8 Intervall–Scheduling mit Gewichten

Wir betrachten eine Verallgemeinerung des Intervall–Scheduling Problems, welches wir früher bereits kennengelernt haben. In der uns bekannten Version gibt es n potenzielle Nutzer einer zentralen Resource. Nutzer j spezifiziert ein Zeitintervall $R_j = [a_j, b_j)$, in dem er die Resource zugeteilt bekommen möchte. Anfragen mit überlappenden Zeitintervallen können nicht simultan befriedigt werden. Ziel ist es, eine möglichst große Anzahl von Anfragen positiv zu bescheiden. In der im Folgenden zu diskutierenden allgemeineren Variante ist jedes Intervall R_j mit einem Gewicht $w_j \in \mathbb{N}$ versehen. Intuitiv könnte w_j die Wichtigkeit des Jobs reflektieren, welchen Nutzer j mit Hilfe der zentralen Resource auszuführen gedenkt. Oder w_j ist der Geldbetrag, den Nutzer j für die Zuteilung der Resource zu zahlen bereit ist. Dies führt uns zu folgendem Problem mit dem Namen „Intervall–Scheduling mit Gewichten“.

Eingabe: eine Kollektion $R = (R_j)_{j=1,\ldots,n}$ von Intervallen der Form $R_j = [a_j, b_j)$ mit $0 \leq a_j < b_j$ sowie eine entsprechende Kollektion $(w_j)_{j=1,\ldots,n}$ von natürlich-zahligen Gewichten.

Aufgabe: Auffinden einer Menge $I \subseteq [n]$, welche unter der Nebenbedingung

$$\forall i \neq j \in I : R_i \cap R_j = \emptyset \tag{1}$$

ein maximales Gesamtgewicht

$$w(I) = \sum_{i \in I} w_i$$

aufweist.

Offensichtlich kollabiert das allgemeinere Problem zu dem uns von früher bekannten Problem, wenn wir alle Gewichtsparameter auf den Wert 1 setzen, d.h., „Intervall–Scheduling mit Einheitsgewichten“ ist dasselbe wie „Intervall–Scheduling“.

Wie wir wissen, lässt sich Intervall–Scheduling optimal lösen mit Hilfe folgender „gierigen Regel“ zur Auswahl von in I aufzunehmenden Intervallen: *Wähle als nächstes Intervall stets dasjenige mit der kleinstmöglichen Endzeit aus.*[1]

[1] natürlich unter Beachtung der Nebenbedingung (1)

Dass dies bei der Variante mit Gewichten i.A. nicht optimal ist, zeigt folgendes Beispiel:

$$R_1 = [1,3), R_2 = [2,5), R_3 = [4,6), w_1 = 1, w_2 = 3, w_3 = 1 \ .$$

Die gierige Auswahlregel nimmt zunächst 1 in I auf (was 2 zur Aufnahme in I disqualifiziert) und dann 3, d.h., sie führt zur Lösung $I = \{1,3\}$ mit Gesamtgewicht 2. Optimal wäre die Lösung $I^* = \{2\}$ mit Gesamtgewicht 3.

Wir nehmen im Folgenden an, dass die Intervalle $R_j = [a_j, b_j)$ aufsteigend nach dem Schlüssel b_j sortiert sind. Zudem nummerieren wir die Intervalle so um, dass anschließend die Bedingung $b_1 \leq b_2 \leq \ldots \leq b_n$ gilt. Sortieren plus Umnummerieren beansprucht Rechenzeit $O(n \log n)$.

Die folgende Abbildung $j \mapsto p(j)$ wird bei der Lösung unseres Problems eine zentrale Rolle spielen:
Es sei $p(j)$ der größte Index i mit $b_i \leq a_j$ (bzw. $p(j) = 0$, falls kein solcher Index i existiert).
Diese Definition impliziert, dass $p(j)$ kleiner als j ist und dass die Intervalle mit den Indizes $p(j)+1, \ldots, j-1$ sich mit R_j überlappen. Die Zahl $p(j)$ ist zudem gleich der Anzahl der Intervalle, die vor (oder genau zum) Zeitpunkt a_j bereits beendet sind.

Beispiel 8.1 *Für*

$$R_1 = [1,4), R_2 = [2,6), R_3 = [5,7), R_4 = [3,10), R_5 = [8,11), R_6 = [9,12)$$

gilt

$$p(1) = 0, p(2) = 0, p(3) = 1, p(4) = 0, p(5) = 3, p(6) = 3 \ .$$

Wir werden später nachweisen, dass sich die $p(j)$-Werte in Zeit $O(n \log n)$ berechnen lassen (bzw. sogar in Linearzeit, wenn vorher die Daten geeignet sortiert wurden). S. Lemma 8.4 weiter unten.

Es sei I^* eine optimale Lösung unseres Problems. Obwohl wir momentan keine Ahnung haben, wie I^* aussieht, lassen sich folgende Beobachtungen machen:

1. Falls $n \in I^*$, dann folgt $p(n)+1, \ldots, n-1 \notin I^*$ und I^* besteht aus $\{n\}$ vereinigt mit der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_{p(n)}$.

2. Falls $n \notin I^*$, dann ist I^* identisch zu der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_{n-1}$.

Im Folgenden bezeichne $\mathrm{OPT}(j)$ das Gesamtgewicht der optimalen Lösung für das Teilproblem zur Kollektion $R_1, \ldots, R_j$, wobei wir $\mathrm{OPT}(0) = 0$ setzen. Offensichtlich gilt folgende Rekursion:

1. $\mathrm{OPT}(0) = 0$.

2. $\mathrm{OPT}(n) = \max\{w_n + \mathrm{OPT}(p(n)) \ , \ \mathrm{OPT}(n-1)\}$.

Der erste (bzw. zweite) Term in dem Max-Ausdruck steht für die Festlegung n in I aufzunehmen (bzw. nicht aufzunehmen).

Die obige Rekursion könnte uns auf die Idee bringen, das Problem rekursiv zu lösen. Dies führt leider zu einem Algorithmus mit exponentieller Laufzeit (im worstcase):

Beispiel 8.2 *Für $j = 1, \ldots, n$ sei $R_j = [2j-2, 2j+1)$. Dann führt der rekursive Aufruf mit einem Parameter $j \geq 2$ zu zwei rekursiven Aufrufen: einer mit Parameter $j-2$ und einer mit Parameter $j-1$. Die Anzahl der Knoten in dem aus Aufruf OPT(n) resultierenden Rekursionsbaum beträgt daher $F(n)$: das n-te Glied der Fibonacci-Folge. Wie wir wissen hat die Folge $(F_n)_{n\geq 0}$ eine exponentielle Wachstumsrate.*

Das Problem mit der rekursiven Prozedur ist, dass der Aufruf für ein und denselben Parameter j vielfach getätigt wird.[2] Eine Technik, dies zu vermeiden, besteht in Rekursion plus „Memorisation“:
Beim ersten rekursiven Aufruf mit Parameter j wird das Ergebnis tabelliert. Vor jedem Aufruf wird kontrolliert, ob bereits eine Tabellierung vorliegt und, falls dem so ist, erfolgt ein „Table-Lookup“ anstelle des rekursiven Aufrufes.

Es ist jedoch konzeptionell einfacher, den „Top-Down“-Ansatz der Rekursion durch einen „bottom-up“ Ansatz zu ersetzen:
Man beginnt mit Teilproblemen einer trivialen Größe und kämpft sich zu immer größeren Teilproblemen durch. Jedes Ergebnis wird tabelliert. Beim Betrachten eines Teilproblemes kann man daher stets auf die Lösungen noch kleinerer Teilprobleme zurückgreifen.
Diesen Ansatz, bekannt unter dem Namen „dynamische Programierung“,

[2] Bei MergeSort und anderen erfolgreichen „Divide & Conquer“-Verfahren tritt dieses Problem *nicht* auf, weil ein Problem stets in *disjunkte* Teilprobleme zerlegt wird.

werden wir nun auf unser Problem der Berechnung der OPT(j)-Werte anwenden. Dabei setzen wir voraus, dass die $p(j)$-Werte bereits berechnet wurden und in einem Array $p[1:n]$ zur Verfügung stehen. Die Tabelle (= Array) $OPT[0:n]$ ergibt sich dann in Linearzeit wie folgt:

1. OPT$[0] \leftarrow 0$.

2. Für $j = 1, \ldots, n$: OPT$[j] \leftarrow \max\{w_j + \text{OPT}[p[j]]\ ,\ \text{OPT}[j-1]\}$.

Der Wert OPT$[n]$ ist dann identisch zum Gesamtgewicht $w(I^*)$ einer optimalen Lösung I^*.

Freilich sind wir nicht nur an dem Gesamtgewicht OPT$[n] = w(I^*)$ einer optimalen Lösung interessiert, sondern auch an der optimalen Lösung selbst. Dazu müssen wir lediglich in obigem Rechenschema mit Hilfe eines Booleschen Arrays $B[1:n]$ protokollieren, welcher der beiden Vergleichsterme mit dem Maximum übereinstimmt. Genau dann, wenn dies der erste von beiden Termen ist, setzen wir $B[j]$ auf TRUE. Danach durchlaufen wir die Indizes $j \in [n]$ in absteigender Reihenfolge, beginnend bei $j = n$. Im Falle $B[n] =$ TRUE müssen wir n in I^* aufnehmen und mit dem Index $j = p(n)$ (in der gleichen Weise) weitermachen. Im Falle $B[n] =$ FALSE nehmen wir n nicht in I^* und machen mit dem Index $n-1$ (inder gleichen Weise) weiter. Ein entsprechend erweitertes Rechenschema liest sich wie folgt:

1. OPT$[0] \leftarrow 0$.

2. Für $j = 1, \ldots, n$:
   Falls $w_j + \text{OPT}[p[j]] \geq \text{OPT}[j-1]$:
   OPT$[j] \leftarrow w_j + \text{OPT}[p[j]]$; $B[j] \leftarrow$ TRUE.
   Andernfalls:
   OPT$[j] \leftarrow \text{OPT}[j-1]$; $B[j] \leftarrow$ FALSE.

3. $I^* \leftarrow \emptyset$; $j \leftarrow n$.

4. Solange $j \neq 0$ mache Folgendes:
   Falls $B[j]$: $I^* \leftarrow I^* \cup \{j\}$; $j \leftarrow p(j)$.
   Andernfalls: $j \leftarrow j - 1$.

I^* wird dabei als Liste verwaltet. Offensichtlich beansprucht diese Konstruktion von I^* lediglich Linearzeit (also Zeit $O(n)$).

Fassen wir das Hauptergebnis dieses Abschnittes zusammen:

Satz 8.3 *Das Problem „Intervall–Scheduling mit Gewichten“ ist in Rechenzeit $O(n \log n)$ optimal lösbar. Genauer: wenn die Hilfswerte $p(1), \ldots, p(n)$ bereits vorliegen, so genügt Linearzeit.*

Bleibt uns noch die Pflicht darzulegen, wie die Hilfswerte $p(j)$ in Zeit $O(n \log n)$ berechnet werden können:

Lemma 8.4 *Betrachte die Tripelmenge*

$$T = \{(1, a_1, 1), \ldots, (n, a_n, 1)\} \cup \{(1, b_1, 0), \ldots, (n, b_n, 0)\} \ ,$$

wobei wir $(a_j, 1)$ bzw. $(b_j, 0)$ als Schlüsselwert des Tripels $(j, a_j, 1)$ bzw. $(j, b_j, 0)$ betrachten. Es sei L eine nach diesen Schlüsselwerten aufsteigend sortierte Liste aller Tripel aus T. Dann gilt: aus L lassen sich die Werte $p(1), \ldots, p(n)$ in Linearzeit berechnen.

Beweis Das Rechenschema ist wie folgt:

1. $p \leftarrow 0$.
2. Solange L nicht vollständig abgearbeitet ist, mache Folgendes:
   (a) Entnimm L das nächste Tripel, sagen wir (j, c_j, β).
   (b) Falls $\beta = 1$ (und somit $c_j = a_j$): $p(j) \leftarrow p$.
   Andernfalls (also falls $\beta = 0$ und somit $c_j = b_j$): $p \leftarrow p + 1$.

Die Variable p zählt, wieviele Intervalle aktuell schon beendet sind. Dem Index j wird ein p-Wert zugewiesen, wenn wir in der Liste L den Zeitpunkt a_j erreicht haben. Somit wird dem Index j über $p(j) \leftarrow p$ stets der richtige Wert zugewiesen. •

Da das Sortieren der Tripel aus T nur Zeit $O(n \log n)$ beansprucht, ergibt sich die

Folgerung 8.5 *Die Hilfswerte $p(1), \ldots, p(n)$ sind aus den Eingabedaten des Problems „Intervall–Scheduling mit Gewichten“ in Zeit $O(n \log n)$ berechenbar.*