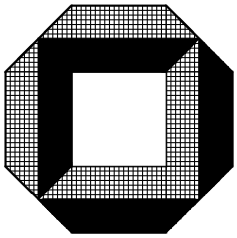
Proseminar

Stochastische Dynamische Optimierung



Universität Karlsruhe (TH)
Fakultät für Mathematik
Institut für Mathematische Stochastik

Priv.-Doz. Dr. D. Kadelka
Dr. B. Klar

Sommersemester 2004

Institut für Mathematische Stochastik
Fakultät für Mathematik
Universität Karlsruhe
Englerstraße 2
76 128 Karlsruhe

Stochastic Scheduling

Clemens Lode

Inhaltsverzeichnis

Kapitel 3

Stochastic scheduling

3.1 Einleitung

Stochastic Scheduling beschäftigt sich mit Problemen, bei denen die Laufzeiten eines Jobs durch Zufallsvariablen dargestellt sind. Dadurch ist die tatsächliche Zeit vor Ende des Jobs unbekannt. Das 'Scheduling' ist je nach Problem auf ein oder mehrere Prozessoren und mit verschiedenartigen Nebenbedingungen bestückt.

Hier im Speziellen möchte ich auf vier Fälle aus diesem Bereich eingehen die mit der Abarbeitung von Jobs auf ein oder mehreren Prozessoren zu tun haben. Dabei benötigt jeder Job eine gewisse Zeit X_j, die eine Exponentialverteilung besitzt.

3.2 Fall: Ein Prozessor

Zum Einstieg ein triviales Problem um mit den verwendeten Variablen und ein paar einfachen Gesetzmässigkeiten im weiteren Verlauf besser zurechtzukommen. Das Problem in diesem Fall ist, dass wir n Jobs auszuführen haben. Ein Job ist dabei eine beliebige Aufgabe die ein Prozessor berechnen kann, der eine bestimmte Zeit braucht, wobei die Jobs nicht gleich sein müssen. Außerdem haben wir im ersten Fall einen einzelnen Prozessor zur Verfügung, der immer nur einen Job gleichzeitig bearbeiten kann. Die Fragestellung ist nun, welche Reihenfolge der Jobs, im Folgenden Politik genannt, die Zeit um alle Jobs auszuführen minimiert.

Was ist die benötigte Gesamtzeit unserer Jobs? Bei einem realistischen Beispiel wäre das schwer zu sagen. Verbraucht z.B. ein Job während und nach der Bearbeitung sehr viel Speicher wäre wohl das Beste diesen hinten hinzustellen, damit die anderen Jobs nicht z.B. auf der langsamen Festplatte ausgelagert werden müssen. Die Zeit die ein einzelner Job benötigt wäre also vom bisherigen Verlauf abhängig.

Wir machen es uns jedoch einfach und begrenzen uns auf den Fall, dass *alle Jobs unabhängig voneinander* bearbeitet werden können. Dies bedeutet in

diesem Fall, dass zu jedem Zeitpunkt auf Basis der bisherigen Ereignisse die bedingte Wahrscheinlichkeit fuer jeden neuen Job eine bestimmte Zeit zu brauchen genau so groß ist, als wäre dies der erste Job in der Auftragsliste.

Im Weiteren werde ich folgende Definitionen und Voraussetzungen gebrauchen:

- M: benötigte Gesamtzeit bis alle Jobs abgearbeitet sind
- Politik
$$\pi = (i_1, i_2, \ldots, i_n)$$
wobei $i_j \neq i_k$ für $j \neq k$ und $0 < i_j \leq n$ eine Jobnummer bezeichnet.
- X_j ist eine Zufallsvariable mit Exponentialverteilung mit Parameter λ_j

Im Weiteren gehen wir außerdem davon aus, dass zum Zeitpunkt *0* bereits ein Job *0* auf dem Prozessor liegt.

Die Gesamtzeit unserer Jobs ist dann natürlich die Summe aller Zeiten und somit von der Reihenfolge unabhängig. Also ist es, wie man es erwartet hätte, vollkommen egal, welche Politik man wählt. Für die erwartete Gesamtzeit ergibt sich also:

$$E(M) = E(X_{i_1}) + E(X_{i_2}) + \cdots + E(X_{i_n}) = \sum_{j=1}^{n} E(X_j)$$

wobei die Summe unabhängig von den i_j, also der Reihenfolge der Jobs, ist.

Abbildung 3.1: Möglicher Durchlauf bei gegebener Politik $\pi_1 = (1, 2, 3)$

Abbildung 3.2: Der selbe Durchlauf mit anderer Politik $\pi_2 = (2, 3, 1)$

Mit dieser kleinen Einführung können wir uns nun an ein schwierigeres Problem wagen, wir legen im zweiten Teil einen weiteren Prozessor hinzu.

3.3 Fall: Zwei Prozessoren parallel

3.3.1 Problemdefinition und Ansatz

Es sind wieder eine Reihe von Jobs gegeben, die unterschiedlich lange Zeit benötigen. Außerdem hat man ein System mit diesmal zwei identische Prozessoren zur Verfügung. Die einzelnen Jobs sind stochastisch unabhängig und exponential verteilt. Das Problem ist nun wieder, eine Politik für die Abarbeitung der Jobs zu finden, so dass die insgesamt benötigte Zeit minimal ist.

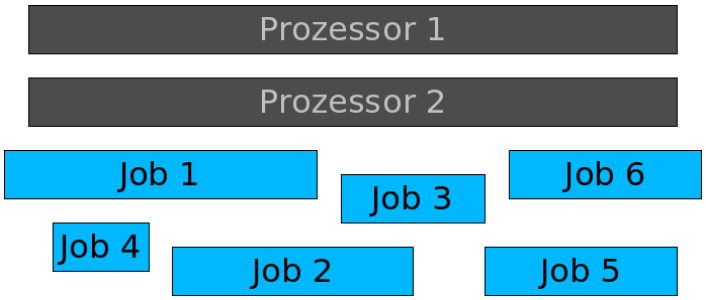


Abbildung 3.3: Übersicht Problemstellung zwei Prozessoren - mehrere Jobs müssen auf zwei Prozessoren verteilt werden

Ein erster Ansatz wäre, die Politiken so aufzubauen, dass wir jedem Job j einen Zeitpunkt k und einen Prozessor l zuordnen. Zwar sind damit alle möglichen Kombinationen abgedeckt, jedoch ergibt sich beim Zeitpunkt k ein Problem: Es ergeben sich einerseits Zeiten in der Prozessoren ungenutzt sind (siehe Abb. 3.4), die deshalb auftreten, weil die Ausführungszeiten X_j der Jobs zu Beginn unbekannt sind. Andererseits gibt es keine festen Zeiteinheiten auf die wir unsere Jobs verteilen könnten was zu unendlich vielen Möglichkeiten führen würde. Vor allem auf Grund der ungenutzten Zeiten bei dieser Art von Politik kann in diesem Fall die optimale Lösung für die gegebenen Voraussetzungen nicht gefunden werden.

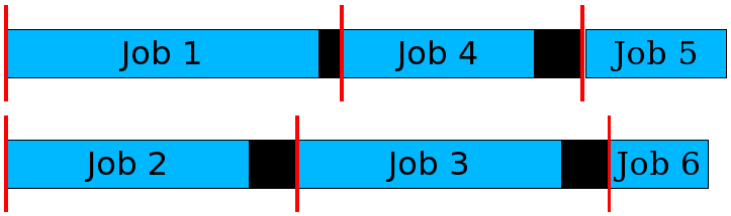


Abbildung 3.4: Zustand Prozessoren mit Politik die nach Zeit zuordnet - schwarz markierte Bereiche geben ungenutzte Zeiten des Prozessors an

3.3.2 Benutzung von Expertenwissen und Onlineregeln

Die Einteilung nach den Zeitpunkten können wir umgehen indem wir eine Voraussetzung, sogenanntes *Expertenwissen*, für unser Problem benutzen. Wir müssen

nämlich über die genaue Positionen der Jobs zu Beginn noch nichts wissen, wir können uns entscheiden welchen Job wir als nächstes einfügen sobald einer der beiden Prozessoren frei wird. Wir dürfen also durch diese Onlineregel die Jobs nahtlos aneinander anfügen, was zur Schließung der Lücken führt. Intuitiv ist klar, dass wir dadurch auch die optimale Lösung erreichen können, da wir nun die Möglichkeit haben, die Jobs so in die Prozessoren einzufügen, dass der Abstand beider 'Jobsäulen', also D minimal wird. Ein genauer mehrseitiger Beweis, dass Politiken die den Prozessor zu keiner Zeit (außer am Ende) unbeschäftigt lassen nicht optimal sind, ist in der Literatur (z.B. für den Fall der Tandemschaltung [3]) finden.

Der zweite Versuch eine Politik aufzustellen, wäre also, dass man jeden Job in eine der zwei Prozessoren in einer bestimmten Reihenfolge einfügt. Wir ordnen jedem der n Jobs einen Prozessor zu (2^n Möglichkeiten) und legen die Reihenfolge der n Jobs fest ($n \cdot (n-1) \cdots 2 \cdot 1 = n!$ Möglichkeiten) wodurch sich insgesamt $2^n n!$ Belegungen für die Politik ergibt.

Da man dabei mehrere Politiken erhält, die eine gleiche Belegung der Jobs auf die Prozessoren zur Folge hat (z.B. 3 und 4 auf Prozessor 1, 1 und 2 auf Prozessor 2 $\Rightarrow$ 1, 2, 3, 4 führt zur selben Belegung wie 3, 4, 1, 2) ist die Zahl der unterschiedlichen Politiken geringer. Wir können k Jobs auf einen und $(n-k)$ Jobs auf den anderen Prozessor legen und dann $n!$ bzw. $(n-k)!$ verschiedene Reihenfolgen Aaufstellen:

$$\sum_{k=0}^{n} \sum_{A:|A|=k} k!(n-k)! = \sum_{k=0}^{n} \binom{n}{k} k!(n-k)! = \sum_{k=0}^{n} n!$$

$$= (n+1)!$$

3.3.3 Aufstellung der zu minimierenden Zielfunktion

Seien zwei identische Prozessoren gegeben um n Jobs abzuarbeiten. Beide laufen unabhängig, können also jede Aufgabe zu jeder Zeit in einer beliebigen Reihenfolge bearbeiten. Startzeit ist $t = 0$.

Für jede Politik π ist jeweils M die Bearbeitungsdauer und D die Zeit die einer der Prozessoren stillsteht. Somit ist zum Zeitpunkt $M - D$ einer der Prozessoren fertig und keine weiteren Aufträge mehr in der Warteschlange und $M + M - D$ ist die Summe aller Zeiten, also $\sum_{j=0}^{n} X_j$

Also gilt: $2M - D - \sum_{j=0}^{n} X_j = 0$ und somit für jede Politik π:

$$E_\pi(2M - D - \sum_{j=0}^{n} X_j) = E_\pi(0)$$

$$2E_\pi(M) = E_\pi(D) + \sum_{j=0}^{n} E_\pi(X_j)$$

Da nach Voraussetzung die X_j von einander also insbesondere auch von der Wahl der Politik π, unabhängig sind, ist $\sum_{j=0}^{n} E(X_j)$ eine Konstante für jede Politik π (für die gleichen X_j) und wir setzen $c = \sum_{j=0}^{n} E(X_j)$:

$$E_\pi(M) = \frac{E_\pi(D) + c}{2} \tag{3.1}$$

Also ergibt sich, dass durch *Minimierung der erwarteten Differenz D* in der einer der Prozessoren unbeschäftigt ist, auch die Bearbeitungsdauer minimiert wird.

3.3.4 Verfeinerung der Politik

Zwar könnte man unter Umständen auch mit dem Modell jedem Job einen Prozessor zuzuordnen auf die einzelnen Aussagen und Sätze kommen, jedoch ist es bei Optimierungsaufgaben grundsätzlich hilfreich, die Zahl der unterschiedlichen Politiken so weit wie möglich zu verringern. Dabei ist natürlich immer darauf zu achten, dass der optimale Fall mit der Politik noch herzustellen ist.

Im Weiteren wird ausgenutzt, dass wir eben keine stationäre Entscheidung vor Beginn des Einfügens treffen müssen, sondern während des Laufs in jedem Moment eingreifen können. Dass dies nur sinnvoll ist, wenn gerade ein Job beendet wurde bzw. ein Prozessor frei ist, wurde im vorherigen Abschnitt gezeigt. Dass es auch möglich ist, die Zuweisungen einzelner Jobs zu den Prozessoren zu Beginn wegfallen zu lassen, wird im Folgenden gezeigt.

Behauptung: Eine Politik die die Jobs nach ihrer Reihenfolge immer auf den Prozessor legt, der gerade frei ist, ist gleichwertig mit einer Politik die zu Beginn jedem Job einen Prozessor zuweist und auf die jeweiligen Prozessoren entsprechend nach der Reihenfolge legt.

Mit "gleichwertig" ist hier gemeint, dass mit beiden Politiken die Jobs so verteilt werden können, dass die insgesamte benötigte Zeit bei beiden Politiken gleich und minimal ist, dass also vor allem der optimale Fall nicht wegfällt

Beweisidee: Es ist klar, dass der Versuch auf den einen Prozessor alle Jobs und auf den anderen Prozessor keine Jobs zu verteilen für $n > 1$ nicht zum optimalen Ergebnis führt, da die erwartete Differenz D in dem Fall maximal wäre. So eine 'kaputte' Politik könnte man aber 'reparieren' in dem man, zu jedem Zeitpunkt zu dem beiden Prozessoren keine Aufgabe zu bearbeiten hat, einfach die nächste Aufgabe aus unserer Politik zuweist. Jede Politik die mit der Konstruktionsmethode aus der Behauptung nicht hergestellt werden kann, kann keine optimale Politik sein, da dies bedeuten würde, dass am Ende zwei Politiken hintereinander gesetzt werden, obwohl zum Ende des vorletzten Jobs der andere Prozessor frei ist. Dagegen hätte ein früheres Verschieben des letzten Jobs auf den unbeschäftigten Prozessor eine Verbesserung zur Folge.

Wenden wir diese Regel auf jede unserer Politiken an, folgt, dass eine Politik die alle n Jobs auf einen Prozessor legt, *gleichwertig* zu allen anderen Kombinationen der Verteilung der Jobs ist.

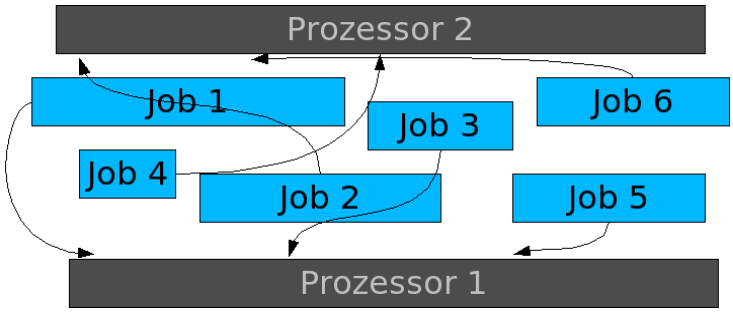


Abbildung 3.5: Zwei Prozessoren - eine Politik

Dies reduziert unseren Zahl unterschiedlicher Politiken auf **n!** und erleichtert die folgenden Beweise, da wir nur noch Politiken haben die die Jobs der Reihenfolge nach anordnen, anstatt noch jedem Job einen Prozessor zuzuordnen.

Bemerkung: Diese Politik ist zwar für sich gesehen stationär, jedoch nicht die daraus resultierende Verteilung der Jobs auf die Prozessoren, da wir während der Bearbeitung noch mit der Regel 'Lege Job auf einen unbeschäftigten Prozessor' die Reihenfolge anpassen.

Die Frage ist nun, ob durch unsere Regel das Problem nicht schon gelöst wurde. Wenn wir immer eine Aufgabe auf den Prozessor legen, der nicht beschäftigt ist, gleichen wir bei jedem Schritt beide Balken aus. Erreichen wir dadurch das optimale Gleichgewicht? Nein. Dies würde der Fall sein, wenn alle Jobs gleiche Zeit benötigten und eine gerade Anzahl Jobs vorliegt, also auf jeden Prozessor gleich viele Jobs gelegt werden können und somit auch D minimal wäre. Legen wir aber beispielsweise die Jobs mit absteigenden λ_i (= aufsteigenden erwartete Zeiten) auf die Prozessoren, schwankt der Unterschied D zwischen beiden bei jedem weiteren Schritt mehr:

Schritt	Idle-Zeit	1. Prozessor	2. Prozessor
0	1	1	0
1	1	1	2
2	2	4	2
3	2	4	6
4	3	9	6
5	3	9	12
...	...	...	...

Investiert man in die Aufgabe etwas Denkarbeit wird auch klar, dass wohl die optimale Politik wäre, wenn wir grundsätzlich mit den längsten Jobs beginnen und am Schluss die kürzesten Jobs legen. Warum? Weil wir bei jedem Schritt die Aufgabe grundsätzlich nur auf den Prozessor legen, der gerade frei ist. Wir versuchen also bereits dadurch die beiden Balken auszugleichen. Je später wir eine längere Aufgabe auf einen Prozessor legen, desto unwahrscheinlicher können wir diesen mit den restlichen Jobs wieder ausgleichen. Dies bleibt nun noch formal zu zeigen.

3.3.5 Erster Beweisschritt: Vergleich zweier Politiken

Sei im weiteren Verlauf zu Begin bereits ein Job *0* bereits im Prozessor, d.h. starte jede Politik mit Job *0* (dies macht den Beweis etwas klarer).

Als nächstes wird gezeigt, dass eine Politik die in aufsteigender Reihenfolge der λ_i (also der absteigenden erwarteten Bearbeitungszeiten) angeordnet ist, optimal ist. Dazu betrachten wir zuerst zwei nahezu gleiche Politiken, bei denen Job *1* und Job *2* miteinander vertauscht sind und beweisen dann induktiv:

Lemma 1: Seien zwei Politiken π und $\bar{\pi}$ gegeben:

$$\pi = (0, 2, 1, i_3, i_4, \ldots, i_n)$$

$$\bar{\pi} = (0, 1, 2, i_3, i_4, \ldots, i_n)$$

Gelte außerdem, dass $\lambda_1 = min_{k\geq 1}(\lambda_k)$, dann gilt:

$$E_{\bar{\pi}}(D) \leq E_{\pi}(D)$$

.

Beweis: Seien p_j und $\bar{p}_j$, $j \geq 0$ die zu π und $\bar{\pi}$ gehörigen Wahrscheinlichkeiten, dass die letzte von den Prozessoren abgeschlossene Aufgabe Job j ist, also z.B. Prozessor *1* fertig ist und Prozessor *2* noch an Aufgabe j arbeitet und keine weiteren Jobs mehr in der Warteschlange/Politik liegen.

$$p_0 = \bar{p}_0 = \mathbb{P}(X_0 > \sum_{j=1}^{n} X_j)$$

ist klar. Der erste Job entspricht der Wahrscheinlichkeit, dass X_0 grösser ist als alle anderen zusammengenommen, da p_0 ja jeweils der erste Job in beiden Politiken π und $\bar{\pi}$ ist.

Wir wollen nun durch Induktion zeigen, dass

$$\bar{p}_1 \leq p_1, \qquad \bar{p}_j \geq p_j, \qquad j = 2, \ldots, n \tag{3.2}$$

Induktionsanfang: Für $n = 1$ fällt $\bar{p}_j \geq p_j$ für $j = 2, \ldots, n$ sowieso weg, da $n < 2$ und $\bar{p}_1 \leq p_1$ gilt nach der Voraussetzung $\lambda_1 = min_{k\geq 1}(\lambda_k)$, da π an der Stelle 1 λ_1 stehen hat und $\bar{\pi}$ dagegen λ_2.

Induktionsvoraussetzung: Es gelte also die Behauptung (3.2) für den Fall, dass nur n-1 Jobs (zusätzlich zu Job 0) vorliegen und somit:

$$\bar{p}_1^* \leq p_1^*, \qquad \bar{p}_j^* \geq p_j^*, \qquad j = 2, \ldots, n-1 \tag{3.3}$$

Induktionsschluss: Seien nun diesmal $(n-1)+1 = n$ Jobs zu erledigen und seien p_j^* und $\bar{p}_j^*$ (wieder zugehörig zu π und $\bar{\pi}, j = 1, \ldots, n-1$ die Wahrscheinlichkeiten, dass Job j der letzte der Jobs $0, \ldots, n-1$ für die jeweilige Politik ist. Da die Jobs unabängig sind, können wir zur Betrachtung des letzten Jobs alle vorherigen abgeschlossenen Jobs ignorieren, diese haben keine Auswirkung auf die Bearbeitungszeit für den letzten Job. Allerdings liegt unter Umständen auf dem anderen Prozessor noch ein letzter Job n der zu Beginn des letzten

Jobs noch nicht abgeschlossen ist. Nach der Gedächtnislosigkeit der Jobs, die aus der Exponentialverteilung folgt, läuft dieser Job n noch X_n Zeiteinheiten obwohl er schon einige Zeit gelaufen ist, genauer gesagt bleibt die Verteilung der Lebensdauer eines Jobs, der s Einheiten gelaufen ist, dieselbe ist wie die eines neuen Jobs. Dies können wir schreiben in der Form:

$$\mathbb{P}(X > s + t | X > s) = \mathbb{P}(X > t)$$

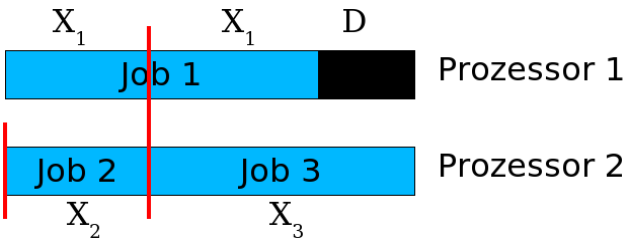


Abbildung 3.6: Abschneiden unter Ausnutzung der Gedächtnislosigkeit - Kein Einfluss auf X_1

Daraus folgt, dass die Wahrscheinlichkeit, dass Job j und nicht Job n der letzte bearbeitete Job ist, $\mathbb{P}(X_j > X_n)$ entspricht:

$$\mathbb{P}(X_j > X_n) = \int_{x=0}^{\infty} \mathbb{P}(X_n < x | X_j = x) f(x, \lambda_j) dx = \int_0^{\infty} (1 - e^{-\lambda_j x}) \lambda_j e^{-\lambda_n x} dx$$

$$= \int_0^{\infty} (\lambda_j e^{-\lambda_j x} - \lambda_j e^{(-\lambda_n - \lambda_j)x}) dx = [-e^{-\lambda_j x} + \frac{\lambda_j}{\lambda_j + \lambda_n} e^{(-\lambda_n - \lambda_j)x}]_0^{\infty}$$

$$= 1 - \frac{\lambda_j}{\lambda_j + \lambda_n} = \frac{\lambda_j + \lambda_n - \lambda_j}{\lambda_j + \lambda_n}$$

$$= \frac{\lambda_n}{\lambda_n + \lambda_j}$$

Da diese Wahrscheinlichkeit unabhängig von der Wahrscheinlichkeit ist, dass j im Fall mit insgesamt $n-1$ Jobs der letzte Job ist, können wir die Gesamtwahrscheinlichkeit durch einfache Multiplikation berechnen:

$$p_j = p_j^* \mathbb{P}(X_j > X_n) = p_j^* \frac{\lambda_n}{\lambda_n + \lambda_j}, \qquad j = 1, \ldots, n-1$$

$$\bar{p}_j = \bar{p}_j^* \mathbb{P}(X_j > X_n) = \bar{p}_j^* \frac{\lambda_n}{\lambda_n + \lambda_j}, \qquad j = 1, \ldots, n-1$$

Daraus und aus (3.3) folgt, dass

$$\bar{p}_1 \leq p_1, \qquad \bar{p}_j \geq p_j, \qquad j = 2, \ldots, n-1 \tag{3.4}$$

Um nun auch p_n und $\bar{p}_n$ vergleichen zu können benutzen wir die Gegenwahrscheinlichkeit und bilden die Differenz:

$$\bar{p}_n - p_n = (1 - \sum_{j=0}^{n-1} \bar{p}_n) - (1 - \sum_{j=0}^{n-1} p_n) = \sum_{j=0}^{n-1} (p_j - \bar{p}_j)$$

$$= \sum_{j=0}^{n-1} (p_j^* - \bar{p}_j^*) \frac{\lambda_n}{\lambda_n + \lambda_j} =$$

Der nullte Job ist bei beiden Politiken gleich, fällt also weg. Dreht man außerdem die Differenz herum und nutzt $\frac{\lambda_n}{\lambda_n+\lambda_j} = 1 - \frac{\lambda_j}{\lambda_j+\lambda_n}$ ergibt sich:

$$= \sum_{j=1}^{n-1} (\bar{p}_j^* - p_j^*) \frac{\lambda_j}{\lambda_j + \lambda_n}$$

Dies können wir mit Hilfe von $\lambda_j \geq \lambda_1 \Leftrightarrow \frac{\lambda_j}{\lambda_j+\lambda_n} \geq \frac{\lambda_1}{\lambda_1+\lambda_n}$ abschätzen. Die Summe über alle Differenzen der Wahrscheinlichkeiten ist *0*, da die Summe aller Wahrscheinlichkeiten *1* ergibt, da ein Job ja der letzte Job sein muss.

$$\geq \frac{\lambda_1}{\lambda_1 + \lambda_n} \sum_{j=1}^{n-1} (\bar{p}_j^* - p_j^*) = 0$$

Aus $\bar{p}_n \geq p_n$ und (3.4) folgt nun die Behauptung (3.2) und die Induktion ist komplett.

Mit diesem Ergebnis können wir nun die Erwartungswerte vergleichen. Der Erwartungswert für die Bearbeitungszeit eines Jobs mit Exponentialverteilung der Zeit ist $\frac{1}{\lambda}$. Wir spalten von der Summe den Job *0*, da dieser ja wieder bei beiden Politiken gleiche erwartete Idle-Zeit besitzt. Der Erwartungswert fuer D für Job *0* ist die Wahrscheinlichkeit, dass Job *0* der jetzte Job ist ($= p_0$) mal die erwartete Zeit die der Job über die anderen Jobs 'hinaushängt' unter der Bedingung, dass der Job tatsächlich der letzte Job ist:

$$E_\pi(D) = \sum_{j=1}^{n} \frac{1}{\lambda_j} p_j + p_0 E(X_0 - \sum_{i=1}^{n} X_i | X_0 > \sum_{i=1}^{n} X_i)$$

Das zweite Glied ist bei beiden Politiken π und $\bar{\pi}$ gleich, fällt also bei der Differenz weg:

$$E_\pi(D) - E_{\bar{\pi}}(D) = \sum_{j=1}^{n} \frac{1}{\lambda_j} (p_j - \bar{p}_j) \geq \frac{1}{\lambda_1} (p_1 - \bar{p}_1) + \frac{1}{\lambda_1} \sum_{j=2}^{n} (p_j - \bar{p}_j)$$

$$= 0$$

Wobei die Ungleichung aus $\lambda_1 = min_k(\lambda_k)$ (3.2) folgt und die letzte Gleichung $=0$ aus $\sum_{j=1}^{n} p_j = \sum_{j=1}^{n} \bar{p}_j$ folgt. Somit ist *Lemma 1* bewiesen, durch Vertauschen der zwei Jobs wurde eine bessere Politik $\bar{\pi}$ gefunden deren erwartete Idle-Zeit kleiner gleich der der alten Politik π ist.

3.3.6 Zweiter Beweisschritt: allgemeiner Fall

Korollar 1: *Lemma 1* gilt auch für beliebige Politiken der Form $\pi = (0, 1, i_2, i_3, \ldots, i_n)$ und $\bar{\pi} = (0, i_2, 1, i_3 \ldots, i_n)$ mit $\lambda_{i_2} \geq \lambda_1$.

Beweis: Da wir zum Beweis von *Lemma 1* nur $\lambda_j \geq \lambda_1$ benutzt haben und nicht direkt verwendet haben, dass $i_2 = 2$ gelten muss, gilt das *Korollar 1.*

Lemma 2: Seien zwei Politiken π und $\bar{\pi}$ gegeben:

$$\pi = (0, i_1, i_2, \ldots, i_{k-2}, i_k, i_{k-1} = 1, i_{k+1}, \ldots, i_n)$$

$$\bar{\pi} = (0, i_1, i_2, \ldots, i_{k-2}, i_{k-1} = 1, i_k, i_{k+1}, \ldots, i_n)$$

Gelte außerdem, dass $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, dann gilt:

$$E_{\bar{\pi}}(D) \leq E_{\pi}(D)$$

.

Beweis: Da die Jobs hier wiederum unabhängig voneinander sind, haben Job *0* bis i_{k-2} keinen Einfluss auf die Nachfolgenden, wie auch i_{k-1} und i_k keinen Einfluss auf die Zeiten der nachfolgenden Jobs haben. Somit können alle abgeschlossenen Jobs *0* bis i_{k-2} ignoriert und an der entsprechenden Stelle frisch angefangen werde. Dabei wird u.U. ein Job auf dem jeweils anderen Prozessor geschnitten. Dieser wird im Folgenden als Job $\bar{0}$ bezeichnet und besitzt aufgrund der Gedächtnislosigkeit genau die Verteilung des unterbrochenen Jobs. Dadurch ergeben sich zwei neue Politiken:

$$\pi = (\bar{0}, 1, i_k, i_{k+1}, \ldots, i_n), \qquad \bar{\pi} = (\bar{0}, i_k, 1, i_{k+1}, \ldots, i_n)$$

Nach *Korollar 1* folgt nun, da $\lambda_1 = min_{k \geq 2}(\lambda_k)$, direkt *Lemma 2.*

Korollar 2: *Lemma 2* gilt auch für beliebige Jobs j sofern alle $\lambda_{i_k} \geq \lambda_j$ mit $i_k > j$, der Job j also genau der Job mit der niedrigsten Nummer aller Jobs $\geq j$ ist.

Beweis: Durch Vertauschen mit Jobs niedriger Nummer können wir den Erwartungswert für D nicht verringern, da nach Voraussetzung alle λ_{i_k} kleiner sind als λ_j. Wir dürfen also wie bei *Lemma 2* die bisherigen Jobs verwerfen und den geschnittenen Job zu einem Job $\bar{0}$ wandeln. In dieser neuen äquivalenten Politik ist der Job j nun wieder insgesamt der Job mit dem kleinsten Wert λ_j, somit gilt wie *Korollar 1* auch *Korollar 2.*

Satz 1: Eine aufsteigend sortierte Politik $\pi = (0, 1, 2, \ldots, n)$ ist die optimale Politik um den Erwartungswert für D zu minimieren, falls $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ gilt.

Beweis: Durch mehrmaliges Anwenden von *Lemma 2* und *Korollar 2* (mit Job 2, Job 3 usw.) erreichen wir eine aufsteigende Reihenfolge, wobei bei jeder Iteration der Erwartungswert fuer die Zeit die ein Prozessor nichts zu tun hat ($=D$) weiter sinkt bzw. gleich bleibt. Ist die aufsteigende Reihenfolge erreicht gibt es keine Möglichkeit den Erwartungswert für D weiter zu senken. Somit ist dies das Optimum.

Dass dies auch das globale Optimum ist, es also nicht z.B. durch vertauschen von drei Jobs ein besseres Ergebnis erziehlt wird ist zwar intuitiv einsichtig, würde aber eine nochmalige Induktion über die Permutationen verlangen, auf die hier nicht weiter eingegangen wird.

3.4 Ein Prozessor mit begrenzter Zeit

In diesem Fall haben wir wie vorher auch n Jobs die ausgeführt werden sollen, es wird diesmal aber eine *feste Zeitbegrenzung* T vorgegeben bis zu der möglichst viele Jobs abgearbeitet werden sollen. Außerdem weisen wir jedem Job j eine Priorität R_j zu und bestimmen die Qualität einer Politik nicht wie vorher über die Zeit sondern über die Summe der Prioritäten bestimmen, es soll also

$$E(\text{Gewinn}) = \sum_{j=0}^{n} R_j E(s_j)$$

maximiert werden, wobei $s_j = 1$ ist, wenn der Job ausgeführt werden konnte und *0* wenn nicht. Auch hier gilt wieder die Unabhängigkeit der Ausführungszeiten der Jobs. Wenn also bereits $T - t_j$ Zeiteinheiten durch andere Jobs verstrichen sind, können wir den nächsten Job j als den ersten Job einer Politik zu dem Problem ansehen, bei der noch t_j Zeiteinheiten übrig sind. Das t_j ist natürlich abhängig von den früheren Jobs. Da wir uns für einen neuen Job jedoch immer erst entscheiden, wenn ein Prozessor frei ist, kennen wir zu diesem Zeitpunkt t_j und können es im weiteren als Konstante benutzen.

Wir schreiben nun die s_j als Wahrscheinlichkeiten aus um es mit dem Erwartungswert benutzen zu können. Die Wahrscheinlichkeit, dass Job j ausgeführt wird und noch t_j Zeiteinheiten übrig sind, beträgt wegen Exponentialverteilung:

$$s_j = \mathbb{P}(X_j < t_j) = 1 - e^{-\lambda_j t_j}$$

Somit ergibt sich für den erwarteten Gewinn für einen Job j und verbleibender (konstanter) Zeit t_j:

$$\begin{aligned} R_j \mathbb{P}(X_j < t_j) = R_j(1 - e^{-\lambda_j t_j}) = \lambda_j R_j \frac{1 - e^{-\lambda_j t_j}}{\lambda_j} \\ = \lambda_j R_j E[min(X_j, t_j)] \end{aligned}$$

Die letzte Gleichheit folgt aus:

$$\begin{aligned} E[min(X_j, t_j)] = \int_0^\infty P[min(X_j, t_j) > x] dx \\ = \int_0^{t_j} e^{-\lambda_j x} dx = \frac{(1 - e^{-\lambda_j t_j})}{\lambda_j} \end{aligned}$$

$min(X_j, t_j)$ entspricht dabei der Zeit, die der Prozessor an diesem Job j arbeitet, ob er abgebrochen wird oder nicht.

Da unser t_j von den vorherigen Jobs abhängt, können wir unser Ergebnis nun nicht einfach in die Gleichung für den Gesamtgewinn einsetzen. Ein formale Ausführung wäre sehr umfangreich und kompliziert. Um trotzdem auf ein Ergebnis zu kommen setzen wir deshalb hier nur dessen Bedeutung ein:

$$E_\pi(\text{Gesamtgewinn}) = \sum_{j=0}^{n} \lambda_j R_j E_\pi(\text{Zeit die Job } j \text{ bearbeitet wurde}) \quad (3.5)$$

Bleibt die Frage offen, was t_j in diesem Zusammenhang bedeutet. Zwar ist die Politik π eine stationäre Politik, die Reihenfolge der Jobs wird zu Beginn festgelegt, jedoch wann und auf welchen Prozessor die Jobs eingesetzt werden wird während der Abarbeitung der Politik festgelegt.

Zwar könnte man mit großem Aufwand den Erwartungswert von t_j über die Politik und die vor Job j stehenden X_j berechnen, um die optimale Politik zu finden und dann wie im vorherigen Kapitel zwei Politiken vergleichen und dann mit Induktion die Optimalität einer bestimmten Reihenfolge festzustellen, jedoch wir jedoch nicht den Erwartungswert an sich sondern es reicht zu wissen, wie sich dieser in Abhängigkeit von der Position des jeweiligen Jobs in der Politik verhält.

Da die Wahrscheinlichkeit, dass unser Job ausgeführt wird, sinkt je später wir ihn positionieren, folgt, dass eine absteigende Reihenfolge von $\lambda_j R_j$ optimal ist. λ_j als auch R_j sind vom Zeitpunkt wann der Job eingesetzt wird, also auch wieviel Zeit bis zum Ende übrigbleibt, unabhängig, $E_\pi(min(X_j, t_j))$ dagegen ist umso kleiner je kleiner das jeweilige t_j ist, also je später wir den Job einsetzen.

Also folgt: **Satz 2:** Der Gesamtgewinn bei gegebener Zeit T wird maximiert, wenn eine Politik gewählt wird, die die Jobs in einer absteigenden Folge von $\lambda_j R_j$ für $j = 1, \ldots, n$ sortiert.

3.5 Zwei Prozessoren mit begrenzter Zeit

Im letzten Teil kombinieren wir nun die vorherigen zwei Problemstellungen. Wir haben zwei Prozessoren, eine feste Zeit T und für jeden Job eine Priorität R_j. Der Gesamtgewinn ist wie im Fall mit einem Prozessor (3.5) definiert.

Es sieht nun so aus, als ob wir hier wie im vorherigen Kapitel, die optimale Politik eine absteigende Reihenfolge der $\lambda_j R_j$ ist, also wir Jobs um so weiter hinten positionieren je größer die erwartete benötigte Zeit und je kleiner die Priorität ist, also je weniger Gewinn pro Zeiteinheit wir machen.

Dies ist hier aber nicht unbedingt der Fall. Im zweiten Kapitel haben wir festgestellt, dass die optimale Reihenfolge eine der aufsteigenden λ_j ist, also lange Jobs möglichst zu Beginn gesetzt werden. Ein kleines Gegenbeispiel soll dies verdeutlichen:

$$\lambda_j R_j \equiv 1 \text{für alle } j \text{ und} \lambda_1 < \lambda_2 < \cdots < \lambda_n$$

.

Es würde folgen, dass, da ja alle $\lambda_j R_j$ gleich groß sind, jede beliebige Reihenfolge optimal ist. Dieses Ergebnis würde im Fall mit nur einem Prozessor auch stimmen, bei zwei Prozessoren tritt jedoch wie im zweiten Kapitel gezeigt eine immer größere Idle Zeit auf, wenn wir die Jobs absteigend nach den λ_j (bzw. aufsteigend nach den benötigten Zeiten) sortieren.

Man kann jedoch zeigen (worauf hier aber nicht weiter eingegangen werden kann), dass wie im Fall ohne Prioritäten eine aufsteigende Reihenfolge optimal ist, wenn wir gewisse Bedingungen voraussetzen. Dabei sollen die λ_j wie vorher auch aufsteigend geordnet sein und $\lambda j R_j$ absteigend, also lange Jobs mit hohen Gewinnen zu Beginn stehen sollen.

Satz 3:

$$\text{Sei } \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$$

$$\text{und } \lambda_1 R_1 \geq \lambda_2 R_2 \geq \cdots \geq \lambda_n R_n \geq 0$$

dann ist die Reihenfolge $(1, 2, \ldots, n)$ optimal und maximiert den zu erwartenden Gesamtgewinn für Zeit $t > 0$.

3.6 Ausblick und Zusammenfassung

Insgesamt hat dieser Vortrag zeigen können, wie man sich Schritt für Schritt an eine Problemstellung des Stochastic Scheduling herannähert und dann durch Benutzung der Voraussetzungen, hier vor allem der Gedächtnislosigkeit, der Unabhängigkeit der Jobs und der Induktion, die einzelnen Aussagen zu beweisen.

In den ersten zwei Kapiteln wurde neben einigen Definitionen vor allem auf die Herangehensweise wert gelegt, wie also eine Politik zu definieren ist. Außerdem wurde gezeigt, dass es für das jeweilige Problem unterschiedlich gut angepasste Politiken gibt. Hier wurde ein Vergleich der Politik 'Setze Jobs an bestimmten Zeiten' und der Politik 'Setze Jobs sobald der Prozessor frei wird' aufgestellt, der gezeigt hat, dass durch Einbringen von genauer Kenntnis des Problems die Politik wesentlich vereinfacht werden konnte, ohne dass der optimale Fall verloren geht.

Im ersten Kapitel wurde auf den Problemfall mit einem einzelnen Prozessor eingegangen. Mit korrekter Politikdefinition ließ sich das Problem trivial lösen, alle Politiken waren optimal, da $E(\sum_{j=0}^{n} X_{i_j})$ von der Anordnung der Jobs i_j unabhängig war.

Das Kapitel hat den Weg fuer das zweite Kapitel geebnet, bei dem auf den Fall mit zwei Prozessoren eingegangen wurde. Auch hier wurde die Politikdefinition selbst erst auf die Problemstellung definiert, in dem nicht für jeweils einen Prozessor eine seperate Politik definiert wurde, sondern durch intelligente Wahl der Einfügeregel wir uns wieder auf eine Politik beschränken konnten ohne dabei den optimalen Fall zu verlieren. Das Ergebnis hier war interessanter, eine Anordnung der Jobs nach absteigendem λ_j war optimal, was einerseits bildlich anhand von Balkengrafiken gezeigt wurde, und Schritt für Schritt über das Vertauschen einzelner Jobs bewiesen wurde.

Das dritte Kapitel ist wieder auf den Fall mit einem Prozessor zurückgekehrt und es wurde der Fall mit begrenzter Zeit und entsprechender Entlohnung bei Fertigstellung des entsprechenden Jobs betrachtet. Im Beweis für die Optimalität einer absteigenden Folge von $\lambda_j R_j$ konnten wir wieder die Unabhängigkeit ausnutzen und einzelne Jobs seperat betrachten. Über Umformung auf den erwarteten Gewinn eines jeden Jobs wurde kam schliesslich als Resultat heraus, dass der erwartete Gewinn davon abhängt, wieviele Zeiteinheiten der Job bearbeitet wurde.

Im vierten Kapitel haben wir zuerst versucht, das Ergebnis aus dem dritten Kapitel zu verwenden, ein Gegenbeispiel belehrte uns jedoch eines Besseren. Erst durch zusätzliche Forderungen an die λ_j konnten wir auf das Ergebnis kommen, dass wieder eine absteigende Folge von den Jobs, hier also durch die Forderung resultierende aufsteigende Folge von λ_j und absteigenden Folge von $\lambda_j R_j$, den Gewinn maximiert.

Man kann sich eine Vielzahl von weiteren Problemen der Art denken: Zwei Prozessoren hintereinander, zweimal zwei Prozessoren hintereinander parallelgeschaltet, Änderung der Prozessorgeschwindigkeit während des Verlaufs, Ausfall von Prozessoren usw. Legen wir mehr und mehr Nebenbedingungen hinzu wird es immer schwieriger eine optimale Politik zu beweisen. Leider sind wie immer die für die Praxis relevanten Beispiele nicht trivial. Ähnliche Anwendungen findet man in jedem Betrieb, bei der eine Menge von *Jobs* an eine Menge von *Prozessoren*, den Mitarbeitern, verteilt werden müssen. Hier kommen auch noch eine Reihe weiterer zusätzliche Bedingungen herein, wie z.B. das Auf- und Abrüsten der Maschinen (= Prozessoren) für bestimmte Art von Jobs, Lagerkosten von *Jobs*, usw. Was man in diesen Fällen machen kann, ist wie Anfangs erwähnt zumindest erst einmal eine möglichst gute Politikdefinition zu finden. Die eigentliche Optimierarbeit muss man dann aber zwangsläufig dem Computer mittels z.B. evolutionären Algorithmen überlassen, die den vorher definierten Raum von Politiken absuchen.

Hier ein Beispiel einer derartigen Software von SAP:

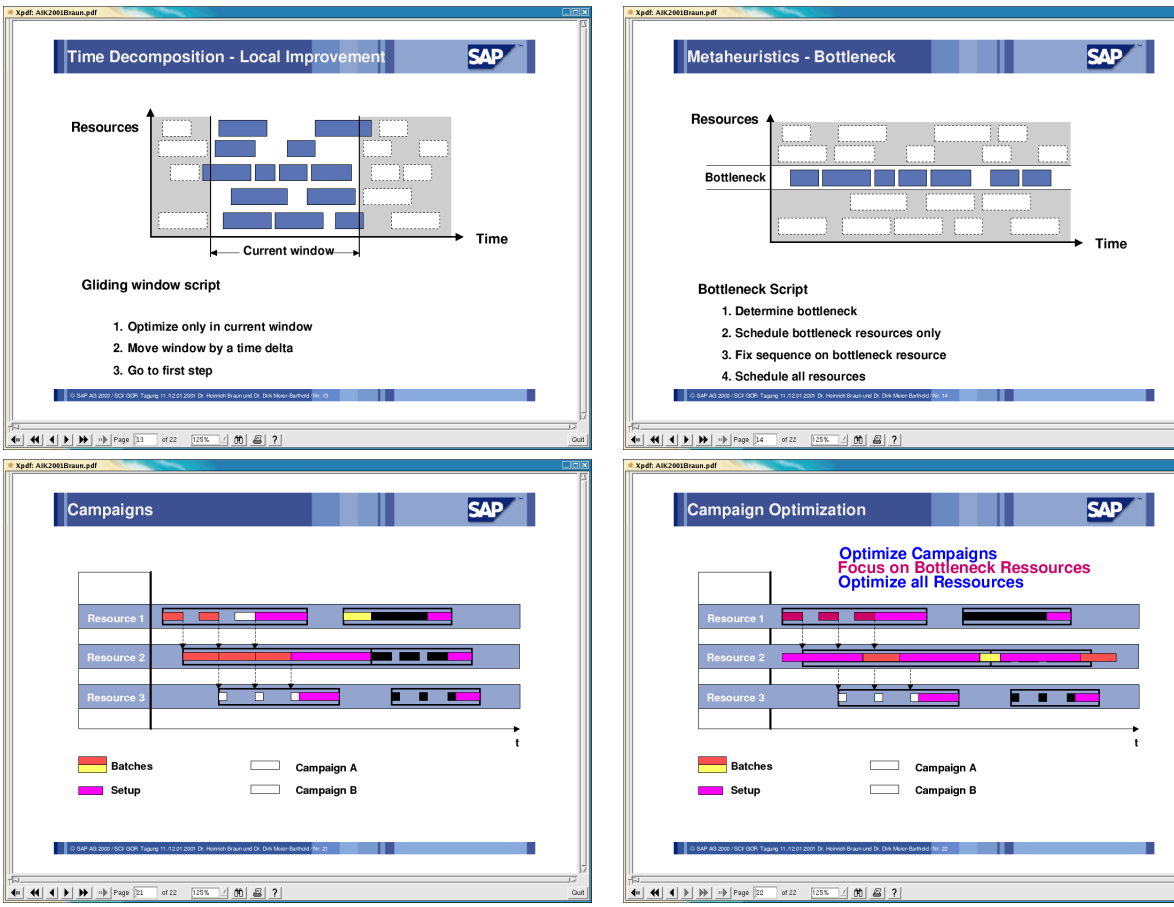


Abbildung 3.7: Beispiel Optimierer von SAP

Und einem Optimierer für Strategiespiele:

Abbildung 3.8: Beispiel Optimierer *Evolution Forge*

Literaturverzeichnis

[1] Sheldon M. Ross: *Introduction to stochastic dynamic programming - Probability and mathematical statistics*, First edition, Academic Press, Inc., 1983

[2] Dr. Heinrich Braun: *Evolutionäre Algorithmen im SAP Supply Chain Management*, http://www.aifb.uni-karlsruhe.de/AIK/aik_07/AIK2001Braun.pdf

[3] Hyun-Soo Ahn, Izak Duenyas, and Rachel Q. Zhang: *Optimal Stochastic Scheduling of a 2-Stage Tandem Queue with Parallel Servers*, November, 1997, http://www.ieor.berkeley.edu/ ahn/webstuff/aap.pdf, Seite 16-21