

Graph Coloring als Online-Problem mit bipartiten Graphen und Advice Complexity

Semesterarbeit

Florian Lüthi*

ZHAW, 22. November 2013

*luethifl@students.zhaw.ch

Zusammenfassung

Dies ist die Zusammenfassung.

Inhaltsverzeichnis

1	Einleitung	4
2	Advice Complexity	5
2.1	Online- vs. Offline-Probleme	5
2.2	Competitive Ratio	5
2.3	Das SKIRENTAL-Problem	6
2.4	Das Orakel	8
2.5	Online SIMPLEKNAPSACK mit Advice	10
3	Online Coloring von bipartiten Graphen	12
3.1	Modelle von (bipartiten) Graphen	12
3.2	Zufallsgraphen	14
3.3	Generierung von Zufallsgraphen	14
3.4	Graph Coloring	16
3.5	Online Graph Coloring: First Fit-Algorithmus	17
4	Online Coloring von bipartiten Graphen mit Advice	23
4.1	Online Coloring Algorithmus mit Advice	23
4.2	Obere und untere Schranken für Advices	23
4.3	Experimente	25
4.3.1	Anzahl Advices für selbe Competitive Ratio wie First Fit	25
5	Kommentierung der Implementation des Programms GraphBonanza	31
5.1	Anforderungen	31
5.2	Umfeld, Architektur und verwendete Bibliotheken	31
5.3	Die GraphBuilder-Klasse	31
5.4	Die FirstFitAlgo-Klasse	31
5.5	Die AlgoWithAdvice-Klasse	31
5.6	Die Looper-Klasse	31
5.7	User Interface	32
5.8	Caveats	32
	Literaturverzeichnis	33

1 Einleitung

2 Advice Complexity

2.1 Online- vs. Offline-Probleme

Optimierungsprobleme lassen sich (unter vielerlei anderen Klassifikationsmöglichkeiten) in Online-Probleme und Offline-Probleme mit dazugehörigen Online- bzw. Offline-Algorithmen unterteilen. Der wesentliche Unterschied zwischen diesen beiden Klassen besteht darin, dass ein Offline-Algorithmus seine gesamte Eingabe zur Laufzeit kennt, während ein Online-Algorithmus dies nicht tut. Er bekommt einen Teil der Eingabe und berechnet daraus sofort einen Teil des Resultats. Ein solches Teilresultat kann im Nachhinein nicht mehr geändert werden.

Es liegt nun auf der Hand, dass ein Online-Algorithmus dazu tendiert, während der Berechnung eines Teilresultats Entscheidungen zu treffen, welche sich nach Berechnungen weiterer Teilresultate als suboptimal herausstellen – dies insbesondere darum, weil die während der Berechnung des Teilresultats i zur Verfügung stehenden Informationen über das gesamte Problem nur aus den Teileingaben der Berechnungen 1 bis i stammen können [1].

Eine weitere Folge des obengenannten Unterschieds manifestiert sich darin, dass ein Online-Algorithmus im Allgemeinen offensichtlich keine optimale Lösung findet. Auf der anderen Seite funktioniert ein qualitativer Vergleich zweier Algorithmen dadurch, dass deren asymptotische Laufzeitkomplexität verglichen wird. Da sich die Qualität der von einem Online-Algorithmus und dessen Offline-Pendant produzierten Lösungen grundlegend unterscheidet (während letzterer eine optimale Lösung anstrebt, ist ersterem diese konzeptbedingt verwehrt), muss eine andere Möglichkeit gefunden werden, diese beiden Algorithmen (und deren Lösungen) vergleichen zu können.

2.2 Competitive Ratio

Eine gängige Möglichkeit, dieses Problem zu lösen, ist die von Sleator und Tarjan eingeführte *Competitive Ratio* [15, 1], welche im Grunde genommen das Verhältnis zwischen den Kosten der optimalen Lösung und derer des zu untersuchenden Online-Algorithmus ausdrückt [1]. Da die Qualität der Lösung eines Online-Algorithmus wesentlich von der Beschaffenheit der Eingabe abhängt, wird dabei vom jeweils schlechtestmöglichen Fall ausgegangen [16].

Formal kann die Competitive Ratio folgendermassen definiert werden: Sei $\text{opt}(I)$ die optimale Lösung eines konkreten Online-Problems für eine Sequenz aus Teileingaben I . \mathcal{I}

sei die Menge aller möglicher solcher Sequenzen. Seien ferner \mathcal{S} alle möglichen Lösungen (mit $\text{opt}(I) \in \mathcal{S}$), $A(I) \in \mathcal{S}$ diejenige Lösung, welche der Online-Algorithmus zu diesem Problem liefern kann, und $C : \mathcal{S} \rightarrow \mathbb{R}$ die Kosten für die jeweilige Lösung.

Ein Online-Algorithmus wird nun als *c-kompetitiv* bezeichnet, wenn es Konstanten $c \geq 0$ und α gibt, sodass für sämtliche mögliche Eingaben I

$$C(A(I)) \leq c \cdot C(\text{opt}(I)) + \alpha$$

gilt [1], und die Competitive Ratio wäre ergo

$$c = \max_{I \in \mathcal{I}} \left\{ \frac{C(A(I))}{C(\text{opt}(I))} \right\}.$$

2.3 Das SKIRENTAL-Problem

Die Competitive Ratio lässt sich durch das wohlbekannte SKIRENTAL-Problem illustrieren: Nehmen wir an, wir wollen zum ersten Mal ein Wochenende lang Skifahren gehen. Wir besitzen aber keine Skis. Es stellt sich nun die Frage, ob wir Skis mieten oder kaufen wollen – wobei erschwerend hinzukommt, dass wir momentan noch keine Ahnung haben, wieviele Wochenenden wir danach noch skifahrend verbringen möchten.

Angenommen, der Kauf eines Paares Ski kostet CHF 500, die Miete äquivalenter Skis für ein Wochenende hingegen CHF 50.¹ Es ist nun einfach auszurechnen, dass sich der Kauf der Skis vor dem ersten Wochenende nur lohnen würde, wenn wir an mindestens 10 Wochenenden Skifahren gehen würden.

Da wir das aber nicht wissen, sind wir quasi der Online-Algorithmus, der für jede Teileingabe (sprich für jedes geplante Skiwochenende) sofort die Frage nach Kauf oder Miete beantworten muss. Die Teileingabe für den Algorithmus ist jeweils die Anzahl an vergangenen Skiwochenenden.

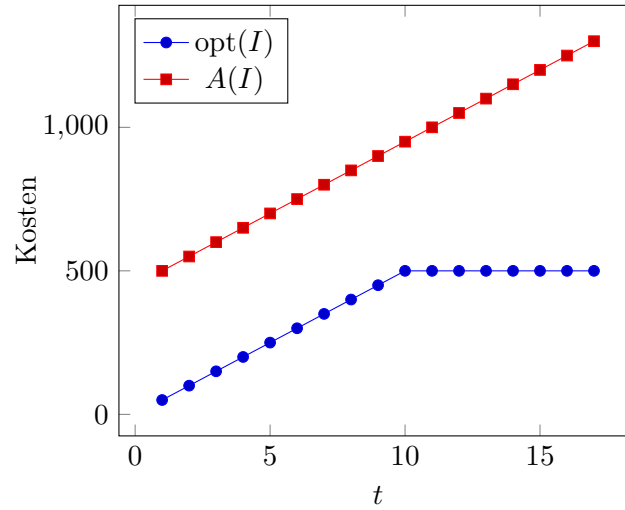
Die oben besprochene Eigenschaft der potenziellen Suboptimalität der Entscheidungen wird beim zehnten solchen Wochenende augenscheinlich, wenn bei den vorangegangenen Wochenenden die Entscheidung jeweils auf Mieten gefallen sein sollte.

Bezeichne t nun dasjenige Wochenende, an welchem wir uns dafür entscheiden, die Skis zu kaufen. Wir schauen uns die folgenden Fälle an:

- Für $t = 1$ gilt, dass $A(I) = 500$, denn wir kaufen ja gleich beim ersten Mal. Die Competitive Ratio c hingegen liegt bei 10, da wir im schlechtesten Fall nur dieses eine Mal Skifahren gehen – es gälte also $\text{opt}(I) = 50$.

¹In der Realität wesentliche Aspekte wie Preisveränderungen über die Zeit, veränderte Qualitätsansprüche und Abnutzung der Skis sollen bei dieser Betrachtung vernachlässigt werden.

Abbildung 2.1: Verlauf von Lösung und Optimum für SkiRental



- Für $t = 2$ gilt, dass $A(I) = 550$, denn wir mieten beim ersten (CHF 50) und kaufen beim zweiten Mal (CHF 500). Die Competitive Ratio c liegt bei 5.5, da wir im schlechtesten Fall nur diese zwei Male Skifahren gehen – optimal wäre zwei Mal mieten, also $\text{opt}(I) = 50$.
- Für $t = 10$ gilt, dass $A(I) = 950$, denn wir mieten die ersten neun Male (CHF 450) und kaufen beim zehnten Mal (CHF 500). Für die optimale Lösung spielt es keine Rolle, ob gemietet oder gekauft wird, also ist $\text{opt}(I) = 500$. Dies ergibt eine Competitive Ratio von $c = 1.9$.
- Für $t = 11$ gilt, dass $A(I) = 1000$, denn wir mieten die ersten zehn Male (CHF 500) und kaufen beim elften Mal (CHF 500). Die Competitive Ratio c liegt bei $c = 2$, da es optimal wäre, gleich beim ersten Mal die Skis zu kaufen (CHF 500).

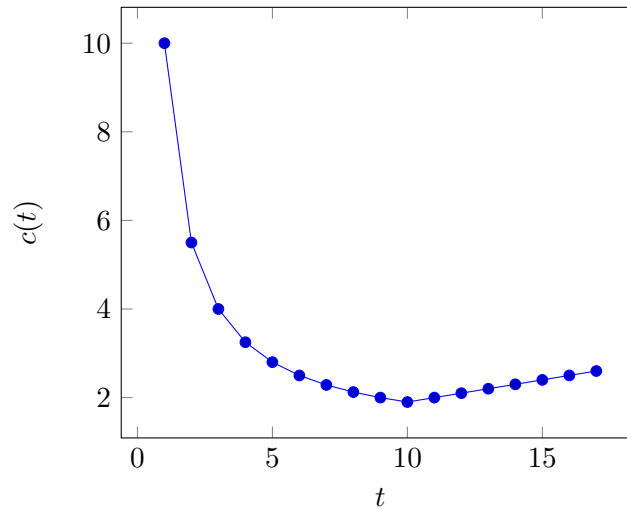
Der Verlauf von $\text{opt}(I)$ und $A(I)$ ist in Abbildung 2.1 skizziert, Abbildung 2.2 zeigt c in Abhängigkeit von t .

Allgemein kann die Competitive Ratio für dieses Problem (mit allgemeinen Kosten b für den Kauf bzw. r für die Miete) beschrieben werden als

$$c(t) = \begin{cases} \frac{b+(t-1) \cdot r}{t \cdot r}, & \text{wenn } r \cdot t \leq b \\ \frac{b+(t-1) \cdot r}{b}, & \text{sonst.} \end{cases}$$

Unsere Instanz des SKIRENTAL-Problems ist also 1.9-kompetitiv; im Allgemeinen ist SKIRENTAL $(2 - \frac{r}{b})$ -kompetitiv [6].

Abbildung 2.2: Verlauf der Competitive Ratio für SkiRental



Zurück auf die Realität bezogen, scheint es eine sinnvolle Strategie bei *Kaufen oder Mieten*-Problemen zu sein, das Objekt zu dem Zeitpunkt zu kaufen, an dem die kumulierten Mietkosten die Kaufkosten aufwiegen oder überschreiten – die Competitive Ratio ist zu diesem Zeitpunkt die beste. Ausserdem kann die Annahme getroffen werden, dass die Wahrscheinlichkeit für eine weitere Benutzung des entsprechenden Objekts um einen mindestens gleich langen Zeitraum wie während der Mietphase (während welcher sich die Kaufkosten amortisieren) relevant ist [16].

2.4 Das Orakel

Es ist augenscheinlich, dass die Lösung $t = 10$ für unsere Instanz des SKIRENTAL-Problems die beste Approximation ist, welche wir erreichen können, solange wir nicht in die Zukunft schauen können. Nichtsdestotrotz ist diese Lösung offensichtlich weit entfernt davon, optimal zu sein – im schlechtesten Fall würden wir 1.9 mal so viel für unsere Skis bezahlen wie eigentlich nötig.

Entsprechend interessant wäre es, eine Aussage darüber machen zu können, wieviel zusätzliche Information der Online-Algorithmus zur Verfügung haben müsste, um das Problem optimal (oder mit einer festgelegten Competitive Ratio [1]) lösen zu können. Während die Competitive Ratio die Kosten darstellt, welche durch die gefundene Lösung zusätzlich zur optimalen Lösung zu gewärtigen wären, würde diese Aussage den Zusatznutzen beschreiben, welcher sich durch den Preis dieser zusätzlichen Kosten erkaufen liesse [6].

Für unser SKIRENTAL-Problem ist die Antwort denkbar trivial: Es ist vollkommen

ausreichend, bei der ersten Entscheidung (sprich bei der Berechnung des ersten Teilergebnisses) zu wissen, ob die Skis gekauft werden oder ob sie für dieses und alle weiteren Wochenenden gemietet werden sollen – mit einem einzigen Bit zusätzlicher Information könnte das Problem also optimal gelöst werden.

SKI-RENTAL ist insofern speziell, als dass dem ganzen Leben (und damit auch der Freizeitplanung) nichtdeterministische Aspekte innewohnen; aus diesem Grunde wird die benötigte Information von keiner Instanz geliefert werden können. Auf der anderen Seite sind aber durchaus Probleme denkbar, welche eigentlich Online-Probleme sind (oder aus praktischen Gründen von Online-Algorithmen gelöst werden), die Qualität der Resultate aber nachhaltig verbessert werden könnten, wenn zusätzliche Offline-Informationen aus der gesamten Eingabe herangezogen werden könnten. Deren Beschaffung könnte mit derart hohen Kosten verbunden sein, dass die Bestrebung wäre, einen möglichst guten Kompromiss zwischen diesen Kosten und der verbesserten Competitive Ratio zu erreichen [6].

Interessant wäre es darum, ein Modell zur Verfügung zu haben, mit welchem sich Probleme nach Anzahl und Art der zusätzlich benötigten Informationen klassifizieren liessen. [6] schlägt ein solches Modell vor [1] – die sogenannte *Advice Complexity*. Das Modell nimmt die Existenz eines *Orakels* \mathcal{O} an, welchem das Problem in seiner Gesamtheit bekannt ist und welches über unlimitierte Rechenleistung verfügt [6, 1]. Das Orakel stellt dem Algorithmus \mathcal{A} bitweise Informationen zur Verfügung [6]. Das Mass der *Advice Complexity* bezeichnet demzufolge die Menge der zwischen dem Algorithmus und dem Orakel ausgetauschten Informationen. Da diese Menge an Informationen generell von der Länge der Eingabe abhängt, bezeichnet die Advice Complexity von \mathcal{A} für Eingaben der Länge n das Maximum an Advice Complexity für alle Abfragen für Totaleingaben mit maximaler Länge von n , normalisiert auf n [1]:

$$B_{(\mathcal{A}, \mathcal{O})} = \max_{I \in \mathcal{I}^n} \frac{B_{(\mathcal{A}, \mathcal{O})}(I)}{n},$$

während die Advice Complexity des Problems \mathcal{P} definiert ist als diejenige Advice Complexity des besten Algorithmus [3].

[6] schlägt vor, die Kommunikation zwischen dem Orakel und dem Algorithmus in wahlweise zwei verschiedenen Modi ablaufen zu lassen:

- Im *Answerer-Modus* entscheidet der Algorithmus selbständig, ob und für welche Eingaben er das Orakel um Rat fragen muss. Für jede Anfrage schickt das Orakel einen Advice-String zurück, welcher aber nicht leer sein kann [6] (denn dies wäre ja kein sinnvoller Rat).

Bezogen auf SKI-RENTAL würde der Algorithmus das Orakel vor dem ersten Wochenende fragen, ob er die Skis kaufen oder mieten soll, und das Orakel würde diese Frage mit *Kaufen* oder *Mieten* beantworten. Danach wäre keine weitere Kommunikation zwischen Orakel und Algorithmus notwendig.

- Im *Helper*-Modus schickt das Orakel dem Algorithmus sporadisch und spontan *Advices*. Wir können uns das so vorstellen, dass der Algorithmus mehrere voneinander unabhängige Verhaltensmuster für die Berechnung der Teilausgabe aufweist, welche vom Orakel entsprechend aufgerufen werden können. Bei jedem notwendigen Wechsel des Verhaltensmusters würde das Orakel erneut einen *Advice* senden.

Bezogen auf *SKI RENTAL* würde das Orakel den Algorithmus vor dem ersten Wochenende entweder in den *Kaufen*- oder *Mieten*-Modus versetzen. Der *Kaufen*-Modus kauft die Skis für die erste Teileingabe und tut nichts mehr für alle folgenden Teileingaben, während der *Mieten*-Modus für jede Teileingabe die Skis mietet.

Wir beobachten, dass die Länge der vom Orakel gegebenen *Advice*-Strings ebenfalls Informationen beinhalten (beispielsweise zeigt ein leerer String im *Helper*-Modus dem Algorithmus an, das momentane Verhaltensmuster beizubehalten). Dies ist insofern ein Problem, als dass es das Modell einerseits komplexer als notwendig macht. Andererseits wird der Vergleich von *Advice Complexity* mit anderen Komplexitätsmassen dadurch erschwert [1].

[2] entwickelt dieses Modell der *Advice Complexity* dahingehend weiter, als dass die Existenz eines *Advice Tapes* angenommen wird, wohin das Orakel alle *Advices* sequentiell schreibt, und zwar vor der Berechnung der ersten Teileingabe durch den Algorithmus. Dadurch wird das Modell direkt vergleichbar mit demjenigen von randomisierten Online-Algorithmen [1]; es kann konstatiert werden, dass für jeden randomisierten Online-Algorithmus \mathcal{R} , welcher b Bits von seinem Random-Tape liest und eine Competitive Ratio von r erreicht, ein Algorithmus mit *Advice* \mathcal{A} und Orakel \mathcal{O} existiert, welcher b *Advice*-Bits benötigt und dessen Competitive Ratio ebenfalls r ist [2]. Dies ist einfach durch die Konstruktion von \mathcal{O} dahingehend zu erreichen, dass \mathcal{O} dieselben Bits auf das *Advice Tape* schreibt, wie auf dem Random Tape für den randomisierten Algorithmus stehen würden.

2.5 Online SIMPLEKNAPSACK mit Advice

Um die teilweise überraschenden Fähigkeiten von Algorithmen mit *Advice* zu demonstrieren, wollen wir uns das SIMPLEKNAPSACK-Problem anschauen: Wir nehmen die Existenz eines Rucksacks an. Dieser Rucksack hat eine maximale Kapazität von 1. Des weiteren nehmen wir eine Menge von Gegenständen $S = \{w_1, \dots, w_n\}$ (deren w_i die entsprechenden Grössen ausdrücken, mit $0 < w_i \leq 1$) an. Das Ziel ist, den Rucksack möglichst gut (aber nur bis zur Kapazitätsgrenze) zu füllen.

Erschwerend kommt noch hinzu, dass dieses Problem als Online-Problem formuliert wird – der Algorithmus bekommt also einen Gegenstand präsentiert und muss sofort entscheiden, ob er diesen im Rucksack verstauen will oder nicht. Das nachträgliche Entfernen von bereits verstauten Gegenständen ist nicht erlaubt [5].

Allgemein existiert kein kompetitiver deterministischer Online-Algorithmus [13]; die einzige Möglichkeit besteht im Grunde genommen darin, sämtliche präsentierten Ge-

gegenstände der Reihe nach einzupacken, solange sie noch in den Rucksack passen. Ein möglicher Adversary wäre nun die Sequenz $\langle \varepsilon, 1 \rangle$, wobei ε beliebig nahe an 0 gewählt werden könnte; dies führt dazu, dass der Algorithmus ε in den Rucksack packt und dann aufhören muss, weil $\varepsilon + 1$ die Kapazitätsgrenze überschreiten würde.

Die Competitive Ratio dieses Algorithmus wäre entsprechend

$$r = \lim_{\varepsilon \rightarrow 0} \frac{1}{\varepsilon} = \infty.$$

Wenn wir nun ein Orakel dazudenken, welches dem Algorithmus bei der ersten Entscheidung mitteilt, ob in der Sequenz ein Gegenstand w_i vorkommt, für welchen $w_i \geq 0.5$ gilt, könnte der Algorithmus folgende Fallunterscheidung machen:

- a) Falls ein solcher Gegenstand vorkommt, würde der Algorithmus solange alle Gegenstände verwerfen, bis ein Gegenstand mit $w_i \geq 0.5$ zur Debatte steht. Diesen würde er wie auch alle nachfolgenden in den Rucksack packen. Da er zumindest ersteren gepackt hat, hätte der Rucksack mindestens eine Gesamtfüllung von 0.5.
- b) Falls hingegen kein solcher Gegenstand vorkommt, würde der Algorithmus von Anfang an alle Gegenstände in den Rucksack packen. Der Rucksack wäre damit ebenfalls zu mindestens zur Hälfte gefüllt, da es keinen Gegenstand ≥ 0.5 geben kann, der nicht in den Rucksack passen würde (ansonsten Fall a) zur Anwendung gelangt wäre).

Daraus folgt, dass die Competitive Ratio des Algorithmus mit einem einzigen Advice Bit

$$r = \frac{1}{0.5} = 2$$

ist [5].

Spannenderweise wird die Competitive Ratio für $b \in \mathcal{O}(1)$ nicht wesentlich verbessert. Der Grund dafür ist im Prinzip, dass eine konstante Anzahl an Advice Bits nicht ausreichen kann, um genügend Informationen darüber transportieren zu können, welche Gegenstände ausgewählt werden sollen. Für $b \in \mathcal{O}(\log(n))$ gilt dies hingegen – die Competitive Ratio reduziert sich dann auf $r = 1 + \varepsilon$ [5].

3 Online Coloring von bipartiten Graphen

3.1 Modelle von (bipartiten) Graphen

Im weiteren Verlauf dieser Arbeit wollen wir uns mit einer speziellen Variante des Graphenfärbeproblems beschäftigen. Damit wir dies tun können, wollen wir einige Begriffe definieren:

Definition 1 (Graph). Ein (ungerichteter, ungewichteter) Graph ist ein Tupel

$$G = (V, E),$$

wobei $V = \{v_1, v_2, \dots\}$ dessen Knoten (*vertices*) bezeichnet und $E = \{e_1, e_2\}$ dessen Kanten (*edges*). Jedes e ist eine 2-elementige Teilmenge von V . [4]

Abbildung 3.1 illustriert einen Graphen mit

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

und

$$E = \{\{v_1, v_2\}, \{v_3, v_6\}, \{v_2, v_5\}, \{v_3, v_4\}\}.$$

Um einen gerichteten Graphen zu erhalten, würden wir die Definition von E so abändern, dass E nicht eine Menge von Teilmengen von V wäre, sondern eine Menge von Tupeln, so dass

$$E \subset V \times V;$$

gerichtete Graphen sind für diese Arbeit aber nicht weiter interessant. Dasselbe gilt für gewichtete Graphen; bei diesen wären entweder die Knoten gewichtet (das heisst, wir bräuchten eine zusätzliche Abbildung $w_v : V \rightarrow \mathbb{R}$) oder die Kanten gewichtet (durch eine entsprechende Abbildung $w_e : E \rightarrow \mathbb{R}$).

Definition 2 (Bipartiter Graph). Wenn alle Knoten eines Graphen G so in zwei disjunkte Mengen S_1 und S_2 (Partitionsklassen oder *shores*) aufgeteilt werden können, dass keine Kanten von G zwischen Knoten aus S_1 und solchen aus S_2 existieren, sprechen wir von einem bipartiten Graphen.

Abbildung 3.2 illustriert einen bipartiten Graphen mit

$$V = S_1 \cup S_2 = \{v_1, v_2, v_5, v_7, v_8\} \cup \{v_3, v_4, v_6, v_9\}$$

und

$$E = \{\{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_9\}, \{v_4, v_8\}, \{v_5, v_6\}, \{v_6, v_8\}, \{v_7, v_9\}\}.$$

Abbildung 3.1: Ein Graph

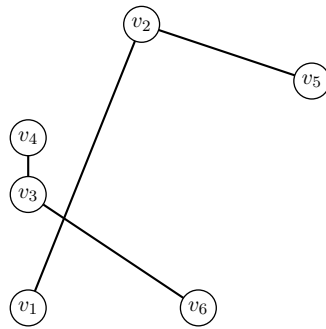


Abbildung 3.2: Ein bipartiter Graph

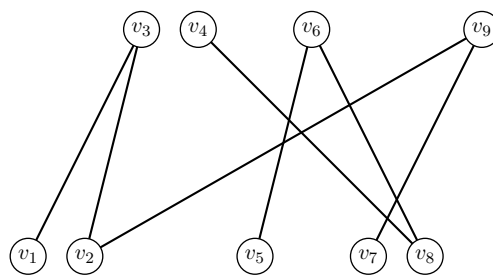
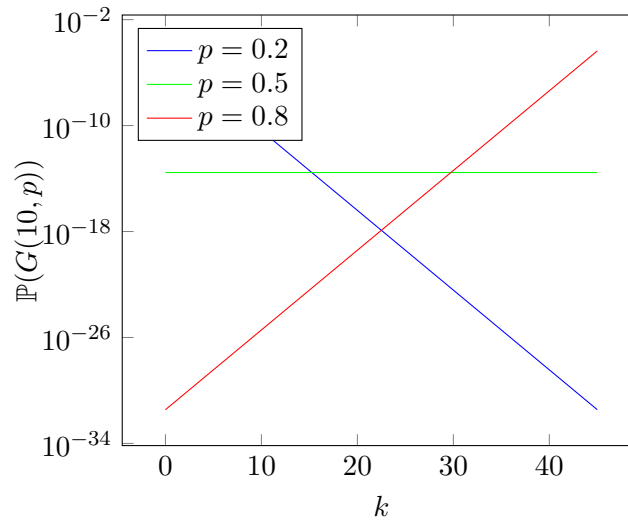


Abbildung 3.3: Verteilung der Auftretenswahrscheinlichkeit eines Zufallsgraphen



3.2 Zufallsgraphen

Wenn wir Eigenschaften von vielen Graphen betrachten wollen, sind Zufallsgraphen ein hilfreicher Denkanlass. Das grundlegende Prinzip ist, eine feste Menge von Knoten V anzunehmen und dann aus allen möglichen Kombinationen von Kanten (sprich aus der Potenzmenge $\mathcal{P}(E)$) mit einer definierten Wahrscheinlichkeit auszuwählen.

Das verbreitete Modell der Erdős-Rényi-Graphen definiert eine Klasse von Graphen $G(n, p)$ mit einer Anzahl Knoten n und einer gleichverteilten Wahrscheinlichkeit p , dass eine Kante zwischen zwei Knoten besteht [7].

Da ein solcher Graph maximal $|\mathcal{P}(E)| = \binom{n}{2}$ Kanten haben kann, beträgt die Wahrscheinlichkeit, dass darin ein Graph mit genau k Kanten auftritt, gegeben als

$$\mathbb{P}(G) = p^k (1 - p)^{\binom{n}{2} - k}.$$

Es ist natürlich möglich, dass einzelne Knoten eines solchen Graphen gar keine Kanten haben, sprich das zusammenhängende Graph-Gebilde nicht alle n Knoten beinhaltet. Diese Knoten werden aber trotzdem zum Graphen dazugezählt. [7]

Abbildung 3.3 zeigt die Auftretenswahrscheinlichkeit eines Graphen in $G(10, p)$ abhängig von der Anzahl Kanten für $p = 0.2, 0.5, 0.8$.

3.3 Generierung von Zufallsgraphen

Zufallsgraphen zu generieren ist im Grunde genommen ziemlich einfach, wie Algorithmus 1 zeigt [14]. Das Problem ist allerdings ebenfalls offensichtlich: Die Komplexitäts-

klasse dieses Algorithmus ist $\mathcal{O}(n^2)$.

Algorithmus 1 Ein naiver Zufallsgraphen-Generator

Eingabe:

Anzahl Knoten n ,
Kantenwahrscheinlichkeit p .

Ausgabe:

Zufallsgraph $G = (V, E)$.

```
for  $i \leftarrow 1, n$  do
  for  $j \leftarrow i + 1, n$  do
     $\theta \leftarrow$  gleichverteilte Zufallszahl
    if  $\theta < p$  then
       $E \leftarrow E \cup \{(v_i, v_j)\}$ 
    end if
  end for
end for
```

Um diese technische Suboptimalität zu lösen, verfolgen Nobari, Lu, Karras und Bresnan in [14] folgende Idee:

1. Jede Kante (v_i, v_j) (und darum auch $\{v_i, v_j\}$) kann als e_k ausgedrückt werden durch die Beziehungen

$$i = \left\lfloor \frac{k}{v} \right\rfloor$$

und

$$j = k \bmod v.$$

2. Damit kann Algorithmus 1 $\in \mathcal{O}(n^2)$ zu einem Algorithmus mit linearer Laufzeitkomplexität $\mathcal{O}(|E|)$ umgeschrieben werden (siehe Algorithmus 2). Da aber $|E| = \binom{n}{2}$ gilt, nützt uns das momentan noch nicht viel.
3. Der eigentliche Trick folgt der Erkenntnis, dass die Wahrscheinlichkeitsverteilung einer Folge von gleichverteilten Zufallsentscheidungen, die zum selben Resultat führen, im Grunde einer geometrischen Verteilung entspricht. Dadurch können wir die Wahrscheinlichkeit, dass die nächsten s Kanten des Graphen mit einer jeweiligen Wahrscheinlichkeit von $1 - p$ ausgelassen werden, ausdrücken als

$$f(s) = p \cdot (1 - p)^s,$$

und die kumulative Verteilungsfunktion wäre ([14])

$$F(s) = \sum_{i=0}^s f(i) = 1 - (1 - p)^{s+1}.$$

Algorithmus 2 Ein umgeformter naiver Zufallsgraphen-Generator

Eingabe:

Kantenwahrscheinlichkeit p .

Ausgabe:

Zufallsgraph $G = (V, E)$.

```
for  $i \leftarrow 1, |E|$  do
   $\theta \leftarrow$  gleichverteilte Zufallszahl
  if  $\theta < p$  then
     $E \leftarrow E \cup \{e_i\}$ 
  end if
end for
```

Dies bedeutet, dass der Algorithmus nach jedem Setzen einer Kante eine einzige neue Zufallszahl α generiert, mit welcher er bestimmen kann, wieviele Kanten s ausgelassen werden können (nämlich die kleinste ganze Zahl, für welche

$$F(s) \geq \alpha$$

gilt). Durch ein paar Takte Algebra können wir nach s auflösen und bekommen eine explizite Funktion ([14]):

$$s(\alpha) = \max(0, \lceil \log_{1-p} \alpha \rceil - 1)$$

Damit können wir unseren Algorithmus optimieren (siehe Algorithmus 3). Es liegt nun auf der Hand, dass dieser Algorithmus desto schneller ist, je kleiner p gewählt wird. [14] versucht darum, der durch die Berechnung des Logarithmus gestiegenen mathematischen Komplexität Herr zu werden, indem eine Menge an Logarithmus-Werten vorberechnet und wiederverwendet werden. Ausserdem werden noch Optimierungen vorgenommen, um den Algorithmus auf Grafik-Prozessoren laufen zu lassen. Dies ist für uns allerdings nicht weiter spannend, und darum wollen wir es bei der einfachen Optimierung wie eben gesehen belassen.

3.4 Graph Coloring

Definition 3 (Knotenfärbung). Für einen ungerichteten Graphen $G = (V, E)$ ist die Abbildung

$$f : V \rightarrow C \subset \mathbb{N}$$

eine Knotenfärbung, wenn keinen zwei Knoten, die durch eine Kante verbunden sind, dieselbe Farbe ($c \in C$) zugeordnet wird.

G heisst k -knotenfärbbar, wenn die maximale Anzahl verwendeter Farben k beträgt – das heisst, wenn für alle $v \in V$ gilt, dass

$$f(v) < k.$$

Algorithmus 3 Ein Zufallsgraphen-Generator mit Edge Skipping

Eingabe:

Kantenwahrscheinlichkeit p .

Ausgabe:

Zufallsgraph $G = (V, E)$.

```
 $i \leftarrow -1$ 
while  $i < |E|$  do
   $\alpha \leftarrow$  gleichverteilte Zufallszahl
   $s \leftarrow \max(0, \lceil \log_{1-p} \alpha \rceil - 1)$ 
   $i \leftarrow i + s + 1$ 
   $E \leftarrow E \cup \{e_i\}$ 
end while
Entferne die letzte Kante
```

Die kleinstmögliche solche Zahl k wird Chromatische Zahl $\chi(G)$ geheissen.

Das Auffinden einer gültigen Färbung für einen Graphen wird als Graphenfärbeproblem bezeichnet. Das dazugehörige Entscheidungsproblem, ob ein Graph k -färbbar ist, ist eines von Karp's 21 \mathcal{NP} -vollständigen Problemen [10]. Praktische Anwendung finden Färbeprobleme vor allem als mathematische Formulierung von Scheduling-Problemen aller Art, beispielsweise für Stundenpläne. Ein anderes wohlstudiertes Problem ist die Frage, wie viele Farben notwendig sind, um auf einer beliebigen Landkarte jedes Land so einzufärben, dass nie zwei benachbarte Länder dieselbe Farbe haben – mit dem lange vermuteten, aber doch überraschenden Ergebnis, dass 4 Farben ausreichend sind [8].

Ausserdem liegt auf der Hand, dass ein bipartiter Graph stets 2-färbbar ist.

3.5 Online Graph Coloring: First Fit-Algorithmus

Wir wollen das Graphenfärbeproblem nun als Online-Problem betrachten. Dafür stellen wir uns vor, dass der zu färbende Graph nicht komplett vorliegt, sondern jeder Knoten einzeln präsentiert wird. Jeder Knoten muss im Moment seiner Präsentation eingefärbt werden. Alle Kanten zwischen bereits aufgedeckten Knoten sind bekannt, alle anderen jedoch nicht. [12]

Offensichtlich kann ein solches Problem nicht per se optimal gelöst werden, da Informationen über das gesamte Problem fehlen. Abbildung 3.4 veranschaulicht den simplen First Fit-Algorithmus 4, der für einen beispielhaften (aber doch unspektakulären) bipartiten Graphen folgendermassen scheitert:

1. Der Knoten v_0 wird präsentiert, und bekommt die Farbe Blau.
2. Der Knoten v_1 wird präsentiert, und bekommt ebenfalls die Farbe Blau, da noch keine Kante bekannt ist.

3. Der Knoten v_2 wird präsentiert, und mit ihm die Kante $\{v_0, v_2\}$. v_2 bekommt die Farbe Rot, da v_0 bereits die Farbe Blau bekommen hat.
4. Der Knoten v_3 wird präsentiert, und mit ihm die Kanten $\{v_0, v_3\}$ und $\{v_1, v_3\}$. v_3 bekommt die Farbe Grün, da v_1 bereits die Farbe Blau und v_2 bereits die Farbe Rot bekommen hat.

Algorithmus 4 First Fit-Algorithmus für Online Graph Coloring

Eingabe:

Der präsentierte Knoten v ,
Der bereits bekannte Graph $G' = (V', E')$,
Die bereits bekannte Färbung $f : V' \rightarrow C$

Ausgabe:

Die Färbung für den präsentierten Knoten $f(v) \in C$

```

 $f(v) \leftarrow -1$ 
 $c_{\max} \leftarrow 0$ 
for all  $e \in E'$  do
     $\{w\} \leftarrow e \cap v$ 
     $c_{\max} \leftarrow \max\{c_{\max}, f(w)\}$ 
end for

for  $c \leftarrow 0, c_{\max} - 1$  do
    if  $f(v) = -1$  then
         $\text{ok} \leftarrow \text{true}$ 
        for all  $e \in E'$  do
             $\{w\} \leftarrow e \cap v$ 
             $\text{ok} \leftarrow \text{ok} \wedge f(w) = c$ 
        end for
        if  $\text{ok}$  then
             $f(v) \leftarrow c$ 
        end if
    end if
end for

if  $f(v) = -1$  then
     $f(v) \leftarrow c_{\max} + 1$ 
end if

```

Der kapitale Fehler passierte bereits beim Färben von v_1 . Es gab anhand der bereits vorhandenen Informationen keinen Grund, nicht anzunehmen, dass v_1 Blau zugeordnet bekommen sollte. Wenn allerdings bekannt gewesen wäre, dass v_1 eine Kante zu v_3 , dieser eine Kante zu v_2 und dieser wiederum eine Kante zu v_0 haben wird, hätte dies

Abbildung 3.4: Nicht optimal färbbarer bipartiter Graph

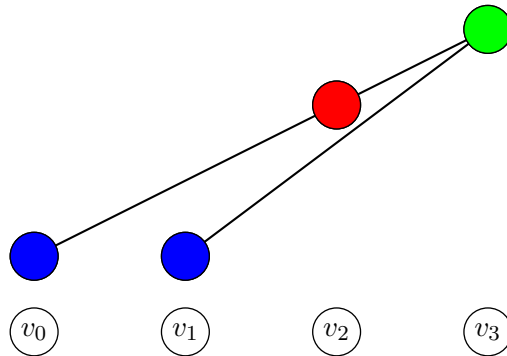
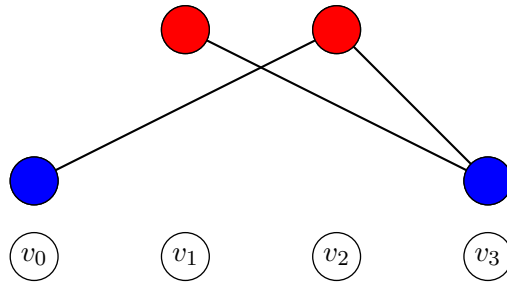


Abbildung 3.5: Optimal gefärbter bipartiter Graph



ausgereicht, um zu erkennen, dass v_1 in der gegenteiligen Shore von v_0 hätte platziert werden müssen, wie in Abbildung 3.5 illustriert wird.

Das Online Coloring-Problem hat keine konstante Competitive Ratio ([3]), wie in [9] gezeigt wird. Für den Nachvollzug wollen wir uns vergegenwärtigen, dass jeder Baum ein bipartiter Graph ist, weil alle Knoten einer Ebene nur mit Knoten der jeweiligen nächsten oder vorhergehenden Ebene verknüpft sein können, wie in Abbildung 3.6 skizziert.

Wir überlegen uns nun eine rekursive Folge von Bäumen:

- Der Baum T_1 besitzt nur einen Knoten.
- Der Baum T_n besteht aus der Vereinigung von exakten Kopien aller Bäume T_1, \dots, T_{n-1} sowie einem zusätzlichen Knoten, der eine Kante zu den jeweiligen Wurzelknoten all dieser Bäume hat. Abbildung 3.7 verdeutlicht dies.

Wenn wir nun die Knoten dieser Bäume in einer geeigneten Reihenfolge präsentieren, maximieren wir die Menge an verdeckter Information, welche jeden Online-Algorithmus scheitern lassen wird. Für den besprochenen First Fit-Algorithmus 4 gehen wir beispielsweise folgendermassen vor:

Abbildung 3.6: Ein Baum

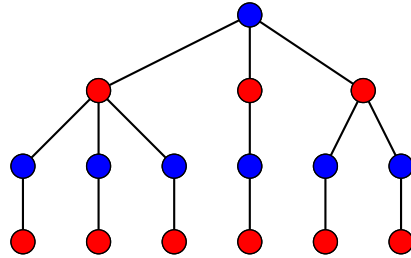
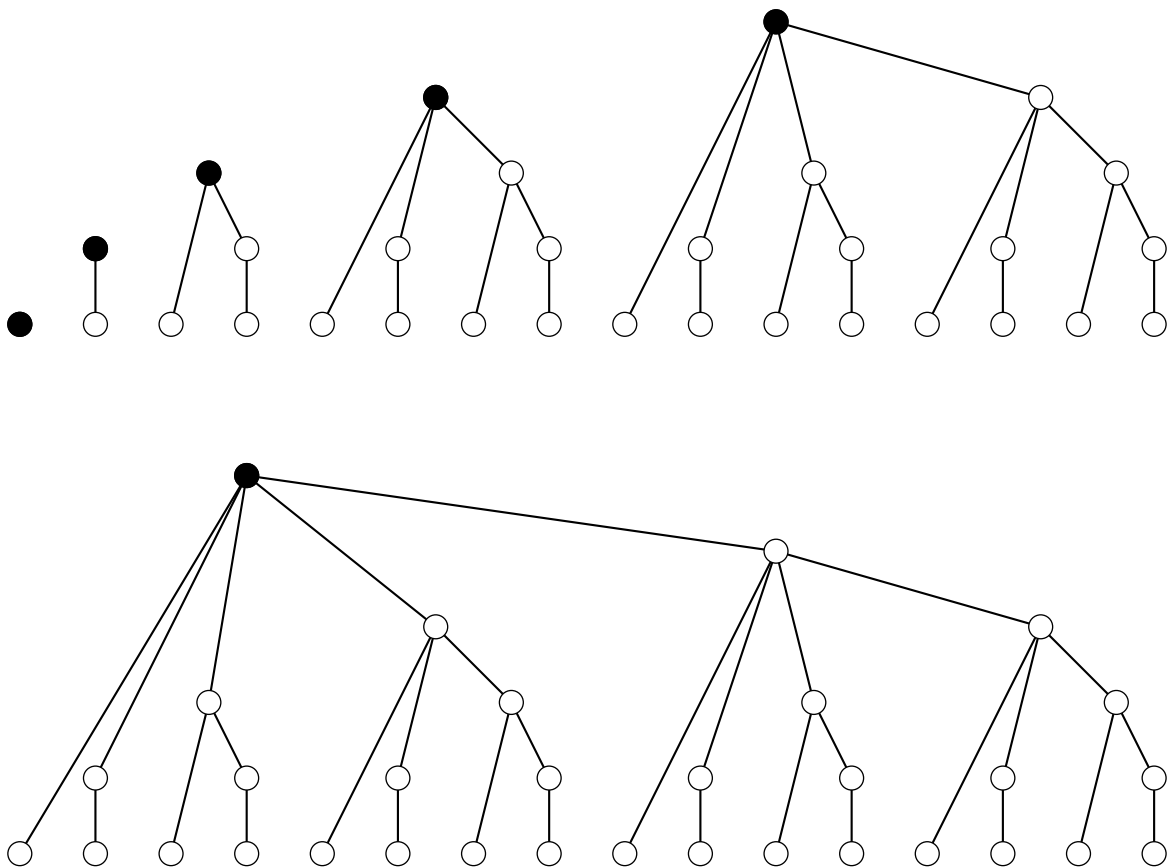


Abbildung 3.7: Die Folge T_1, \dots, T_6



1. Wir präsentieren zuerst alle Knoten, welche durch rekursives Kopieren von T_1 entstanden sind – diese bilden auch gleichzeitig alle Blattknoten aller Bäume. Da diese Knoten natürlich keine gemeinsamen Kanten haben, wird der Algorithmus allen dieselbe Farbe zuweisen – beispielsweise Blau.
2. Nun präsentieren wir alle Knoten, welche durch rekursives Kopieren des für T_2 hinzugefügten Knotens entstanden sind. (Wenn wir uns die im vorhergehenden Schritt präsentierten Knoten aus den Bäumen wegdenken würden, würden die T_2 -originären Knoten die neue Menge an Blattknoten bilden.) Der Algorithmus kann diese Knoten nicht blau färben, da sie alle eine Kante zu schon präsentierten Knoten haben – ergo bekommen sie die Farbe Rot.
3. Den zweiten Schritt wiederholen wir nun für die in T_3, T_4, \dots hinzugefügten und in jeden weiteren Baum kopierten Knoten. Schon bei T_3 stellen wir fest, dass der dazumal hinzugefügte Knoten nur eine dritte Farbe zugeordnet bekommen kann, da er Kanten zu je einem der in den beiden vorangegangenen Schritten präsentierten Knoten hat.

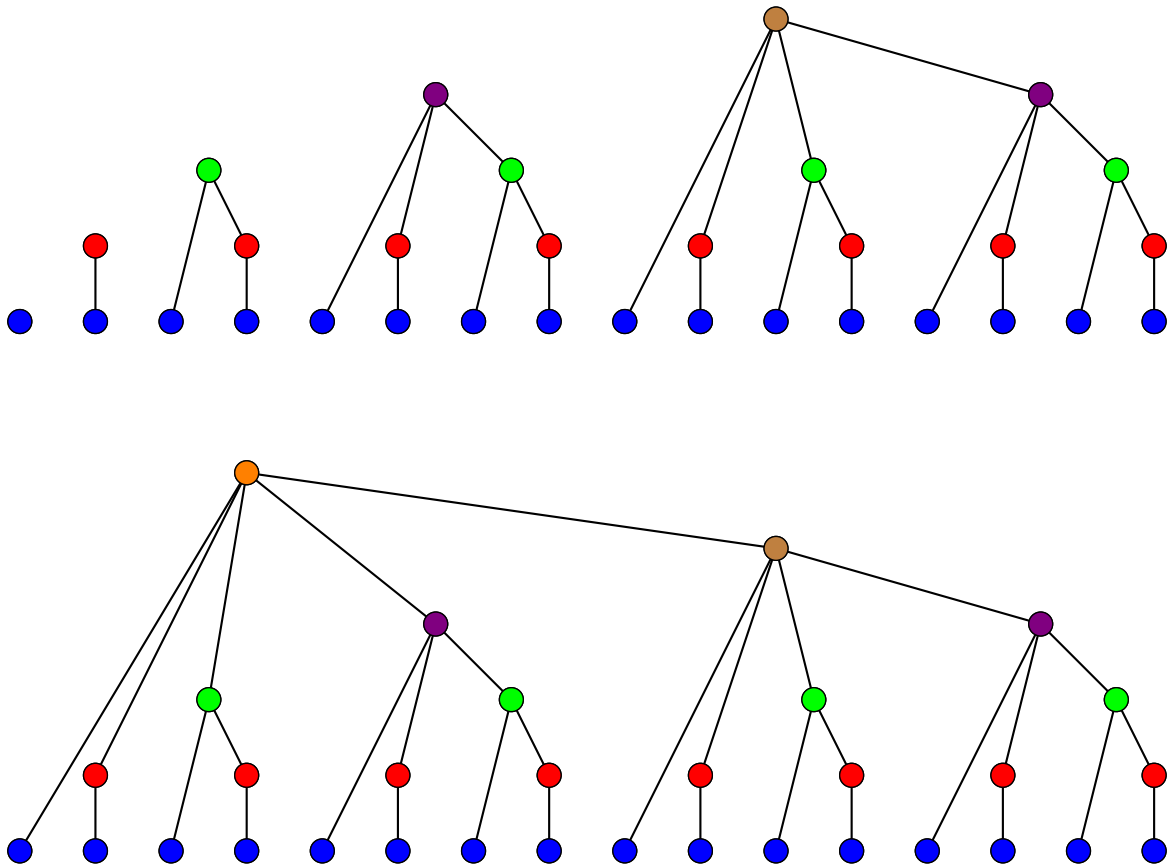
Die Abbildung 3.8 veranschaulicht das Resultat. Es ist nun offensichtlich, dass der Algorithmus für jeden Baum T_n genau n Farben benötigt. Da jeder dieser Bäume 2^{n-1} Knoten hat, und es unendlich viele solcher Bäume gibt (das Spiel lässt sich bei genügend Platz und Zeit endlos weitertreiben), konstatieren wir eine nicht-konstante Competitive Ratio von

$$r = \frac{\log_2 k + 1}{2}$$

in Abhängigkeit der Anzahl Knoten k . Gleichzeitig ist darum $\log_2 k + 1$ eine untere Schranke für die minimale Anzahl Farben von Online-Algorithmen auf bipartiten Graphen – allerdings eine zu optimistische, denn [3] erhöht diese auf

$$1.13747 \cdot \log_2 k.$$

Abbildung 3.8: First Fit auf T_1, \dots, T_6



4 Online Coloring von bipartiten Graphen mit Advice

Im vorhergehenden Kapitel haben wir festgestellt, dass Online Coloring auf bipartiten Graphen nicht kompetitiv ist. Wir wollen nun betrachten, wie sich der Einsatz eines Orakels im Sinne der Advice Complexity auf die Kompetitivität auswirken würde – und hoffen auf ähnlich erstaunliche Ergebnisse wie im Falle von SIMPLEKNAPSACK.

4.1 Online Coloring Algorithmus mit Advice

Das Hauptproblem von Online-Färb-Algorithmus scheint zu sein, dass es Situationen während der Präsentation der Knoten geben kann, welche zu gleichen Färbungen von Knoten auf beiden Shores führen können, wie wir durch Betrachten der Abbildungen 3.4 und 3.5 rekapitulieren können. Dies ist einzig der fehlenden Information über die Zukunft geschuldet. Ein allenfalls eingesetztes Orakel sollte also in genau diesem Aspekt helfen können. Die benötigte Unterstützung beschränkt sich aber auf ein schmales Band: um eine konstante Competitive Ratio $\frac{k}{2}$ zu erreichen (sprich: k Farben zuzulassen), ist einzig wichtig, die Farben k und $k - 1$ nicht durcheinander zu bringen [3].

Dies führt zum Algorithmus 5, welcher wie der First Fit-Algorithmus immer die niedrigstmögliche Farbe zuordnet. Falls aber für einen Knoten die einzigen beiden möglichen Farben $k - 1$ und k sind, fragt er das Orakel, wie der Knoten gefärbt werden soll. Damit wird sichergestellt, dass alle Knoten mit der Farbe $k - 1$ auf der einen Seite und alle Knoten mit der Farbe k auf der anderen Seite des Graphen sind.¹

4.2 Obere und untere Schranken für Advices

Bezüglich den Schranken für Advices lassen sich folgende Aussagen machen:

- Um jeden bipartiten Graphen mit n Knoten optimal färben zu können, sind höchstens $n - 2$ Advices nötig. Die Begründung liegt darin, dass ein Algorithmus konstruiert werden kann, der folgendermassen funktioniert:

¹Es mag bemerkt werden, dass die Ausgestaltung des Algorithmus trotz dessen enger Verwandtschaft zum First Fit-Algorithmus völlig anders ist. Der Grund liegt darin, dass durch die Eingabe von k die Menge an möglichen Farben beschränkt ist; dadurch kann darüber iteriert werden, was die Auswahl der niedrigstmöglichen Farbe vereinfacht.

Algorithmus 5 Online Graph Coloring-Algorithmus mit Advice

Eingabe:

Der präsentierte Knoten v ,
Der bereits bekannte Graph $G' = (V', E')$,
Die bereits bekannte Färbung $f : V' \rightarrow C$,
Die maximale Anzahl erlaubter Farben k ,
Das Orakel $\mathcal{O} : V \rightarrow \{0, 1\}$

Ausgabe:

Die Färbung für den präsentierten Knoten $f(v) \in C$

```
 $C' \leftarrow \{1, \dots, k\}$ 
for all  $c \in C'$  do
  for all  $e \in E'$  do
     $\{w\} \leftarrow e \cap v$ 
    if  $f(w) = c$  then
       $C' \leftarrow C' - \{f(w)\}$ 
    end if
  end for
end for

if  $C' \neq \{k-1, k\}$  then
   $f(v) \leftarrow \min C'$ 
else
   $f(v) \leftarrow k-1 + \mathcal{O}(v)$ 
end if
```

1. Der erste Knoten bekommt die Farbe 1.
 2. Bei der Präsentation des zweiten Knotens wird das Orakel gefragt, ob jedem bei der Präsentation isolierten Knoten ebenfalls die Farbe 1 zugeordnet werden soll. Nicht-isolierte Knoten bekommen die bezüglich des Nachbarknoten entgegengesetzte Farbe. Bei einer Bejahung dieser Frage beträgt die Anzahl Advices genau $1 \leq n - 2$.
 3. Falls das Orakel bei Schritt 2 verneint, wird es bei jedem isolierten Knoten nach der Farbe gefragt. Das Orakel würde die erste Frage aber nur verneinen, wenn es mindestens 2 nicht isolierte Knoten gibt – ansonsten wäre Optimalität auch noch gewährleistet, wenn die Farbe 1 auf beiden Shores auftauchen würde. Also beträgt die Anzahl Advices höchstens $n - 2$. [3]
- Um jeden bipartiten Graphen mit n Knoten optimal färben zu können, sind mindestens $n - 3$ Advices nötig. [3]
 - Um jeden bipartiten Graphen mit n Knoten mit höchstens $k > 2$ Farben färben zu können, sind höchstens $\frac{n}{\sqrt{2^{k-1}}}$ Advices nötig [3]. Dies ergibt sich aus der Beobachtung, dass die Befragung des Orakels, ob die Farbe k oder $k - 1$ zugewiesen werden soll, nur jeweils notwendig wird, wenn der aktuell präsentierte Knoten $k - 2$ Kanten zu Nachbarn mit genau den Farben $\{1, \dots, k - 2\}$ hat. Durch Induktion lässt sich dann zeigen, dass dies nur für jeden $\sqrt{2^{k-1}}$ ten aller n Knoten der Fall ist [3, 11].

4.3 Experimente

Mit Hilfe des zu dieser Arbeit gehörenden Programms **GraphBonanza** werden wir nun einige Experimente durchführen.

4.3.1 Anzahl Advices für selbe Competitive Ratio wie First Fit

In diesem ersten Experiment wollen wir untersuchen, wie viele Advice Bits durchschnittlich, minimal und maximal benötigt werden, um eine repräsentative Menge von $G(n, p)$ -Graphen zu färben. Dabei wird die maximale Anzahl Farben k für den Algorithmus mit Advice so gewählt, dass er jeweils dem Resultat von First Fit entspricht. Tabelle 4.1 zeigt das Resultat für 10000 Instanzen von $G(n = 1000, p = 0.1)$. Die Daten sind über die vom First Fit-Algorithmus benötigten Anzahl Farben gruppiert (Spalte FF: # colors) und weisen die minimale, maximale und durchschnittliche Anzahl Advices aus, welche der Algorithmus mit Advice angefragt hat. Die Tabellen 4.2 und 4.3 zeigen die Resultate desselben Experiments für $p = 0.5$ respektive $p = 0.9$.

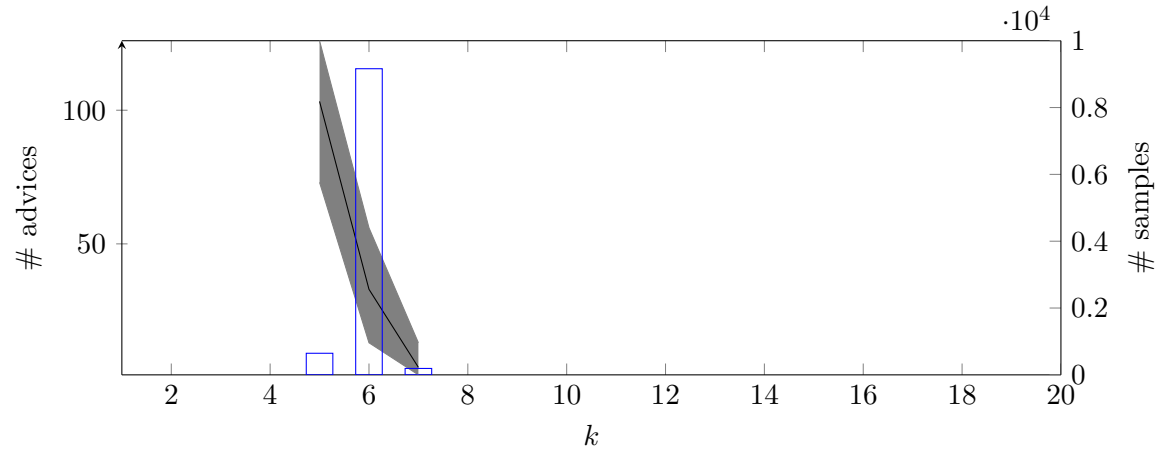
Tabelle 4.1: First Fit gegen Algorithmus mit Advice für 10000 Instanzen von $G(1000, 0.1)$

FF: # colors	# samples	AA: min advices	AA: avg advices	AA: max advices
3	150	15	46.5	174
4	215	13	51.3	164
5	210	9	50.5	199
6	219	9	42.6	107
7	398	5	32.5	127
8	732	1	18.5	115
9	1087	1	15.0	107
10	1499	1	12.9	95
11	1701	1	11.8	83
12	1639	1	10.5	64
13	1141	1	9.6	41
14	656	1	9.2	57
15	271	1	8.0	33
16	63	1	6.9	16
17	11	1	6.2	16
18	1	3	3.0	3

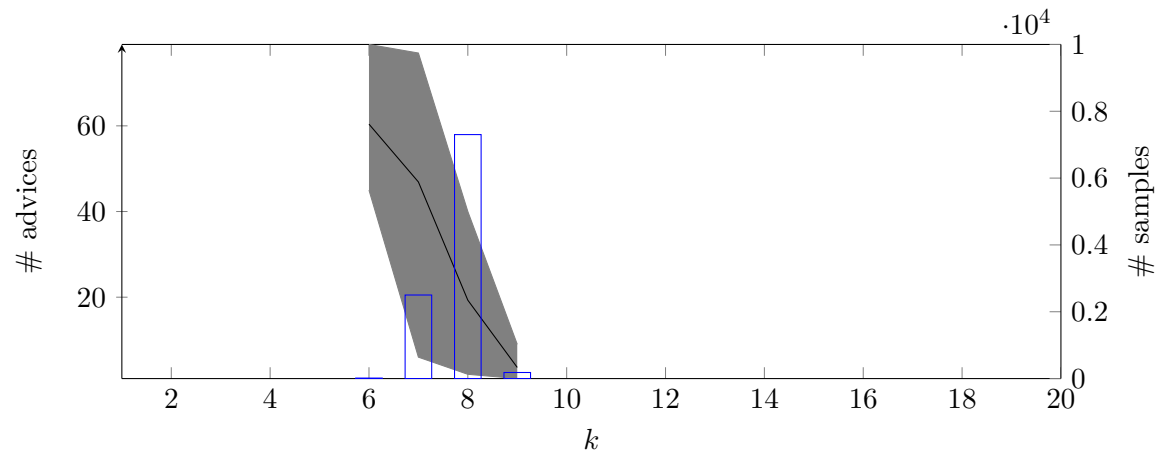
Tabelle 4.2: First Fit gegen Algorithmus mit Advice für 10000 Instanzen von $G(1000, 0.5)$

FF: # colors	# samples	AA: min advices	AA: avg advices	AA: max advices
2	4196	1	4.0	34
3	3057	1	5.5	25
4	1451	1	6.0	28
5	694	1	6.1	31
6	340	1	6.4	29
7	158	1	5.7	42
8	62	1	5.7	20
9	19	2	6.0	16
10	11	2	6.0	13
11	7	2	3.7	6
12	3	3	5.0	8
13	2	5	6.0	7

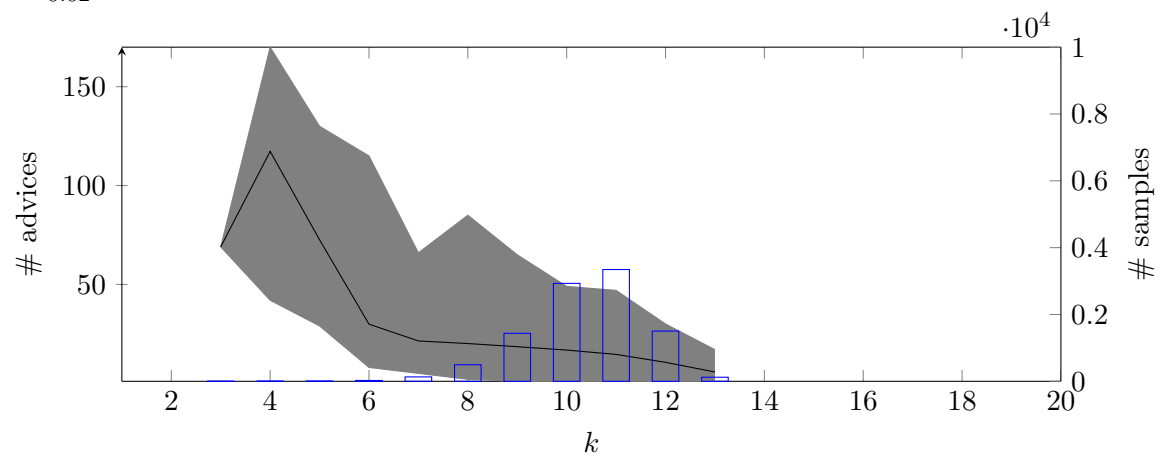
Abbildung 4.1: FF gegen AA für 10000 Instanzen von $G(1000, p)$, I



(a) $p = 0.01$

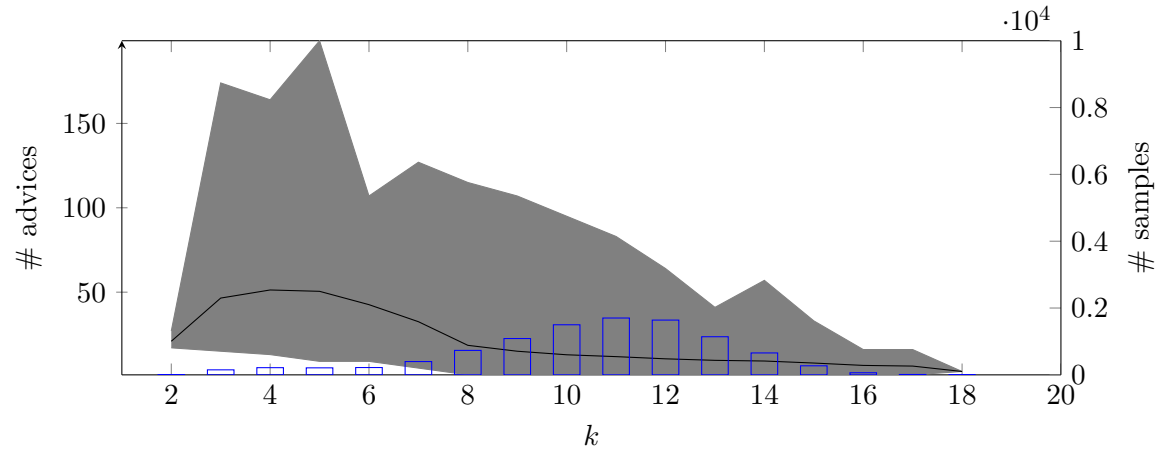


(b) $p = 0.02$

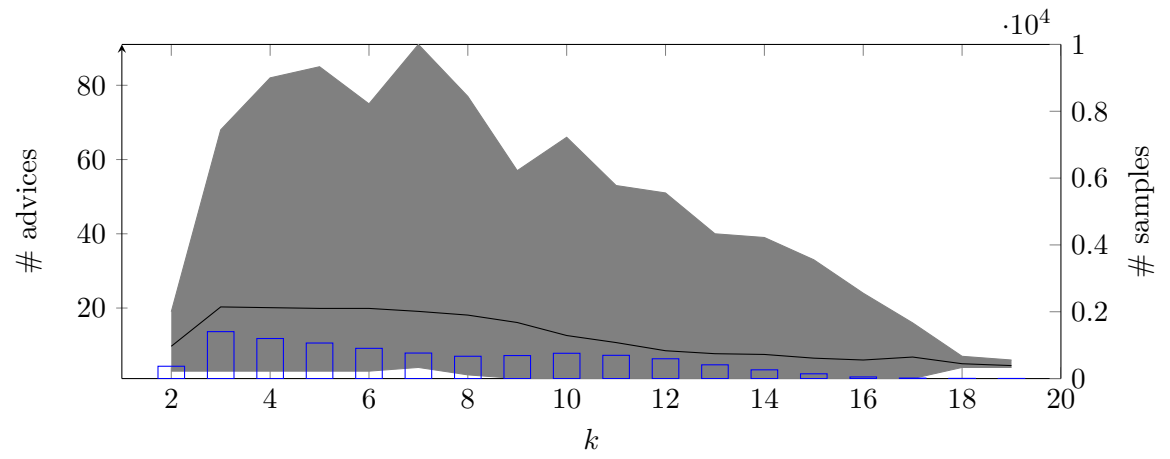


(c) $p = 0.05$

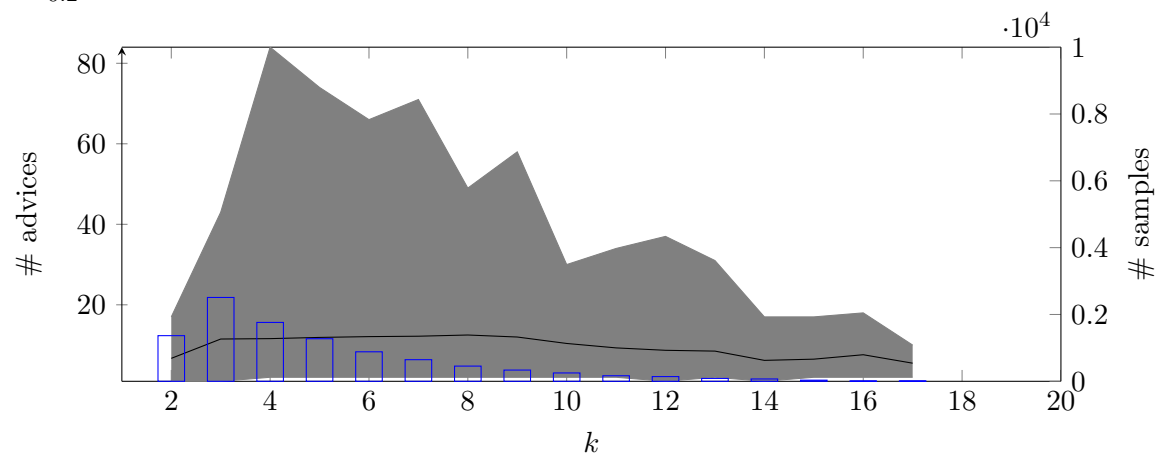
Abbildung 4.2: FF gegen AA für 10000 Instanzen von $G(1000, p)$, II



(a) $p = 0.1$

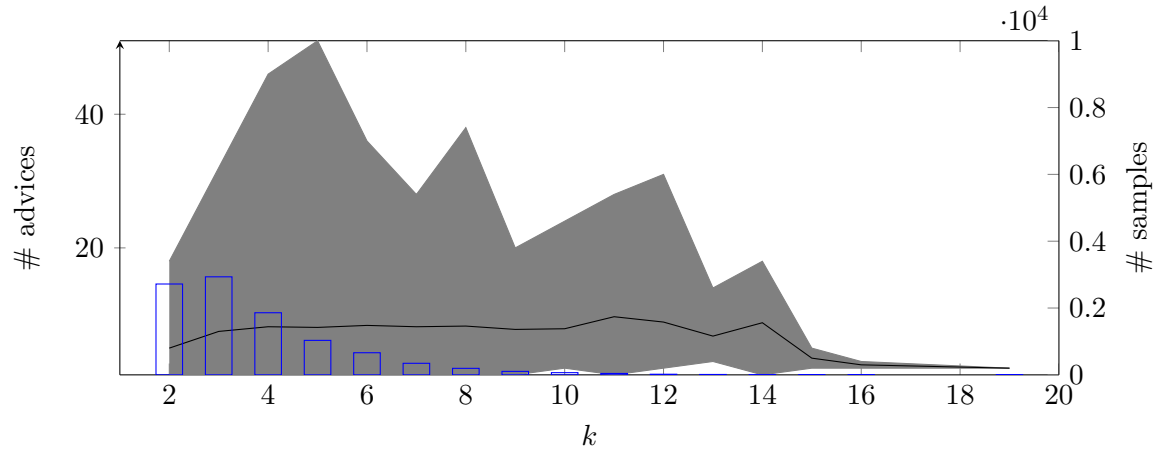


(b) $p = 0.2$

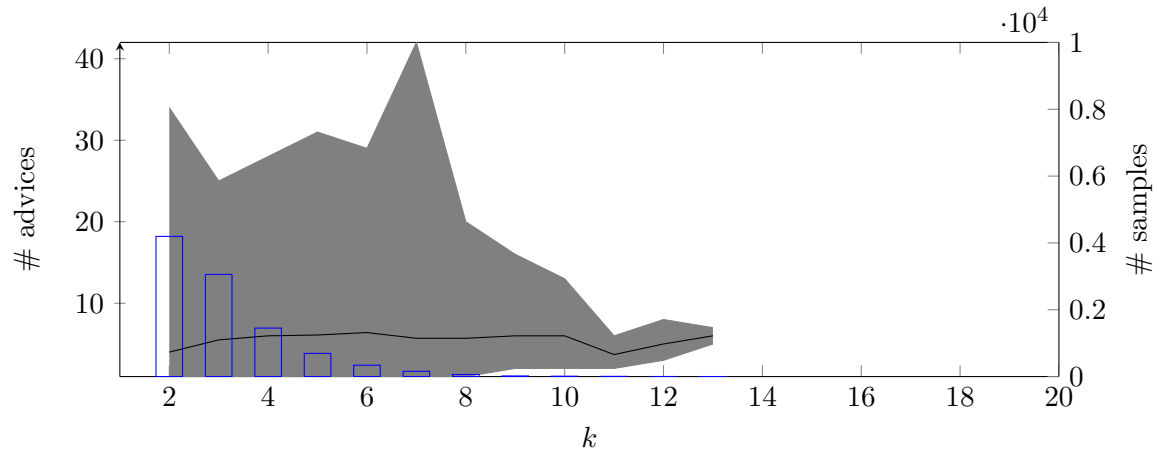


(c) $p = 0.3$

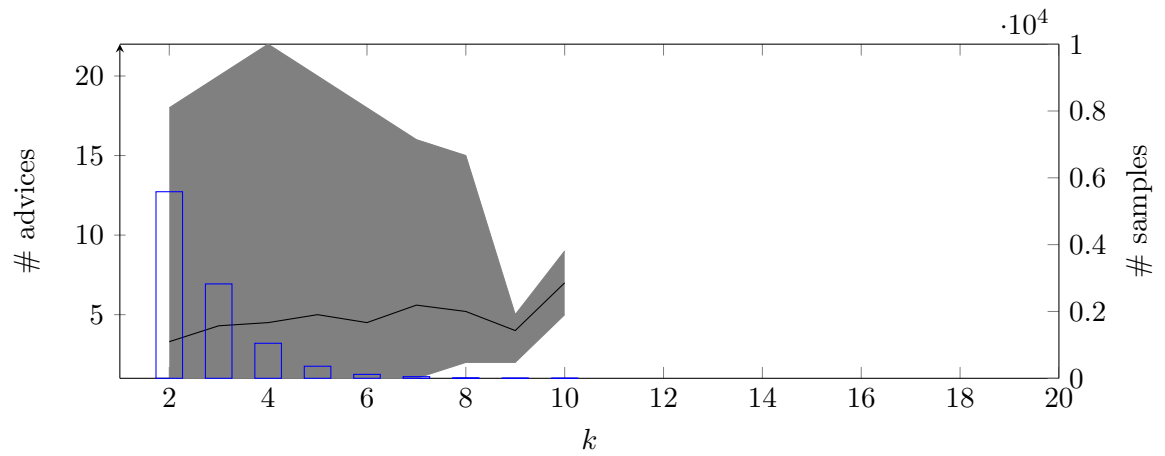
Abbildung 4.3: FF gegen AA für 10000 Instanzen von $G(1000, p)$, III



(a) $p = 0.4$



(b) $p = 0.5$



(c) $p = 0.6$

Tabelle 4.3: First Fit gegen Algorithmus mit Advice für 10000 Instanzen von $G(1000, 0.9)$

FF: # colors	# samples	AA: min advices	AA: avg advices	AA: max advices
2	9024	1	14	199
3	894	1	12	12
4	73	1	10	11
5	9	1	4	4

5 Kommentierung der Implementation des Programms GraphBonanza

In den nachfolgenden Abschnitten sollen ein Überblick über die Anforderungen an das Programm sowie deren technische Umsetzung gegeben werden. Wo notwendig, sinnvoll und/oder interessant, werden auch einzelne Code-Abschnitte wiedergegeben und kommentiert.

5.1 Anforderungen

Pi pa po

5.2 Umfeld, Architektur und verwendete Bibliotheken

Das Programm wurde als reine Browser-Lösung unter Verwendung von **HTML** und **JavaScript** umgesetzt. Dadurch ist es theoretisch auf sämtlichen gängigen Plattformen lauffähig und hat keine Abhängigkeiten zu anderen Komponenten und Diensten wie Laufzeitumgebungen, Webservern, Datenbanken oder ähnlichem. Der Nachvollziehbarkeit und der Verwendung durch Dritte ist ebenfalls förderlich, dass kein Compiler vonnöten ist.

In der Praxis wird die Portabilität natürlich eingeschränkt durch die teilweise unvollständige und fehlerhafte Implementierung von **JavaScript**, **HTML** und **CSS** durch veraltete Browser wie zum Beispiel Internet Explorer. Es zeigt sich aber, dass das Programm in aktuellen Versionen von **Firefox** und **Chrome** wie erwartet funktioniert.

Das UML-Diagramm in Abbildung ?? zeigt einen Überblick über die implementierten Klassen.

5.3 Die GraphBuilder-Klasse

5.4 Die FirstFitAlgo-Klasse

5.5 Die AlgoWithAdvice-Klasse

5.6 Die Looper-Klasse

5.7 User Interface

5.8 Caveats

Literaturverzeichnis

- [1] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Kráľovič, Richard Kráľovič, and Tobias Mömke. On the advice complexity of online problems. In Yingfei Dong, Ding-Zhu Du, and Oscar Ibarra, editors, *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin Heidelberg, 2009.
- [2] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Kráľovič, Richard Kráľovič, and Tobias Mömke. Online algorithms with advice. technical report 614, 2009.
- [3] MariaPaola Bianchi, Hans-Joachim Böckenhauer, Juraj Hromkovič, and Lucia Keller. Online coloring of bipartite graphs with and without advice. In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *Computing and Combinatorics*, volume 7434 of *Lecture Notes in Computer Science*, pages 519–530. Springer Berlin Heidelberg, 2012.
- [4] Norman Biggs. *Algebraic Graph Theory (Cambridge Mathematical Library)*. Cambridge University Press, 2 edition, 2 1994.
- [5] Hans-Joachim Böckenhauer, Dennis Komm, Richard Kráľovič, and Peter Rossmanith. On the advice complexity of the knapsack problem. In *Proceedings of the 10th Latin American International Conference on Theoretical Informatics, LATIN’12*, pages 61–72, Berlin, Heidelberg, 2012. Springer-Verlag.
- [6] Stefan Dobrev, Rastislav Kráľovič, and Dana Pardubská. Measuring the problem-relevant information in input. *RAIRO - Theoretical Informatics and Applications*, 43:585–613, 7 2009.
- [7] P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.
- [8] Georges Gonthier. A computer-checked proof of the Four Colour Theorem. 2005.
- [9] A. Gyárfás and J. Lehel. On-line and first fit colorings of graphs. *Journal of Graph Theory*, 12(2):217–227, 1988.
- [10] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

- [11] H.A. Kierstead and W.T. Trotter. On-line graph coloring. *On-line Algorithms*, 7:88 – 92, 1992.
- [12] László Lovász, Michael Saks, and W.T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75(1–3):319 – 325, 1989.
- [13] A. Marchetti-Spaccamela and C. Vercellis. Stochastic on-line knapsack problems. *Mathematical Programming*, 68(1-3):73–104, 1995.
- [14] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 331–342, New York, NY, USA, 2011. ACM.
- [15] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.
- [16] Luca Trevisan. Cs261: Optimization, lecture 17. <http://theory.stanford.edu/~trevisan/cs261/lecture17.pdf>, 2011.

Abbildungsverzeichnis

2.1	Verlauf von Lösung und Optimum für SkiRental	7
2.2	Verlauf der Competitive Ratio für SkiRental	8
3.1	Ein Graph	13
3.2	Ein bipartiter Graph	13
3.3	Verteilung der Auftretenswahrscheinlichkeit eines Zufallsgraphen	14
3.4	Nicht optimal färbbarer bipartiter Graph	19
3.5	Optimal gefärbter bipartiter Graph	19
3.6	Ein Baum	20
3.7	Die Folge T_1, \dots, T_6	20
3.8	First Fit auf T_1, \dots, T_6	22
4.1	FF gegen AA für 10000 Instanzen von $G(1000, p)$, I	27
4.2	FF gegen AA für 10000 Instanzen von $G(1000, p)$, II	28
4.3	FF gegen AA für 10000 Instanzen von $G(1000, p)$, III	29

List of Algorithms

1	Ein naiver Zufallsgraphen-Generator	15
2	Ein umgeformter naiver Zufallsgraphen-Generator	16
3	Ein Zufallsgraphen-Generator mit Edge Skipping	17
4	First Fit-Algorithmus für Online Graph Coloring	18
5	Online Graph Coloring-Algorithmus mit Advice	24