Diplomprüfung Theoretische Informatik

Effiziente Algorithmen Prof. Vöcking | Termersetzungssysteme Prof. Giesl | Funktionale Programmierung Prof. Giesl

Dauer: ≈ 40-45 min.
Note: 1.0

Zusammenfassung

Sowohl Prof. Giesl als auch Prof. Vöcking sind sehr angenehme, faire Prüfer. Man hat nicht das Gefühl, ausgequetscht zu werden. Das ganze ist eher ein Gespräch und die Atmosphäre ist entspannt. Die Fragen sind nicht die allerleichtesten, manches geht auch über den Vorlesungsstoff hinaus, aber dann darf man auch ruhig etwas länger nachdenken, zur Not bekommt man auch Tips.

1 Effiziente Algorithmen

Zunächst ging es um Approximationsalgorithmen. Sollte erklären was das ist. Dann direkt zu Makespan Scheduling, worum es da geht. Dann LeastLoaded und LongestProcessingTime erklären. Die beiden trivialen Schranken $opt \geq p_i$ und $opt \geq \frac{1}{m}\sum_i p_i$ erwähnt. Dann zeigen, dass LPT eine 4/3-Approx ist, wobei ich nicht mehr zeigen musste, dass LPT für den Spezialfall mit zwei Jobs pro Maschine den optimalen Schedule berechnet, sollte es nur erwähnen.

Dann ging es um das PTAS und die grobe Idee dahinter: Wo kriegt man den optimalen Makespan her? Was sind große und kleine Jobs? Warum skaliert man so, wie man es tut? Nur grob die Idee wie man das Problem auf den skalierten Jobs löst und was die Laufzeit dafür ist. Dann noch argumentieren, warum die kleinen Jobs keine Probleme machen. Kurz darüber gesprochen, dass so ein PTAS nur theoretischen Wert hat, weil der Faktor n^{1/ε^2} zu krass ist.

Als nächstes kam allgemeines Scheduling. Erste Frage, welchen Approximationsfaktor das ganze überhaupt hat, dann den Algorithmus beschreiben. Zunächst das LP grob skizziert: Variablen x_{ij}, die 1 sein sollen, wenn der Job i auf Maschine j kommt. Man darf aber Variable x_{ij} nur einführen, wenn Job i nicht schon Laufzeit größer als optimaler Makespan auf j hat. Dann Allokationsgraphen erklärt. Argumentiert, warum das ein Pseudowald sein muss. Hatte mich hier erst beim LP vertan: Die Gleichungen, dass die x_{ij} kleiner gleich 1 sein müssen kann man sich schenken, da die optimale Lösung das eh erfüllt. Erklärt, dass man nun alle Jobs, die $x_{ij} = 1$ haben, schon mal der entsprechenden Maschine zuordnet. Den Rest musste ich gar nicht mehr erklären.

Sprung zu Cuts. Was ist überhaupt ein Cut? Definition gegeben. Vergessen, dass natürlich die Mengen Q und S nicht leer sein sollten, war aber nicht weiter schlimm (weil es ja im Grunde eh klar ist...). Als nächstes Zusammenhang zu Flussproblemen herstellen. Min-Cut = Max-Flow argumentiert. Sollte das Theorem aber nicht zeigen. Kurz erklären, was das mit linearen Programmen zu tun hat. Also gesagt, dass Min-Cut das zu Max-Flow duale LP ist. Sollte erklären, wie man aus Algo für Max-Flow dann Min-Cut bestimmt, wenn im allgemeinen Min-Cut ja keine Quelle und Senke sind: Beliebigen Knoten als Quelle festhalten und alle anderen mal als Senke probieren.

2 Termersetzungssysteme

Giesl: *Wofür braucht man denn Termersetzungssysteme?*
Ich: In der Vorlesung wollten wir das Wortproblem entscheiden, ansonsten kann man noch damit rechnen, z.B. funktionale Programmiersprachen darüber definieren.
Giesl: *Was ist denn das Worproblem?*
Ich: Haben Menge $\mathcal{E}$ von Gleichungen zwischen Termen und wollen wissen, ob jedes Modell von $\mathcal{E}$ auch $s \equiv t$ für zwei Terme s, t erfüllt.
Giesl: *Ist das im allgemeinen entscheidbar?*
Ich: Nö, erst mal nur für Spezialfälle wenn man nur Grundidentitäten hat, oder ein konvergentes, äquivalentes TES $\mathcal{R}$.
Giesl: *Und semi-entscheidbar?*
Ich: Ja. Satz von Birkhoff sagt, dass wir statt $\equiv_{\mathcal{E}}$ auch $\longleftrightarrow^*_{\mathcal{E}}$ benutzen können, der ja nur auf der Syntax arbeitet. Damit bauen wir dann den Beweisbaum auf ausgehend von s. Wenn wir irgendwo t finden, haben wir es.
Giesl: *Was ist denn der Verzweigungsgrad?*
Ich: Variableneigenschaft erklärt. Unendlicher Grad, wenn die verletzt ist, sonst endlich. Aber auch bei unendlich immernoch semientscheidbar: Baum im Diagonalverfahren durchlaufen.

Giesl: *Gut. Jetzt haben Sie schon richtig gesagt, dass das Worproblem für Grundidentitäten entscheidbar ist. Jetzt stellen wir uns vor, dass in $\mathcal{E}$ zwar nur Grundidentitäten sind, aber wir wollen $s \equiv_E t$ wissen für beliebige Terme s, t. Ist das entscheidbar?*
Ich: Das war mal in den Übungen gewesen, aber nicht in der Vorlesung. Bin deswegen nicht sofort drauf gekommen, dachte erst es wäre unentscheidbar, aber dann etwas mehr überlegt. Giesl sagte, ich solle mehr an die Semantik denken. Bin dann letztlich drauf gekommen, dass man einfach alle Variablen in den Termen s, t durch frische Konstanten ersetzt und sozusagen die Belegung der Variablen durch die Interpretation $I = (\mathcal{A}, \alpha, \beta)$ von der Variablenbelegung β auf die Funktionsinterpretation α umwälzt.
Giesl: *Na gut. Jetzt gehen wir aber mal von dem Fall aus, dass wir das allgemein Wortproblem haben. Das ist ja tatsächlich unentscheidbar. Nehmen wir mal, hm, nicht zu schwer, die Konkatenation von Listen. Schreiben sie das mal als Gleichungssystem.*
Ich:

$$\mathsf{app}(\mathsf{nil}, x) \equiv x$$
$$\mathsf{app}(\mathsf{cons}(x, xs), ys) \equiv \mathsf{cons}(x, \mathsf{app}(xs, ys))$$

Also, leere Liste ist linksneutral und das cons wird nach außen gezogen. Jetzt müssen wir zeigen, dass das TES äquivalent und fundiert ist.
Giesl: *Gut, und jetzt möchten wir wissen, ob* nil *auch rechtsneutral ist. Wie prüfen wir das?*
Ich: Konvergentes, äquivalentes TES $\mathcal{R}$ bestimmen und es damit ausrechnen. Erst mal Gleichungen richten, einfach von links nach rechts. Erst mal zeigen, dass TES fundiert ist. Das müsste mit der RPO gehen. Für erste Gleichung folgt das aus der Einbettungsordnung. Für den anderen Fall brauchen wir die Präzedenz $\mathsf{app} \sqsupset \mathsf{cons}$, dann muss man noch zeigen, dass die linke Seite größer als die Argumente der rechten sind, das folgt sofort aus Einbettungsordnung.
Giesl: *Die Idee ist als: $\mathcal{R}$ terminiert ... es gibt Simplifikationsordnung mit $l \succ r$ für alle Regel. Beim ..., kommt da gdw. rein?*
Ich: Ja. Äh nein! Also klar, wenn ich die Simplifikationsordnung habe, dann terminiert das TES. Aber es gibt TES, die terminieren, aber mit der Simplifikationsordnung kann man das nicht zeigen. Mein erstes Beispiel war Schmu, aber mit $\mathsf{ff} \to \mathsf{fgf}$ geht es, weil pro Schritt die Zahl nebeneinander stehender f kleiner wird. Andererseits ist die linke in die rechte Seite eingebettet.
Giesl: *Gut, und wenn ich aus Simplifikationsordnung Reduktionsordnung mache? Also: Wenn $\mathcal{R}$ terminiert, gibt es dann eine Reduktionsordnung $\succ$ mit $l \succ r$ für alle Regeln?*
Ich: Ja. Die kann man zwar nicht konstruktiv bestimmen, aber wir wählen einfach $\to_{\mathcal{R}}$ für $\succ$.
Giesl: *Ist das eine Reduktions*ordnung*?*
Ich: Oh, nein. Nur eine Reduktionsrelation. Nehmen wir halt die transitive Hülle davon, $\to^+_{\mathcal{R}}$.
Giesl: *Gut. Jetzt terminiert unser $\mathcal{R}$ also. Was müssen wir noch zeigen?*
Ich: Dass das TES auch konfluent ist. Dazu schauen wir uns die kritischen Paare an. Es gibt aber keine. Also ist das TES schon konfluent, also konvergent.
Giesl: *Die Ausgangsfrage war ja: Gilt $\mathsf{app}(x, \mathsf{nil}) \equiv_{\mathcal{E}} x$?*
Ich: Es gilt nicht, weil beide Terme schon Normalformen bezüglich $\to_{\mathcal{R}}$ sind, also werden sie nicht zu denselben Normalformen reduziert. Das müsste aber der Fall sein, wenn die Gleichung gelten würde.
Giesl: *Na gut, aber ist das nicht komisch? Eigentlich stimmt das ja für Listen?*
Ich: Das Problem ist, dass es jede Menge Modelle von $\mathcal{E}$ gibt, auch mit komischen Schmutztermen wo nicht unbedingt nur normale Listen drin sind. Dort gibt es dann auch solche, bei denen nil eben nicht rechtsneutral ist.
Giesl: *Na gut. Und wie ist das jetzt mit den von uns* vorgesehenen *Anwendungen von diesem Gleichungssystem?*
Ich: Induktive Gültigkeit definiert, erklärt, dass unser TES das Definitionsprinzip erfüllt und man zeigen kann, dass nil also für alle Konstruktorgrundterme rechtsneutral sein wird.
Giesl: *Jetzt ist ja die Definition von induktiver Gültigkeit recht unhandlich: Ich muss da irgendwas für alle Konstruktorgrundterme prüfen...*
Ich: Dafür gibt es die Konsistenzbeweismethode. Habe den Satz angeschrieben und die aufwändigere Richtung von beiden gezeigt. Dann war auch die Zeit schon um.

3 Funktionale Programmierng

Giesl: *Haskell ist ja eine funktionale Programmiersprache mit* higher-order-Funktionen*, z.B. gibt es da* `fold`*. Man sagt, dass* `fold` *besonders mächtig ist, weil man damit viele andere bekannte higher-order-Funktionen definieren kann. Als Beispiel wollen wir uns mal* `map` *ansehen.*

Ich: Stand erst mal voll auf dem Schlauch, hatte effektiv nur die Theorie und die Monaden gelernt weil ich dachte, von den normalen Programmiersachen würde nichts drankommen. Zum Glück habe ich die Übungen gemacht... Habe erst Quatsch erzählt mit Konkatenation und so. Sollte dann erst mal `fold` hinschreiben. Das ging zum Glück (vgl. Skript). Jetzt hatte Prof. Giesl den Term `1 : 2 : []` als Baum hingemalt und ich sollte erklären, was `fold g e` macht, bin aber nur mit Mühe drauf gekommen: `fold` ersetzt das `:` durch `g` und das `[]` durch `e`.

Dann erklärt, dass `map f` auf alle Blätter die Funktion `f` anwendet.

Giesl: *Genau. Und wir setzen jetzt mal an:*
`map f = fold _ _` Was muss da denn jetzt wo hin?

Ich: Nach einem ersten falschen Ansatz kam ich dann auf die richtige Lösung:
`fold (\x y -> (f x) : y) []`

Giesl: *Ja genau. (Er klang da recht erfreut. Später meinte er dann auch, dass das recht knifflig ist und man das nicht aus dem Ärmel schütteln können muss).*

Wir springen dann zur denotationellen Semantik. So eine Funktion wie `fold` ist ja rekursiv definiert. Welchen Wert ordnen wir der Funktion zu?

Ich: Weil das rekursiv ist kann man nicht einfach den Wert für die rechte Seite bestimmen und ihn dann zuordnen. Wir brauchen eine higher-order-Funktion, die Funktionen auf diese rechte Seite abbildet. Deren kleinster Fixpunkt, oder aber die kleineste obere Schranke vom $\texttt{ff}^i(\bot)$ liefern dann den Wert.

Giesl: *Wie sähe diese higher-order-Funktion denn aus?*

Ich: Wollte das erst als λ-Ausdruck hinschreiben, Giesl meinte dann ich könne ruhig volles Haskell benutzen. Also
`ff f g e (x : xs) = g x (f g e xs)` Man ersetzt das `fold` durch die nun gebundene Variable `f`.

Giesl: *Und warum sollte dieser kleinste Fixpunkt überhaupt existieren?*

Ich: Die $\texttt{ff}^i(\bot)$ bilden ja eine Kette, also haben sie, weil unsere Ordnung auf dem Funktionsdomain ja vollständig ist, eine kleinste obere Schranke. Da hatten wir den Satz, dass das gleich dem kleinsten Fixpunkt ist.

Giesl: *Dazu muss das `ff` aber stetig sein. Wir haben diese Funktion jetzt einfach so definiert. Wieso ist die stetig?*

Ich: Weil sie berechenbar ist.

Giesl: *Ja genau. Ich meine, wir haben sie mit einer Programmiersprache beschrieben, also können wir das auführen, also ist das berechenbar. Gut, machen wir einen Sprung zum Typechecking. Wie sieht es denn mit dem Typ von diesem λ-Term aus: `x x`*

Ich: Hier hatte ich mich erst vertan und den Algorithmus falsch angewendet und hatte dann raus, dass der Ausdruck unkorrekt getypt sei, weil das erste `x` aus der Typannahme $\forall a.a$ den Typ a bekäme und dann die Typannahme zu a verfeinert wird. Ersteres stimmt, aber die Typannahme wird nicht verändert. Nach ein paar gezielten Nachfragen von Prof. Giesl bin ich dann aber darauf gekommen, dass der Typ von `x x` einfach `b -> c` ist, weil sowohl das erste als auch das zweite `x` beliebigen Typ haben können.

Giesl: *Und wie sieht das bei λ-Abstraktionen aus?*

Ich: Da verlangt man, dass das Typschema flach ist, also dass jedes Vorkommen einer Variable innerhalb des Ausdrucks auch genau denselben Typ hat.

Dann war die Zeit auch schon um (das geht immer viel schneller, als man sich das denkt) und nach kurzer Wartezeit durfte ich meine Note abholen. Die 1.0 hat mich sehr gefreut, wegen der paar Unsicherheiten war ich nicht mehr sicher, ob das noch Hinhauen würde. Aber Prof. Giesl meinte, dass das ja auch schwierigere Sachen waren, die man sich dann erst in der Prüfung erarbeiten soll. Nur beim Typecheking hätte ich eigentlich nicht drauf reinfallen sollen...

Eine Prüfung kann ich bei beiden Professoren nur empfehlen. Als Tip zur Prüfungsvorbereitung kann ich empfehlen, wenn möglich schon beim Besuch der Vorlesung gut mit zu arbeiten und z.B. die Übungen zu machen. Das Lernen und Wiederholen fällt einem dann spürbar leichter, jedenfalls gingen mir genau die Themen aus Effiziente Algorithmen, bei denen ich in der Vorlesung war und die Blätter mitgemacht hatte, viel leichter von der Hand.

Wie man sieht gehen die Fragen durchaus auch ins Detail, springen aber ab und zu auch hin und her. Auf Lücke sollte man also lieber nicht lernen sondern zu allem wenigstens die Grundideen parat haben.

Noch ein letzter Tip: Es bringt nichts, sich schön getexte *Zusammenfassungen* des Stoffs zu schreiben. Insbesondere das Skript von Prof. Vöcking ist kurz und prägnant genug. Besser ist es, sich zu jedem Kapitel zielgerichtete Fragen aufzuschreiben. Das Überfliegen von Zusammenfassungen gibt nämlich kein Feedback darüber, ob man es jetzt wirklich selbst verinnerlicht hat, während man bei den Fragen direkt gefordert ist.