at http://www.ii.uib.no/~martinv/Papers/OnlineMatching.pdf. 60, 61

[27] Steven R. Finch. *Mathematical Constants (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, 2003. 5

[28] Fedor V. Fomin, Jan Kratochvíl, Daniel Lokshtanov, Federico Mancini, and Jan Arne Telle. On the complexity of reconstructing H-free graphs from their star systems. In *Proc. of the 8th Latin American Symposium on Theoretical Informatics (LATIN 2008)*, volume 4957 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 2008. 105

[29] Michal Forišek, Lucia Keller, and Monika Steinová. Advice complexity of graph coloring in paths, cycles and spider graphs. Unpublished manuscript, submitted. 53, 54

[30] Michal Forišek, Lucia Keller, and Monika Steinová. Advice complexity of online coloring for paths. In *Proc. of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*. 22, 48

[31] Joseph A. Gallian. A dynamic survey of graph labeling. *The Electronic Journal of Combinatorics*, 6, 1998. 51

[32] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, Springer-Verlag, 1931. 1

[33] András Gyárfás and Jenö Lehel. On-line and first fit colorings of graphs. *Journal of Graph Theory*, 12(2):217–227, 1988. 27

[34] David Hilbert. Die Grundlagen der Mathematik. *Abhandlungen aus dem Seminar der Hamburgischen Universität*, 6:65–85, 1927. 1

[35] Johan Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS 1996)*, pages 627–636. IEEE Computer Society, 1996. 136

[36] Juraj Hromkovič. *Algorithmic Adventures: From Knowledge to Magic*. Springer-Verlag, 2009. 1

[37] Juraj Hromkovič. *Algorithmics for Hard Problems. Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics.* Springer-Verlag, 2nd edition, 2004. 5, 9, 14, 139

[38] Juraj Hromkovič, Rastislav Královič, and Richard Královič. Information complexity of online problems. In *Proc. of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010)*, volume 6281 of *Lecture Notes in Computer Science*, pages 24–36. Springer-Verlag, 2010. 2, 20, 21

[39] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1990)*, pages 301–309. Society for Industrial and Applied Mathematics, 1990. 22

[40] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, Springer-Verlag, 1988. 18

[41] Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proc. of the 22nd Annual ACM Symposium on the Theory of Computing (STOC 1990)*, pages 352–358. ACM, 1990. 3, 59, 60

[42] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *Journal of Computer and System Sciences*, 74(3):335–349, 2008. 136

[43] Hal A. Kierstead. Recursive and on-line graph coloring. In Yu. L. Ershov, S.S. Goncharov, A. Nerode, J.B. Remmel, and V.W. Marek, editors, *Handbook of Recursive Mathematics Volume 2: Recursive Algebra, Analysis and Combinatorics*, volume 139 of *Studies in Logic and the Foundations of Mathematics*, pages 1233–1269. Elsevier, 1998. 27

[44] Hal A. Kierstead and William T. Trotter. On-line graph coloring. In Lyle A. McGeoch and Daniel D. Sleator, editors, *On-line Algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 85–92. AMS|DIMACS|ACM, 1992. 27

[45] Dennis Komm. *Advice and Randomization in Online Computation*. PhD thesis, ETH Zurich, 2012. 15, 24, 25, 102

[46] Dennis Komm, Richard Královič, and Tobias Mömke. On the advice complexity of the set cover problem. In *Proc. of the 7th International Computer Science Symposium in Russia (CSR 2012)*, volume 7353 of *Lecture Notes in Computer Science*, pages 241–252. Springer-Verlag, 2012. 22

[47] Dennis Komm and Richard Královič. Advice complexity and barely random algorithms. *Theoretical Informatics and Applications (RAIRO)*, 45(2):249–267, EDP Sciences, 2011. 21, 22

[48] Dennis Komm and Richard Královič. Advice complexity and barely random algorithms. In *Proc. of the 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011)*, volume 6543 of *Lecture Notes in Computer Science*, pages 332–343. Springer-Verlag, 2011. 22

[49] László Lovász, Michael E. Saks, and William T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75(1–3):319–325, 1989. 27, 28, 30

[50] Yishay Mansour and Shai Vardi. A local computation approximation scheme to maximum matching. *CoRR*, abs/1306.5003, 2013. 58

[51] Silvio Micali and Vijay V. Vazirani. An O(sqrt(|v|) |E|) algorithm for finding maximum matching in general graphs. In *Proc. of the 21st Annual Symposium on Foundations of Computer Science (FOCS 1980)*, pages 17–27. IEEE Computer Society, 1980. 58

[52] Burkhard Monien and Ewald Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985. 136

[53] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. In *Proc. of the 45th Symposium on Foundations of Computer Science (FOCS 2004)*, pages 248–255. IEEE Computer Society, 2004. 58

[54] Muriel Niederle, Alvin E. Roth, and Tayfun Sönmez. Matching and market design. In Steven N. Durlauf and Lawrence E. Blume, editors, *The New Palgrave Dictionary of Economics*. Palgrave Macmillan, 2008. 57

[55] Hirotaka Ono. Personal communication. 131

[56] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991. 136

[57] Michael D. Plummer and László L. Lovász. *Matching Theory*. North-Holland Mathematics Studies. Elsevier Science, 1986. 58

[58] Sebastian Seibert, Andreas Sprock, and Walter Unger. Advice complexity of the online coloring problem. In *Proc. of the 8th International Conference on Algorithms and Complexity (CIAC 2013)*, volume 7878 of *Lecture Notes in Computer Science*, pages 345–357. Springer-Verlag, 2013. 22, 27, 93

[59] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996. 1

[60] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Association for Computing Machinery, 1985. 2, 18

[61] Neil J.A. Sloane. Sequence A000073 in The On-Line Encyclopedia of Integer Sequences. published electronically at http://oeis.org/A000073, 2012. 5

[62] Monika Steinová. *Measuring Hardness of Complex Problems: Approximability and Exact Algorithms*. PhD thesis, ETH Zurich, 2012. 131, 132, 133, 136, 140, 142

[63] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, Oxford University Press, 1936. 1

[64] Sundar Vishwanathan. Randomized online graph coloring. *Journal of Algorithms*, 13(4):657–669, 1992. 27

[65] Emo Welzl. Boolean satisfiability – combinatorics and algorithms. Lecture notes, 2004. 8

[66] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001. 6, 79

[67] Andrew C.-C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27(2):207–227, Association for Computing Machinery, 1980. 18