

Greedy Algorithms - Gierige Algorithmen

Marius Burfey

23. Juni 2009

Inhaltsverzeichnis

1	„Greedy Algorithms“	1
2	Interval Scheduling - Ablaufplanung	2
2.1	Problembeschreibung	2
2.2	Entwurf eines gierigen Algorithmus	2
2.3	Beispiele für Auswahlregeln	2
2.4	Die optimale Regel	3
2.5	Algorithmus	3
3	Analyse des Algorithmus	4
3.1	Kompatibilität	4
3.2	Optimalität der Schritte	4
3.3	Optimalität der Gesamtlösung	5
3.4	Implementierung und Laufzeit	6
4	Erweiterung der Problemstellung	7
4.1	Das Interval Partitioning Problem	7
4.2	Entwicklung eines Algorithmus zum Interval Partitioning Problem	8
4.3	Algorithmus zum Interval Partitioning Problem	9
4.4	Analyse des Algorithmus	9

1 „Greedy Algorithms“

Ein Algorithmus heißt „greedy“, bzw. gefräßig oder gierig, wenn er eine Problemlösung in kleinen Schritten aufbaut, wobei bei jedem Schritt nur kurzfristig unter Berücksichtigung einer Entscheidungsregel die optimale Lösung für diesen Schritt gewählt wird.

Wenn ein gieriger Algorithmus erfolgreich terminiert, und dabei eine optimale Lösung findet, sagt dies über das Problem aus, dass es eine lokale Entscheidungsregel gibt, die man zur Konstruktion global optimaler Lösungen heranziehen kann. Einen gierigen Algorithmus für nahezu jedes Problem zu finden, ist leicht; die interessante Herausforderung liegt darin, Fälle zu finden, in denen er gut bzw. optimal arbeitet und dies zu beweisen.

2 Interval Scheduling - Ablaufplanung

2.1 Problembeschreibung

Wir betrachten eine Zusammenstellung von Anfragen $\{1,2,\dots,n\}$, die jeweils eine Startzeit $s(i)$ und eine Abschlusszeit $f(i)$ haben, und möchten, dass möglichst viele von ihnen auf einer bestimmten Ressource erledigt werden.

Eine Anfragemenge ((sub-)set of requests) heißt *kompatibel*, wenn keine zwei Anfragen überlappen.

Ziel des Interval Scheduling, also der Ablaufplanung, ist es, einen möglichst großen kompatiblen Teil der Anfragen zu akzeptieren. Kompatible Sets maximaler Größe werden *optimal* genannt.

2.2 Entwurf eines gierigen Algorithmus

Ansatz:

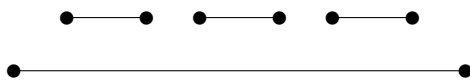
Wir wählen anhand einer einfachen Regel die erste Anfrage i_1 und lehnen anschließend alle Anfragen ab, die zu i_1 nicht kompatibel sind. Danach wählen wir ein i_2 und lehnen alle Anfragen ab, die zu i_2 nicht kompatibel sind. Dies wird wiederholt bis keine Anfragen mehr übrig sind.

Die Herausforderung liegt darin, eine einfache, aber effektive Regel zur Auswahl der nächsten Anfrage zu entwerfen. Es gibt sehr viele Möglichkeiten für Regeln, die keine guten Lösungen bringen.

2.3 Beispiele für Auswahlregeln

- Eine sehr offensichtliche Regel könnte sein, immer die Anfrage zu wählen, die als erstes startet, also diejenige mit minimalem $s(i)$.

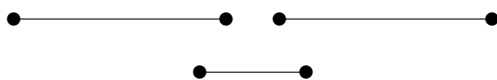
→ Die Ressource wird so schnell wie möglich belegt.



Problem: Wenn die erste Anfrage sehr lange läuft, werden ggf. viele kurze, aber spätere Anfragen abgelehnt.

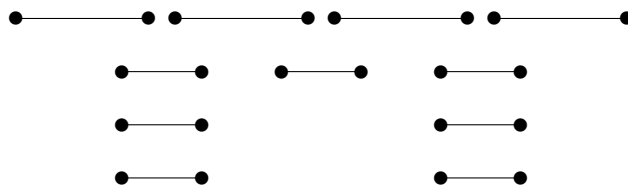
- Dieses Problem wird gelöst, indem man die Aufträge mit der kürzesten Laufzeit, also minimalem $f(i) - s(i)$, wählt.

→ Die Ressource wird möglichst kurz belegt.



Problem: Wenn eine kurze Anfrage zwei andere Anfragen überlappt, wird nur diese gewählt, obwohl man besser die zwei anderen akzeptieren sollte.

- Um auch dieses Problem umgehen zu können, wählt man die Anfrage mit den wenigstens Überlappungen bzw. inkompatiblen Anfragen.
→ Die Ressource wird möglichst überlappungsfrei belegt.



Problem: Auch hier können Anfragen mit wenigen Überlappungen entscheidende Anfragen überlappen, die dann nicht ausgewählt werden.

2.4 Die optimale Regel

Die gierige Regel, die zur optimalen Lösung führt, basiert darauf, die Ressource so früh wie möglich wieder freizugeben, also wird die Anfrage, deren $f(i)$ minimal ist, gewählt.

2.5 Algorithmus

Greedy Algorithm 1 Interval Scheduling Problem

initialisiere R als Set aller Anfragen und A als leer

while R ist noch nicht leer **do**

 wähle eine Anfrage i , die die kleinste Abschlusszeit $f(i)$ hat

 füge Anfrage i zu R hinzu

 lösche alle Anfragen aus R , die nicht kompatibel zu Anfrage i sind

end while

return das Set A als das Set der akzeptierten Anfragen

3 Analyse des Algorithmus

Wir zeigen zunächst, dass das erzeugte Set kompatibel ist. (\rightarrow 3.1)

Anschließend zeigen wir, dass das Set auch optimal ist:

Zu Vergleichszwecken führen wir ein optimales Set \mathcal{O} von Intervallen ein.

Vorgehen: Der gierige Algorithmus ist stets mindestens genau so gut wie jeder andere Algorithmus - wenn nicht sogar besser.

Bei diesem Verfahren wird gezeigt, dass der gefräßige Algorithmus in jedem Teilschritt besser abschneidet als jeder andere Algorithmus und dadurch auch insgesamt eine optimale Lösung erzeugt. Im konkreten Fall vergleichen wir jede Teillösung mit den anfänglichen Teilabschnitten aus \mathcal{O} , um zu zeigen, dass jeder Schritt die beste Lösung wählt. (\rightarrow 3.2)

Im Anschluss zeigen wir, dass auch die durch dieses Vorgehen erzielte Gesamtlösung optimal ist. Idealerweise müsste gezeigt werden, dass $A = \mathcal{O}$, aber da es mehrere optimale Lösungen geben kann, zeigen wir nur, dass A mindestens genau so gut ist wie eine davon. \rightarrow Es werden gleich viele Anfragen akzeptiert: $|A| = |\mathcal{O}|$ (\rightarrow 3.3)

3.1 Kompatibilität

Im Algorithmus (Zeile 5) werden jeweils die Anfragen aus der Menge der verfügbaren Anfragen entfernt, die nicht kompatibel zur aktuell betrachteten Anfrage sind. Also folgt aus der Vorgehensweise des Algorithmus direkt, dass die im Set A enthaltenen Anfragen kompatibel sind.

1. *A ist ein kompatibles Set von Anfragen*

3.2 Optimalität der Schritte

Notation:

i_1, \dots, i_k ist das Set von Anfragen in A in der Reihenfolge, in der sie hinzugefügt wurden. Beachte: $|A| = k$

j_1, \dots, j_m ist das Set von Anfragen in \mathcal{O} , sortiert von links nach rechts in Reihenfolge von Start- und Abschlusszeiten. Da \mathcal{O} kompatibel ist, müssen die Startzeiten in der selben Reihenfolge sein wie die Abschlusszeiten.

2. $f(i_r) \leq f(j_r) \forall r \leq k$

Beweis. Wir zeigen dies mit Hilfe von Induktion:

Für $r=1$ gilt die Behauptung, da der Algorithmus startet, indem er die Anfrage i_1 mit der minimalen Abschlusszeit wählt:

$$f(i_1) \leq f(j_1)$$

Für $r > 1$: Wir nehmen an, dass die Behauptung für $r - 1$ stimmt und zeigen sie für r .

$f(i_{r-1}) \leq f(j_{r-1})$ trifft zu. Damit nun das r -te Intervall des Algorithmus (i_r) nach dem optimalen Intervall (j_r) endet, muss der Algorithmus ein späteres Intervall als i_r wählen. Da er aber immer die Möglichkeit hat, j_r zu wählen, wird er nie ein i_r wählen, das nach j_r endet.

\Rightarrow Behauptung gilt auch für r .

Formal:

Da \mathcal{O} aus kompatiblen Intervallen besteht, gilt: $f(j_{r-1}) \leq s(j_r)$.

Einsetzen der Hypothese $f(i_{r-1}) \leq f(j_{r-1})$ führt zu:

$$f(i_{r-1}) \leq s(j_r)$$

Das heißt: Das Intervall j_r ist in R , wenn der gierige Algorithmus i_r auswählt.

Da der Algorithmus das verfügbare Intervall mit der kleinsten Abschlusszeit wählt und j_r zu diesen verfügbaren Intervallen gehört, gilt: $f(i_r) \leq f(j_r)$

3.3 Optimalität der Gesamtlösung

Nun möchten wir zeigen, dass die akzeptierte Menge von Anfragen des Algorithmus nicht nur für jeden Schritt sondern auch insgesamt besser ist, als die optimale Lösung \mathcal{O} .

Zu zeigen ist: $k = m$, also dass A genau so viele Anfragen enthält wie \mathcal{O} :

3. Der gierige Algorithmus liefert ein optimales Set A .

Beweis. Widerspruchsbeweis:

Wenn A nicht optimal ist, dann muss das optimale Set \mathcal{O} mehr Anfragen enthalten, also $m > k$. Wir nutzen (2) mit $r=k$:

$$f(i_k) \leq f(j_k)$$

Da $m > k$, müsste es eine Anfrage j_{k+1} in \mathcal{O} geben. Diese Anfrage müsste nach j_k enden und deshalb auch nach i_k .

Nachdem der Algorithmus alle Anfragen gelöscht hat, die nicht mit den Anfragen i_1, \dots, i_k kompatibel sind, müsste in R immer noch j_{k+1} enthalten sein. Da der Algorithmus mit i_k endet und dies nur tut, wenn R leer ist, kommen wir zu einem Widerspruch.

3.4 Implementierung und Laufzeit

Zu Beginn des Algorithmus werden die n Anfragen nach ihrer Abschlusszeit aufsteigend sortiert und in dieser Reihenfolge benannt; also $f(i) \leq f(j)$ für $i < j$. Dies beansprucht $O(n \log n)$ Zeit.

In zusätzlich $O(n)$ Zeit erstellen wir ein Array $S[1 \dots n]$ mit der Eigenschaft, dass $S[i]$ den Wert $s(i)$ enthält.

Nun wählen wir die Anfragen, indem wir mit aufsteigendem $f(i)$ durch die Intervalle laufen. Zunächst wählen wir das erste Intervall und iterieren so lange, bis ein Intervall erreicht wird, für das erstmalig $s(j) \geq f(1)$ gilt. (Also die erste Anfrage, die nach Abschluss der ersten Anfrage startet.) Dieses Intervall wird ebenfalls gewählt.

Allgemein: Wir laufen durch die Intervalle und wählen jeweils das Intervall j für das erstmalig $s(j) \geq f$ gilt, wobei f die Abschlusszeit des letztgewählten Intervalls ist. Das bedeutet, dass nur ein Mal durch die Intervalle gelaufen werden muss, dieser Teil des Algorithmus also $O(n)$ Zeit benötigt.

Insgesamt ergibt sich also eine Laufzeit von $O(n \log n)$.

4 Erweiterung der Problemstellung

Das oben betrachtete Problem ist ein recht simples Ressourcenbelegungsproblem; in der Realität können sich weitere Probleme ergeben:

- Wir gehen davon aus, dass dem Algorithmus alle Anfragen bekannt sind, wenn er das optimale Set auswählt. In der Praxis hingegen kann es vorkommen, dass der Planer bzw. das Planungssystem schon Entscheidungen über Annahme oder Ablehnung bestimmter Anfragen treffen muss, bevor er bzw. es Information über **alle** Anfragen hat.
→ **Online Algorithms**, welche ohne Wissen über zukünftige Eingaben entscheiden.
- Unser Ziel war es, die Anzahl der erfüllten Anfragen zu maximieren. Man könnte davon ausgehen, dass verschiedene Anfragen zu unterschiedlichen Profiten führen, also Werte v_i haben. Das Ziel wäre dann, das Einkommen zu maximieren.
→ **Weighted Intervall Scheduling Problem**

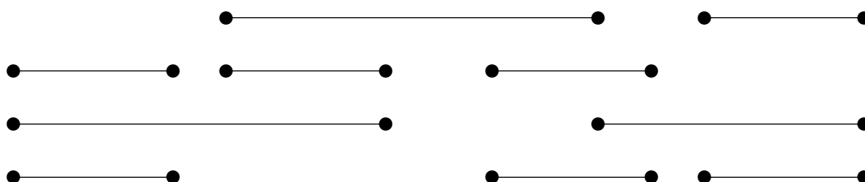
Es können diverse Variationen und Kombinationen auftreten, wir betrachten nun eine weitere Möglichkeit:

4.1 Das Interval Partitioning Problem

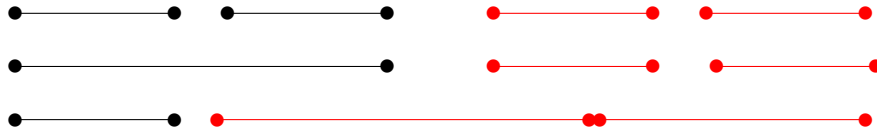
Im obigen Problem gibt es eine einzelne Ressource und viele Anfragen in Form von Zeitintervallen. Es werden *möglichst viele Anfragen* für die eine Ressource angenommen, während der Rest abgelehnt wird.

Ein anderes Problem ist, alle Aufträge zu bearbeiten und dabei *möglichst wenige Ressourcen* zu nutzen. Da hier alle Intervalle aufgeteilt werden sollen, nennt man dies **Interval Partitioning Problem**.

Beispiel:



optimal:



Wir können uns im Allgemeinen eine Lösung unter Verwendung von k Ressourcen vorstellen, wenn wir alle Anfragen in k Zeilen von nicht überlappenden Intervallen darstellen können.

Im Beispiel ist es nicht möglich, weniger als 3 Ressourcen zu nutzen, da sich z.B. die Anfragen a, b und c überlappen und daher verschiedene Ressourcen benötigen. Wir definieren die **Tiefe** d eines Intervallsets als die maximale Anzahl von Intervallen, die zu einem Zeitpunkt parallel laufen.

4. *In jeder Instanz eines Interval Partitioning Problems ist die Anzahl der benötigten Ressourcen mindestens so groß wie die Tiefe des Intervalls.*

Beweis. Wir nehmen an, dass ein Intervall die Tiefe d hat. Seien I_1, \dots, I_d Intervalle, die zu einem Zeitpunkt parallel laufen. Dann muss jedes dieser Intervalle eine eigene Ressource zugewiesen bekommen.

\Rightarrow Es werden mindestens d Ressourcen benötigt.

4.2 Entwicklung eines Algorithmus zum Interval Partitioning Problem

Wir betrachten nun zwei Fragen:

1. Können wir einen effizienten Algorithmus entwerfen, der alle Intervalle unter Verwendung der minimal möglichen Anzahl Ressourcen zuweist?
2. Gibt es immer eine Aufteilung, sodass eine Anzahl Ressourcen zugewiesen wird, die genau der Tiefe entspricht?

Wir entwerfen einen einfachen gierigen Algorithmus, der alle Intervalle zuweist und dabei nur so viele Ressourcen benutzt, wie die Tiefe ist. Das impliziert die Optimalität des Algorithmus (\rightarrow (4)), da keine Lösung weniger Ressourcen nutzen kann. Die Analyse des Algorithmus wird einen anderen Ansatz zum Beweis der Optimalität nutzen: Man behauptet, dass jede mögliche Lösung mindestens einen bestimmten Wert erreichen muss, und zeigt, dass der zu betrachtende Algorithmus diese Schranke immer erreicht.

4.3 Algorithmus zum Interval Partitioning Problem

Sei d die Tiefe des Intervallsets. Wir zeigen, wie man jedem Intervall ein Label zuweist, wobei die Labels in $\{1, 2, \dots, d\}$ liegen und die Zuweisung die Eigenschaft hat, dass überlappende Intervalle unterschiedliche Nummern erhalten.

Der Algorithmus ordnet die Intervalle nach ihrer Startzeit und durchläuft sie in dieser Reihenfolge. Er versucht, jedem betrachteten Intervall eine Nummer zuzuweisen, die noch nicht einem anderen überlappenden Intervall zugewiesen wurde.

Greedy Algorithm 2 Interval Partitioning Problem

```
sortiere die Intervalle nach ihrer Startzeit
bezeichne die Intervalle in dieser Reihenfolge mit  $I_1, I_2, \dots, I_n$ 
for  $j = 1$  to  $n$  do
  for jedes Intervall  $I_i$ , das vor  $I_j$  steht und es überlappt do
    schlieÙe Label von  $I_i$  von der Einteilung aus
  end for
  if es gibt ein Label  $\{1, 2, \dots, d\}$ , das nicht ausgeschlossen wurde then
    weise  $I_j$  ein nicht zugewiesenes Label zu
  else
    lasse  $I_j$  ohne Label
  end if
end for
```

4.4 Analyse des Algorithmus

5. *Unter Verwendung des obigen Algorithmus wird jedem Intervall ein Label zugewiesen und keine zwei überlappenden Intervalle erhalten das selbe Label.*

Beweis:

1. Zeige, dass kein Intervall nicht zugewiesen wird: Betrachte ein Intervall I_j und nehme an, dass t Intervalle, die früher in der sortierten Liste kommen, dieses Intervall überlappen.

Diese t Intervalle bilden mit I_j ein Set von $t + 1$ Intervallen, die alle an einem bestimmten Punkt im Zeitablauf passieren (die Startzeit von I_j). Das heißt: $t + 1 \leq d$ Also: $t \leq d - 1$

Also ist mindestens eins der d Labels nicht von diesem Set von t Intervallen ausgeschlossen. \rightarrow Es existiert ein Label, das I_j zugewiesen werden kann.

2. Zeige, dass keine überlappenden Intervalle das selbe Label zugewiesen bekommen haben: Nehme an, dass zwei Intervalle I und I' sich überlappen und I' in der Ordnung nach I kommt. Wenn I' vom Algorithmus betrachtet wird, gehört I zu dem Intervallset, dessen Label von der Betrachtung ausgeschlossen sind. Daher wird der Algorithmus I' nicht das selbe Label zuweisen, das er für I benutzt hat.

Wenn man d Labels zur Auswahl hat und dann von links nach rechts durch die Intervalle läuft und jedem Intervall ein verfügbares Label zuweist, kann man nie einen Punkt erreichen, an dem alle Labels in Benutzung sind.

Da unser Algorithmus d Labels benutzt, können wir (4) benutzen, um zu schließen, dass er immer die minimale Anzahl Labels nutzt.

6. *Der obige gierige Algorithmus teilt jedem Intervall eine Ressource zu, wobei er so viele Ressourcen benutzt, wie die Tiefe des Intervallsets. Dies ist die optimale Anzahl benötigter Ressourcen.*