Approximationsalgorithmen

In polynomieller Zeit lässen sich nicht exakte Lösungen von *NP*-harten Problemen berechnen.

Approximationsalgorithmen versuchen das beste aus einer Polynomialzeitberechnung zu machen, indem sie wenigstens eine Näherunslösung bestimmen.

Im Gegensatz zu Heuristiken, verlangen wir aber von Approximationsalgorithmen eine Garantie, daß die berechnete Näherungslösung nah am Optimum ist.

Approximationsalgorithmen

Sei A ein Algorithmus, der in polynomieller Zeit eine Näherungslösung zu einem gegebenen Optimierungsproblem berechnet.

Sei $F^*(I)$ der optimale Wert der Zielfunktion und $F(I)$ der vom Algorithmus A erzielte Wert.

Definition

Das *Approximationsverhältnis* von A ist α, wenn

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \alpha \text{ und } \frac{F^*(I) - F(I)}{F^*(I)} \leq \alpha$$

gilt.

Approximationsalgorithmen

Diese Definition ist sowohl für Minimierungs- als auch für Maximierungsprobleme sinnvoll.

Für Minimierungsprobleme ist diese Ungleichung ausschlaggebend:

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \alpha$$

Für Maximierungsprobleme dagegen die andere:

$$\frac{F^*(I) - F(I)}{F^*(I)} \leq \alpha$$

Beispiel: Vertex Cover

Der Approximationsfaktor ist 2.

$C := \emptyset$;
while $E \neq \emptyset$ **do**
  choose some $e \in E$;
  $V := V - e$;
  $C := C \cup e$;
  $E := \{ e' \in E \mid e \cap e' = \emptyset \}$
od;
return C

Kompetitive Analyse

Um das Approximationsverhältnis zu analysieren, bietet sich eine *kompetitive Analyse* der folgenden Form an:

Wir gehen von einer Instanz und einer zugehörigen optimalen Lösung aus.

Wir müssen dann beweisen, daß der Approximationsalgorithmus eine Lösung findet, die nur um α schlechter als die vorgegebene optimale Lösung ist.

Kompetitive Analyse

Im Falle von Vertex Cover ist eine kompetitive Analyse recht einfach:

Sei $G = (V, E)$ ein Graph und $C \subseteq V$ ein minimales Vertex Cover.

Der Algorithmus wählt eine Kante $e = \{v_1, v_2\} \in E$ aus fügt v_1 und v_2 zu seinem VC hinzu. Einer von beiden Knoten ist aber auch in C. Zu einem Knoten in C, fügt der Algorithmus also höchstens 2 zu seiner Lösung hinzu.

Das macht ein Verhältnis von höchstens 2 zwischen konstruierter und optimaler Lösung aus.

Approximationsalgorithmen

Definition

1. Ein Approximationsalgorithmus ist ein *$f(n)$-Approximationsalgorithmus*, wenn das Approximationsverhältnis höchstens $f(n)$ für alle Instanzen der Länge n ist.
2. Ein *ϵ-Approximationsalgorithmus* ist ein $f(n)$-Approximationsalgorithmus mit $f(n) \leq \epsilon$ für alle $n \in \mathbf{N}$.

Approximationsschemata

Definition

Sei $A(\epsilon)$ ein Approximationsalgorithmus mit einem Parameter ϵ.

1. $A(\epsilon)$ ist ein *PTAS (polynomial time approximation scheme)*, falls $A(\epsilon)$ ein ϵ-Approxmiationsalgorithmus für jedes $\epsilon > 0$ ist und polynomielle Laufzeit in der Eingabelänge für jedes $\epsilon > 0$ hat.

2. $A(\epsilon)$ ist ein *FPTAS (fully polynomial time approximation scheme)*, falls $A(\epsilon)$ ein ϵ-Approxmiationsalgorithmus für jedes $\epsilon > 0$ ist und polynomielle Laufzeit in der Eingabelänge und $1/\epsilon$ für jedes $\epsilon > 0$ hat.

Scheduling

Wir betrachten folgendes Problem:

Gegeben sind n Aufgaben und m identische Prozessoren, sowie die Ausführungszeiten $t_1, \ldots, t_n$ der n Aufgaben.

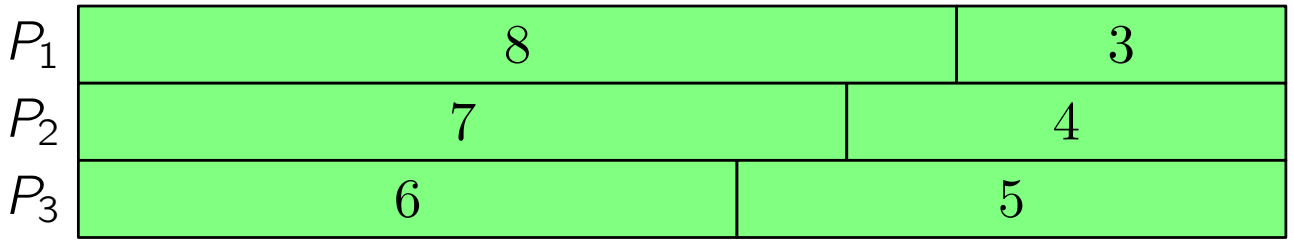
Gesucht ist eine Verteilung der Aufgaben auf die m Prozessoren, so daß die *Schlußzeit* minimiert wird, also die Zeit, bis alle Aufgaben erledigt wurden.

- Wir nennen dieses Problem im folgenden *Scheduling*.
- Für $m = 1$ ist es sehr leicht zu lösen.
- Ab $m = 2$ ist es NP-vollständig.

Beispiel

Sei $m = 3$, $n = 7$ und $(t_1, \ldots, t_6) = (8, 7, 6, 5, 4, 3)$.

Dieser Schedule ist optimal:



Die Optimalität folgt hier aus der Tatsache, daß alle Prozessoren gleichzeitig fertig werden.

Die LPT-Regel

Ein sehr einfaches Verfahren, in polynomieller Zeit einen Schedule zu erzeugen, nennt sich *LPT = longest processing time*.

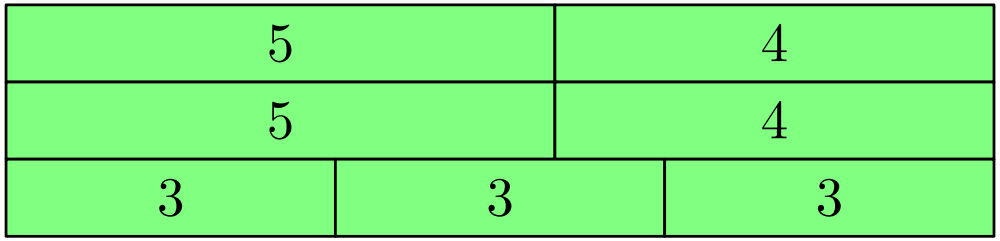
Definition

Die LPT-Regel weist dem nächsten freiwerdenden Prozessor die Aufgabe mit der größten Bearbeitungszeit zu. (Gibt es mehrere gleich große, dann nehmen wir irgendeine von ihnen.)

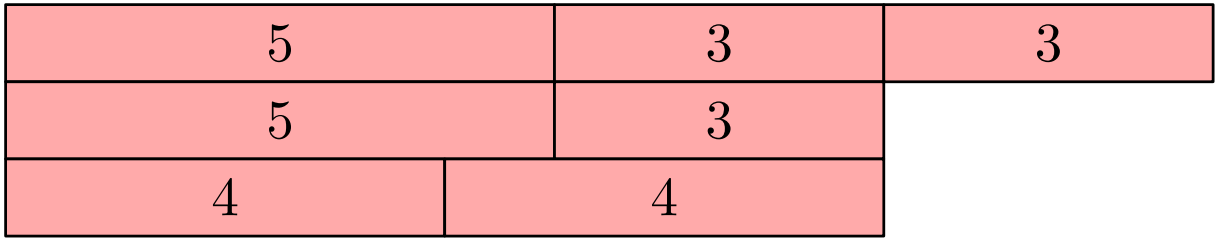
Beispiel

Sei $m = 3$, $n = 7$, $(t_1, \ldots, t_7) = (5, 5, 4, 4, 3, 3, 3)$.

Ein optimaler Plan ist offensichtlich:



Die LPT-Regel liefert dagegen:



Natürlich kann die LPT-Regel nicht immer optimale Pläne konstruieren.

Wie gut ist aber ein LPT-Schedule?

LPT-Schedules

Theorem

Sei I eine Instanz des m-Prozessorscheduling und $F^*(I)$ die optimale Schlußzeit. Sei $F(I)$ die Schlußzeit eines LPT-Schedules.

Dann gilt:

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \frac{1}{3} - \frac{1}{3m}$$

Insbesondere liefert die LPT-Regel einen 1/3-Approximationsalgorithmus.

Beweis

Für $m = 1$ ist ein LPT-Schedule optimal, also $F(I) = F^*(I)$. Sei nun $m > 1$.

Nehmen wir an, daß Theorem ist falsch. Dann gibt es eine Instanz $(t_1, \ldots, t_n)$, für die der LPT-Schedule schlechter ist, als vom Theorem behauptet.

Wir können annehmen, daß das Theorem für alle Instanzen mit weniger als n Aufgaben gilt und daß $t_1 \geq \ldots \geq t_n$. Die LPT-Regel soll die Aufgaben in der Reihenfolge ihrer Indizes zuordnen.

Sei $F(I)$ die Schlußzeit des LPT-Schedules.

Behauptung 1: Die Aufgabe mit Index n wird genau zur Schlußzeit, aber nicht früher beendet.

Wenn nicht, dann gibt es eine Aufgabe $k < n$, die später als Aufgabe n zur Schlußzeit beendet wird. Sei I' die Instanz, die aus den Aufgaben $1, \ldots, k$ besteht.

$$\frac{F(I') - F^*(I')}{F^*(I')} \geq \frac{F(I) - F^*(I)}{F^*(I)} > \frac{1}{3} - \frac{1}{3m}$$

Dies ist ein Widerspruch dazu, daß das Theorem für weniger als n Aufgaben gilt.

Behauptung 2: In einem optimalen Plan für I werden nicht mehr als zwei Aufgaben von einem Prozessor erledigt.

Wegen Behauptung 1 wird die Aufgabe n zum Zeitpunkt $F(I) - t_n$ im LPT-Schedule gestartet. Bis zu diesem Zeitpunkt müssen alle Prozessoren beschäftigt sein, denn sonst wäre Aufgabe n früher zugeordnet worden. Daraus folgt:

$$F(I) - t_n \leq \frac{1}{m} \sum_{n=1}^{n-1} t_i$$

$$F(I) \leq \frac{1}{m} \sum_{n=1}^{n} t_i + \frac{m-1}{m} t_n$$

Wir haben die untere Schranke $F^*(I) \geq \frac{1}{m} \sum_{i=1}^{n} t_i$.

Behauptung 2: In einem optimalen Plan für I werden nicht mehr als zwei Aufgaben von einem Prozessor erledigt.

Daraus folgt $F(I) - F^*(I) \leq \frac{m-1}{m} t_n$ oder

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \frac{m-1}{m} \frac{t_n}{F^*(I)}.$$

Da das Theorem für I nicht gilt, erhalten wir

$$\frac{1}{3} - \frac{1}{3m} < \frac{m-1}{m} \frac{t_n}{F^*(I)},$$

woraus $F^*(I) < 3t_n$ folgt.

Damit ist Behauptung 2 bewiesen.

Wir sind im Beweis des Theorems nun so weit:

Falls das Theorem für eine Instanz I versagt, dann werden in einem optimalen Plan für I jedem Prozessor höchstens zwei Aufgaben zugeordnet.

Dies passiert aber nie, denn es gilt:

Behauptung 3: Falls ein optimaler Plan für I jedem Prozessor höchstens zwei Aufgaben zuordnet, dann ist ein LPT-Schedule für I ebenfalls optimal.

Beweis von Behauptung 3: Übungsaufgabe!

□

Ein PTAS für Scheduling

Folgendes Verfahren führt zu einem PTAS:

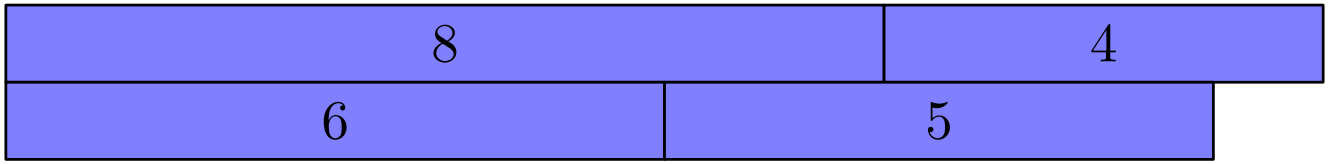
1. Wähle ein festes k (unabhängig von der Eingabe).
2. Berechne einen optimalen Plan für die k Aufgaben, mit der längsten Bearbeitungszeit.
3. Verplane die verbleibenden $n - k$ Aufgaben mit der LPT-Regel.

Warum sollte dieses Verfahren intuitiv gut sein?

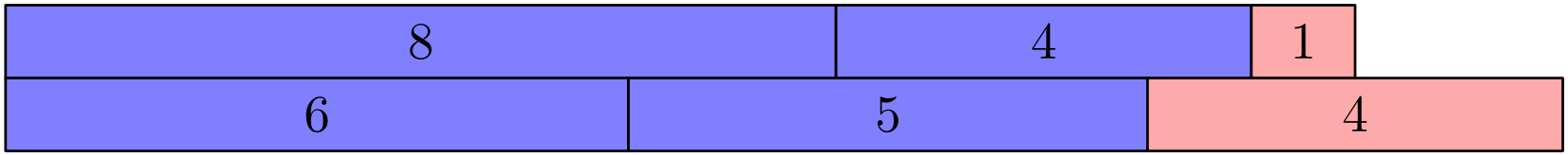
Beispiel

Sei $m = 2$, $n = 6$, $(t_1, \ldots, t_6) = (8, 6, 5, 4, 4, 1)$. Wir wählten $k = 4$.

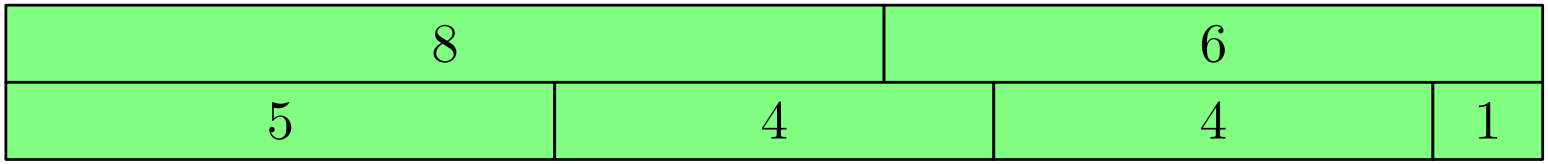
Schritt 1: Optimaler Plan für die 4 längsten Aufgaben.



Schritt 1: LPT-Plan für die verbleibenden 2 Aufgaben.



Die optimale Lösung ist:



Ein PTAS für Scheduling

Theorem

Sei $F(I)$ die Schlußzeit eines Plans, der durch das angebene Verfahrens erzeugt wird und $F^*(I)$ die optimale Schlußzeit von I.

Dann gilt

$$\frac{F(I) - F^*(I)}{F^*(I)} \leq \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}.$$

Beweis

- Sei r die optimale Schlußzeit für die k längsten Aufgaben.
- Wir können $F(I) > r$ annehmen, denn sonst $F(I) = F^*(I)$.
- Sei wieder o.B.d.A. $t_1 \geq \cdots \geq t_n$.
- Wir nehmen $n > k$ an (sonst $F(I) = F^*(I)$).
- Wir nehmen $n > m$ an (sonst $F(I) = F^*(I)$).

Wir wählen ein $j > k$, so daß die Bearbeitung der Aufgabe j zum Zeitpunkt $F(I)$ endet.

Falls es so ein j nicht gibt, dann gilt schon $F(I) = F^*(I)$.

Gibt es aber so ein j, dann müssen alle Prozessoren bis zum Zeitpunkt $F(I) - t_j$ beschäftigt sein.

Wegen $t_{k+1} \geq t_j$ sind insbesondere alle Prozessoren auch bis zum (früheren) Zeitpunkt $F(I) - t_{k+1}$ beschäftigt.

Daher gilt

$$\sum_{i=1}^{n} t_i \geq m(F(I) - t_{k+1}) + t_{k+1},$$

denn alle Prozessoren sind je $F(I) - t_{k+1}$ Schritte beschäftigt und einer muß noch Aufgabe t_{k+1} danach erledigen.

Beweis (Fortsetzung)

Aus

$$\sum_{i=1}^{n} t_i \geq m(F(I) - t_{k+1}) + t_{k+1},$$

folgt

$$F^*(I) \geq \frac{1}{m} \sum_{i=1}^{n} t_i \geq F(I) - \frac{m-1}{m} t_{k+1}$$

und schließlich

$$F(I) - F^*(I) \leq \frac{m-1}{m} t_{k+1}. \qquad (*)$$

Um $\dfrac{F(I) - F^*(I)}{F^*(I)}$ abschätzen zu können, brauchen wir noch eine (gute) untere Schranke für $F^*(I)$.

Beweis (Fortsetzung)

Wir betrachten die ersten $k+1$ Aufgaben. Es gibt nur m Prozessoren.

Also muß es einen Prozessor geben, der mindestens $\lceil (k+1)/m \rceil = 1 + \lfloor k/m \rfloor$ von diesen $k+1$ Aufgaben allein erledigt.

Wegen $t_1 \geq \ldots \geq t_n$ dauert es mindesten t_{k+1} Schritte, jede dieser Aufgaben zu erledigen.

Es gibt also einen Prozessor, der $(1 + \lfloor k/m \rfloor)t_{k+1}$ Schritte lang beschäftigt ist.

$$F^*(I) \geq (1 + \lfloor k/m \rfloor)t_{k+1} \qquad (**)$$

Beweis (Fortsetzung)

$$F(I) - F^*(I) \leq \frac{m-1}{m} t_{k+1}. \qquad (*)$$

$$F^*(I) \geq (1 + \lfloor k/m \rfloor) t_{k+1} \qquad (**)$$

Kombinieren wir $(*)$ und $(**)$ erhalten wir

$$\frac{F(I) - F^*(I)}{F^*(I)} \geq \frac{(m-1)/m}{1 + \lfloor k/m \rfloor} = \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}.$$

□

Ein PTAS für Scheduling

Um ein PTAS zu erhalten, müssen wir bedenken, daß ϵ Teil der Eingabe ist.

Daraus muß der Algorithmus ein geeignetes k berechnen, mit

$$\frac{1 - 1/m}{1 + \lfloor k/m \rfloor} \leq \epsilon.$$

Das funktioniert beispielsweise, wenn wir ein k wählen, daß grösser als $(m-1)/\epsilon - m$ ist.

Wie groß ist die Laufzeit? Sie ist $O(n \log n + m^k)$, wenn wir den exakten Teil mit einem Branch-and-Bound-Algorithmus lösen.

Das entspricht $O(n \log n + m^{(m-1)/\epsilon - m})$.

Für jedes feste m haben wir ein PTAS.

Frage: Was passiert wenn m nicht fest ist?

Entwurf eines FPTAS

Wir betrachten Optimierungsprobleme der Form

$$\text{maximiere } \sum_{i=1}^{n} p_i x_i$$

$$\text{unter } \sum_{i=1}^{n} a_{ij} x_i \leq b_j \text{ für } j = 1, \ldots, m$$

$$x_i \in \{0, 1\}$$

$$p_i \geq 0$$

$$a_{ij} \geq 0$$

Ohne Beschränkung der Allgemeinheit können wir noch $a_{ij} \leq b_j$ für alle i und j annehmen, denn sonst muß ja $x_i = 0$ gelten.

Entwurf eines FPTAS

Definition

Für $1 \leq k \leq n$ nennen wir $x_i = y_i$, $i = 1, \ldots, k$ eine *zulässige Einschränkung*, wenn es wenigstens eine zulässige Lösung gibt, die mit $x_i = y_i$ für $i = 1, \ldots, k$ anfängt.

Wir nennen dann so eine zulässige Lösung eine *Vervollständigung* dieser zulässigen Einschränkung.

Seien nun $x_i = y_i$ und $x_i = z_i$ für $i = 1, \ldots, k$ zwei verschiedene zulässige Einschränkungen mit

$$\sum_{i=1}^{k} p_i y_i = \sum_{i=1}^{k} p_i z_i.$$

Wir sagen, daß $x_i = y_i$ die $x_i = z_i$ *dominiert*, falls es eine Vervollständigung von $x_i = y_i$ gibt, deren Wert der Zielfunktion mindestens so groß ist, wie die Zielfunktionswerte aller Vervollständigungen von $x_i = z_i$.

Wir betrachten Probleme, für welche es eine einfache Regel gibt, die Dominanz von zulässigen Einschränkungen entscheidet.

Beispiel

Knapsack: Gegeben n Gegenstände und zu jedem Gegenstand sein Wert und seine Größe. Ausserdem ist die Größe eines Rucksacks gegeben.

Ziel ist es, Gegenstände auszusuchen, die in den Rucksack gepackt werden sollen. Ihr Wert soll maximal sein, aber sie müssen alle hineinpassen.

Dieses Problem läßt sich leicht auf die gewünschte Art modellieren. Die Werte der Gegenstände sind die p_i und die a_{i1} sind ihre Größe.

Dominanz ist leicht zu entscheiden. Wie?

Entwurf eines FPTAS

Allgemein lassen sich diese Optimierungsprobleme exakt durch dynamisches Programmieren lösen:

Wenn bereits zulässige Einschränkungen S^i für $x_1, \ldots, x_i$ gegeben sind, können wir daraus alle zulässigen Einschränkungen für $x_1, \ldots, x_{i+1}$ berechnen. Aus dieser Menge können wir noch alle dominierten zulässigen Einschränkungen entfernen und erhalten dann S^{i+1}. Wir können mit dem leeren S^0 beginnen und eine optimale Lösung aus S^n entnehmen.

Problem: Von S^i nach S^{i+1} kann sich die Größe verdoppeln.

Wir betrachten verschiedene Methoden, die Größe von S^i polynomiell zu halten.

Runden

Da wir Dominanz verwenden, enthält S_i keine zwei zulässigen Einschränkungen mit gleichem Wert der Zielfunktion.

Gibt es also nur wenige mögliche, verschiedene Werte der Zielfunktion, dann bleibt S^i klein.

Eine grobe Idee besteht darin, eine Instanz I durch eine andere I' zu ersetzen, die sich nur durch die Koeffizienten p_i der Zielfunktion unterscheiden. Die neuen Koeffizienten q_i von I' sollen nur wenige verschiedene Werte der Zielfunktion erlauben.

Dabei haben I und I' immer noch dieselben zulässigen Lösungen. Ihre optimalen Lösungen können aber verschieden sein.

Wir müssen dafür sorgen, daß sie sich nicht zu sehr unterscheiden.

Beispiel

Sei $(p_1, \ldots, p_4) = (1.1, 2.1, 1001.6, 1002.3)$.

Die Zielfunktion nimmt folgende Werte in S^i an:

S^1: $\{0, 1.1\}$

S^2: $\{0, 1.1, 2.1, 3.2\}$

S^3: $\{0, 1.1, 2.1, 3.2, 1001.6, 1002.7, 1003.7, 1004.8\}$

S^4: $\{0, 1.1, 2.1, 3.2, 1001.6, 1002.3, 1002.7, 1003.4,$
$1003.7, 1004.4, 1004.8, 1005.5, 2003.9, 2005, 2006, 2007.1\}$

Insgesamt werden 31 zulässige Einschränkungen betrachtet. Ihre Zahl verdoppelt sich jeweils, da keine zwei vorkommen, für die die Zielfunktion übereinstimmt.

Wir ersetzen jetzt $(p_1, \ldots, p_4) = (1.1, 2.1, 1001.6, 1002.3)$ durch $(q_1, \ldots, q_4) = (0, 0, 1000, 1000)$.

Jetzt erhalten wir nur:

$$S^1 : \{0\}$$
$$S^2 : \{0\}$$
$$S^3 : \{0, 1000\}$$
$$S^4 : \{0, 1000, 2000\}$$

Der Wert der neuen Zielfunktion kann um höchstens 7.1 zu klein sein und der optimale Wert ist mindestens $\max\{p_i\} = 1002.3$. Das heißt $F^*(I) - F^*(I') \leq 7.1$ und $F^*(I) \geq 1002.3$. Wir erhalten

$$\frac{F^*(I) - F^*(I')}{F^*(I)} \leq 0.007.$$

Runden

Gegeben ein ϵ, müssen wir $q_1, \ldots, q_n$ so bestimmen, daß

$$\frac{F^*(I) - F^*(I')}{F^*(I)} \leq \epsilon$$

und

$$\sum_{i=1}^{n} |S^i| \leq u(n, 1/\epsilon)$$

gilt, wobei u ein Polynom ist.

Falls wir S^{i+1} aus S^i in Zeit polynomiell in $|S^i|$ berechnen können, dann haben wir ein FPTAS.

Runden

$$\frac{F^*(I) - F^*(I')}{F^*(I)} \leq \epsilon$$

können wir erzielen, durch

$$\sum_{i=1}^{n} |p_i - q_i| \leq \epsilon F^*(I). \qquad (*)$$

Wir definieren

$$q_i = \lfloor p_i/(\epsilon L/n) \rfloor \epsilon L/n.$$

Alle q_i sind dadurch ein Vielfaches von $\epsilon L/n$ und unterscheiden sich von p_i um höchstens $\epsilon L/n$. Das erfüllt $(*)$.

L ist eine geeignete untere Schranke für $F^*(I)$. Wir können immer $L := \max\{p_i\}$ wählen.

Runden

$$q_i = \lfloor p_i/(\epsilon L/n) \rfloor \epsilon L/n.$$

Alle q_i sind ein Vielfaches von $\epsilon L/n$.

Wie groß kann $|S^i|$ werden?

Wegen $p_j \leq L$, ist der Wert der Zielfunktion jeder zulässigen Einschränkung in S^i höchstens $i \cdot L$. Die Zielfunktionswerte sind Vielfache von $\epsilon L/n$.

Es gibt also höchstens $\left\lceil iL \Big/ \epsilon L/n \right\rceil = O(i \cdot n/\epsilon)$ verschiedene Elemente in S^i.

(Gleiche Zielfunktionswerte kommen nicht vor, wegen Dominanz.)

Runden

Wir erhalten

$$\sum_{i=1}^{n} |S^i| = O(n^3/\epsilon).$$

Das ergibt ein FPTAS in $O(n^3/\epsilon)$ Zeit unter folgenden Voraussetzungen:

- Dominanz in linearer Zeit entscheidbar
- S^{i+1} aus S^i in $O(|S^i|)$ Schritten berechenbar
- Eine untere Schranke $L \geq \max\{p_i\}$
- Zum allgemeinen Schema passend

Beispiel

Wir haben einen Rucksack der Kapazität 1112 und Gegenstände deren Größe und Wert $1, 2, 10, 100, 1000$ ist. Wir wählen $\epsilon = 0.1$.

$L = \max\{p_i\} = 1000$. Wir runden auf Vielfache von $\epsilon L/n = 0.1 \cdot 1000/5 = 20$. Die q_i sind daher $0, 0, 0, 100, 1000$.

Wir erhalten
$S^5 = \{(0,0,0,0,0), (0,0,0,1,0), (0,0,0,0,1), (0,0,0,1,1)\}$.

Die optimale Lösung von I' ist $(0,0,0,1,1)$ mit Zielwert 1100.

Die eigentliche optimale Lösung ist 1112, da wir den Rucksack ganz füllen können.

Intervall-Partitionierung

Intervall-Partitionierung ist eine weitere Methode, um die Mengen S^i klein zu halten.

Sei P_i das Maximum von allen $\sum_{j=1}^{i} p_j x_j$, unter allen zulässigen Einschränkungen in S^i.

- Wir unterteilen das Intervall $[0, P_i]$ in Unterintervalle der Größe $\epsilon P_i/(n-1)$ (wobei das letzte kleiner sein kann).
- Von allen zulässigen Einschränkungen, deren $\sum_{j=1}^{i} p_j x_j$ in das gleiche Unterintervall fallen, eliminieren wir alle bis auf eine mit Dominanzregeln. Dabei tun wir so, als wären ihre Zielfunktionen gleich.
- Das S^{i+1} wird dann aus dem reduzierten S^i berechnet.

Intervall-Partitionierung

- Der Fehler, der beim Identifizieren aller zulässigen Einschränkungen in einem Unterintervall gemacht werden kann, ist höchstens $\epsilon P_i/(n-1)$.
- $P_i \leq F^*(I)$, denn P_i ist kleiner oder gleich der Zielfunktion einer zulässigen Lösung.
- Fehler können sich fortpflanzen. Bei jedem Übergang $S^i \to S^{i+1}$ kann ein neuer Fehler hinzukommen.
- Der Gesamtfehler ist höchstens

$$F^*(I) - F(I) = \sum_{i=1}^{n-1} \frac{\epsilon P_i}{n-1} \leq \epsilon F^*(I).$$

Intervall-Partitionierung

Die Anzahl der Unterintervalle ist höchstens $\lceil n/\epsilon \rceil + 1$.

Also ist $|S_i| = O(n/\epsilon)$ und

$$\sum_{i=1}^{n} |S_i| = O(n^2/\epsilon).$$

Unter guten Voraussetzungen ergibt dies einen $O(n^2/\epsilon)$-Zeit-Algorithmus.

Ist eine bessere untere Schranke für $F^*(I)$ bekannt, so kann diese jederzeit statt P_i verwendet werden.

Beispiel

Wir haben wieder einen Rucksack der Kapazität 1112 und Gegenstände deren Größe und Wert $1, 2, 10, 100, 1000$ ist. Wir wählen $\epsilon = 0.1$.

$P_1 = 1$ und die Unterintervalle haben Länge $0.1/4 = 0.025$. Das ergibt $S^1 = \{(0), (1)\}$.

Daraus wird zunächst $S^2 = \{(0,0), (0,1), (1,0), (1,1)\}$ berechnet.

Die zugehörigen Werte der Zielfunktion sind $0, 2, 1, 3$. Das ergibt $P_2 = 3$ und Unterintervalle der Länge $0.1 \cdot 3/4 = 0.075$.

Die Intervalle sind $[0, 0.075), [0.075, 0.15), [0.225, 0.3), \ldots$. Es findet keine Reduzierung statt, da nur ein Wert in jedes Unterintervall fällt.

Als nächstes wird $S^3 = \{(0,0,0),(0,0,1),\ldots,(1,1,1)\}$ bestimmt. Die Werte sind jetzt $0, 10, 2, 12, 1, 11, 3, 13$.

Die Unterintervalle haben Länge $0.1 \cdot 13/4 = 0.325$. Wieder findet keine Reduktion statt.

Beim nächsten Schritt haben wir eine Länge von $0.1 \cdot 113/4 = 2.825$. Jetzt fallen beispielsweise $0, 1, 2$ in das gleiche Intervall.

Hätten wir die Gegenstände in umgekehrter Reihenfolge betrachtet, wäre es einfacher gewesen.

Wir können aber auch eine eigene untere Schranke L statt der P_i verwenden.

$L = \max\{p_i\} = 1000$. Die erste Intervalllänge ist daher bereits $0.1 \cdot 1000/4 = 25$.

Jetzt sind die Werte der Zielfunktion für S^1, S^2 und S^3 nur 0, für S^4 sind es 0 und 100 und für S^5 schließlich 0, 100, 1000 und 1100.

Als beste Lösung wird $(0, 0, 0, 1, 1)$ bestimmt.

Separation

Die Intervall-Partitionierung hat $[0, P_i]$ stur in Unterintervalle geteilt. Nur in gleich Unterintervalle fallende zulässige Einschränkungen werden zusammengefasst.

Separation geht ähnlich vor, teilt aber nicht in Unterintervalle ein.

Stattdessen werden zulässige Einschränkungen als gleichwertig angesehen, wenn sich ihre Werte um höchstens $\epsilon P_i/(n-1)$ unterscheiden.

Auch bei diesem Vorgehen kann der hinzugefügte Fehler nur $\epsilon P_i/(n-1)$ sein.

Separation

Sind die Werte $3.9, 4.1, 7.9, 8.1, 11.9, 12.1$ dann führt eine Intervalllänge von 2 zu keiner Reduktion.

Separation halbiert allerdings die Anzahl der Werte.

Intervall-Partitionierung kann aber ebenfalls besser sein:

Wir betrachten hierzu eine Knapsack-Instanz mit $(p_1, \ldots, p_5) = (w_1, \ldots, w_5) = (3, 1, 5.1, 5.1, 5.1)$. Es sei eine untere Schranke bekannt, so daß

$$\epsilon L/(n-1) = 2.$$

Was ergibt sich nun bei Intervall-Partitionierung und was bei Separation?

Nichtapproximierbare Probleme

Wir betrachten das folgende Optimierungsproblem:

$$\text{Maximiere } \sum_{i=1}^{n} a_i x_i$$

$$\text{unter } \sum_{i=1}^{n} a_i x_i \leq m$$

$$x_1, \ldots, x_n \in \{0, 1\}$$

$$\text{mit } m = \frac{1}{2} \sum_{i=1}^{n} a_i$$

Die optimale Lösung ist m, falls sich $\{a_1, \ldots, a_n\}$ in zwei Mengen mit gleicher Summe partitionieren läßt.

Dies ist ein NP-hartes Problem.

Nichtapproximierbare Probleme

Betrachte nun folgende Variante:

$$\text{Minimiere } 1 + k\left(m - \sum_{i=1}^{n} a_i x_i\right) \text{ unter } \sum_{i=1}^{n} a_i x_i \leq m$$

$$x_1, \ldots, x_n \in \{0, 1\}$$

Die optimale Lösung ist jetzt 1, falls sich $\{a_1, \ldots, a_n\}$ in zwei Mengen mit gleicher Summe partitionieren läßt.

Andernfalls ist sie mindestens $1 + k$.

Angenommen es gäbe einen ϵ-Approximationsalgorithmus.

Wähle $k > \epsilon$. Dann löst der Approximationsalgorithmus das Problem exakt, falls die optimale Lösung m ist.

Dieses Problem ist für kein $\epsilon > 0$ approximierbar, solange $P \neq NP$.