Rheinisch-Westfälische Technische Hochschule
Institut für Geometrie und Praktische Mathematik

Mathematisches Praktikum (MaPra) — Sommersemester 2011

Prof. Dr. Wolfgang Dahmen — M.Sc. Mathieu Bachmann, Dipl.-Math. Jens Berger, M.Sc. Liang Zhang

Aufgabe 7

Bearbeitungszeit: zwei Wochen (bis Montag, den 11. Juli 2011)

Testattermin: Donnerstag, der 14. Juli 2011

Mathematischer Hintergrund: Optimierungsverfahren, Rucksackproblem, Tabu Search

Elemente von C++: Streams, Templates

25 Punkte

Aufgabenstellung

In vielen Bereichen der Wirtschaft, beispielsweise in der Produktions- und Projektplanung, beim VLSI-Design oder in der Transport- und Tourenplanung, haben wir es mit Problemstellungen zu tun, die sich bei näherer Untersuchung als *kombinatorische Optimierungsprobleme* darstellen. Oft ist es möglich, die gestellte Aufgabe mathematisch als ganzzahliges, lineares Optimierungsproblem zu formulieren. [1]
Ein bekanntes Beispiel ist das Problem des Handelsreisenden (traveling salesman problem, TSP), der in einer Rundreise eine vorgegebene Anzahl von Städten bereisen soll. Bei n Städten ergeben sich $\frac{1}{2}(n-1)!$[1] mögliche Wege. Schon bei vergleichsweise wenig Städten, z.B. $n = 15$, erhält man die enorme Zahl von 43.589.145.600 Möglichkeiten.[2]
Die Ermittelung eines Optimums in akzeptabler Zeit ist aufgrund dieser großen Anzahl von Zuständen im allgemeinen nicht mehr durchführbar. [4] Eine Möglichkeit solche Probleme dennoch in der Praxis anzugehen ist, nicht unbedingt auf einer optimalen Lösung zu bestehen, sondern sich mit einer möglichst guten Lösung zufrieden zu geben, die insbesondere schnell berechnet werden kann. Zu diesem Zweck existieren verschiedene Verfahren [6] wie beispielsweise das zufallsgesteuerte *Simulated Annealing* oder eben das in dieser Aufgabe zu implementierende *Tabu Search* (TS) [3, 2]. Für ersteres existieren Konvergenzaussagen, worauf jedoch hier nicht weiter eingegangen werden soll, bei TS hingegen ist sehr wenig in dieser Richtung bekannt.
Ihre Aufgabe wird es sein, einen Tabu Search Algorithmus zur Lösung des Rucksackproblems (knapsack problem) zu implementieren. Getestet wird der Algorithmus an vorgegebenen Testbeispielen, wobei die Parameter des Verfahrens frei einstellbar sein sollen.

Das Rucksackproblem

Das Rucksackproblem [5] besteht darin, einen Rucksack mit einem maximalen Fassungsvermögen (maximale Gesamtmasse) mit verschiedenen Gegenständen, jeweils versehen mit Preis und Masse, so zu füllen, daß die Füllung in einem gewissen Sinne optimal ist. Optimal bedeutet hier, daß die Gesamtmasse der eingepackten Gegenstände nicht die maximale Gesamtmasse überschreitet und der Gesamtwert der eingepackten Gegenstände maximal ist. Dabei gilt in der hier betrachteten speziellen Variante, daß jeder Gegenstand nur einmal zum Einpacken zur Verfügung steht („0/1“-knapsack problem). Häufig wird das Problem anhand eines Schmugglers geschildert, der zollfrei Gegenstände über eine Grenze bringen möchte, um diese dann dort zu verkaufen ... aber das soll hier natürlich *nicht* als Beispiel aufgeführt werden.

[1] Die Rundreise links- und rechtsherum zählt nur einfach, daher der Faktor $\frac{1}{2}$.
[2] Nur zum Vergleich: auf einem System mit einem mit 3400 MHz getaktetem AMD Phenom II-Prozessor dauert eine leere (!) Schleife von 0 bis 43.589.145.600 bereits ca. 70 Sekunden.

Es gibt keinen bekannten Algorithmus, der das Rucksackproblem löst und dessen Komplexität durch ein Polynom beschränkt ist. Das zugehörige Entscheidungsproblem ist *NP-vollständig*, es existieren sogar Ansätze, basierend auf dem Rucksackproblem ein Public-Key Kryptosystem zu entwerfen.

Beispiel Folgende Menge von Gegenständen soll in obigem Sinne optimal eingepackt werden. Der Rucksack kann maximal 20 kg verkraften.

Gegenstand Nr.	1	2	3	4	5	6
Masse [kg]	6	7	5	8	5	3
Gewinn [€]	9	10	14	11	8	8

Ein erster, naiver Ansatz besteht darin, die Gegenstände mit dem größten Gewinn einzupacken, also Nr. 2, 3 und 4, kurz (234). Das ergibt eine Gesamtmasse von genau 20 kg und einen Gewinn von 35 €, dies ist aber leider nicht das Optimum. Der geneigte Leser kann an dieser Stelle schonmal darüber nachdenken, welche Strategien möglich sind, eine gute Lösung zu ermitteln; natürlich *ohne* alle Möglichkeiten durchzugehen.[3]

Globale und lokale Optima

Ausgehend von obiger Startsituation (234) muß man zunächst etwas auspacken, bevor man andere Gegenstände einpacken kann, da die maximale Gesamtmasse erreicht ist. Eine Strategie könnte darin bestehen, zunächst den Gegenstand auszupacken, der am wenigsten zum Gewinn beiträgt, also hier 2. Nun macht es wenig Sinn genau diesen wieder einzupacken, man lässt daher 2 erstmal in Ruhe und nimmt stattdessen den nächstbesten, also Gegenstand 1. Die Kombination (134) ergibt eine Masse von 19 kg und einen Gewinn von 34 €, ist also nicht besser. Um weitere Gegenstände zu testen muß man nun wieder auspacken. Analog zu vorher lässt man das zuletzt eingepackte in Ruhe, d.h. 1 bleibt, und das nächstschlechteste geht, hier 4. So fährt man fort und es ergibt sich nachfolgender Ablauf. Dabei bezeichnet $+n$ das Einpacken und $-n$ das Auspacken eines Gegenstandes n.

k	$x^{(k)}$	Aktion	$x^{(k+1)}$	Gewinn [€]	Gesamtmassen [kg]
0	234	-2	34	25	13
1	34	+1	134	34	19
2	134	-4	13	23	11
3	13	+2	123	33	18
4	123	-1	23	24	12
5	23	+4	234	35	20

Nach sechs Aktionen landet man offenbar wieder bei (234), man bewegt sich im Kreis. Mit obiger Strategie ist dies das *lokale Optimum* zum Startwert (234). Dabei wurde jedoch das *globale Optimum*, nämlich (2356), nur knapp verfehlt. Packt man in Schritt 5 statt Gegenstand 4 die Nr. 6 ein (gleicher Gewinn wie 5, aber leichter), so kann man auch noch 5 hinzunehmen und erreicht (2356) mit dem Gewinn von 40 €. Es scheint also nicht bei jedem Vorgang sinnvoll, die jeweils günstigste Alternative zu wählen. Man erreicht zwar die beste unmittelbar benachbarte Lösung, verfehlt dadurch aber unter Umständen eine bessere und gerät stattdessen ins Kreisen.
Tabu Search verfolgt nun die Idee, den besten Zug auszuführen, der nicht zu einer bereits betrachteten Lösung zurückführt.[4]

[3] Zum Beispiel könnte man die Gegenstänge nicht nach ihrem Wert, sondern nach dem Wert bezogen auf ihre Masse beurteilen. Aber auch dieses Vorgehen führt in obigem Beispiel nicht sofort zur optimalen Lösung.

[4] Eine andere Strategie verfolgt beispielsweise Simulated Annealing (=ausglühen). Hier wählt man nicht immer den besten Zug, sondern akzeptiert auch schlechtere Lösungen mit einer gewissen Wahrscheinlichkeit. Diese Wahrscheinlichkeit wird dann im Laufe der Iteration immer kleiner.

Tabu Search

Die Grundidee ist oben bereits beschrieben: Man lässt Gegenstände für eine gewisse Zeit in Ruhe, sie sind sozusagen für die *Tabudauer* (TD) tabu. Dazu führt man eine *Tabuliste* (TL) ein, in der die komplementären, verbotenen Aktionen gesammelt werden. Bei der Auswahl der ein- oder auszupackenden Gegenstände spielen dann alle in der TL aufgeführten Gegenstände keine Rolle, solange bis die jeweilige TD abgelaufen ist. Das Beispiel von oben entspricht einer TD von 1.
Zur Beschreibung des Algorithmus sind folgende zwei Funktionen hilfreich: `add` und `clear`. Die Funktion `add` sucht zunächst unter den verfügbaren Gegenständen, die nicht in der aktuellen Lösung sind, die nicht tabu sind und die noch in den Rucksack passen, den besten aus. Falls ein solcher existiert, wird er in die aktuelle Lösung x eingefügt und die Funktion wird mit `true` verlassen. Existiert kein solcher Gegenstand, wird `false` zurückgegeben. `clear` sucht unter den Gegenständen in der aktuellen Lösung x, die nicht tabu sind, den schlechtesten aus. Dieser wird dann aus x entfernt.
Der Algorithmus für TS sieht dann wie folgt aus:

```
while (add())          // Erzeugung eines Startvektors
loop                   // loop je nach Abbruchkriterium
   if (!add())         // ergaenze Gegenstand
      clear()          // kein Platz mehr, also loeschen
   else
      merke Loesung, falls sie besser als die letzte beste
         Loesung ist
```

Sind unter den zuzufügenden Gegenständen mehrere mit gleichem Gewinn, nimmt man natürlich den Leichtesten. Analog wählt man beim Entfernen bei Gleichheit den Schwersten. Bewegt man einen Gegenstand in oder aus der Lösung x, so muß die TL aktualisiert werden. Weiterhin muß für die Gegenstände in der TL nachgehalten werden, ob sie noch tabu sind. Zur Illustration sei obiges Beispiel nun mit TD = 3 gerechnet:

k	$x^{(k)}$	Aktion	TL	$x^{(k+1)}$	Gewinn [€]	Masse [kg]	Bemerkung
0	234	-2	+2	34	25	13	
1	34	+1	+2,-1	134	34	19	
2	134	-4	+2,-1,+4	13	23	11	
3	13	+6	-1,+4,-6	136	31	14	erstmalig: +2 wird aus TL entfernt
4	136	+5	+4,-6,-5	1356	39	19	erstmalig: zwei Gegenstände hinzugefügt
5	1356	-1	-6,-5,+1	356	30	13	
6	356	+2	-5,+1,-2	2356	40	20	globales Optimum $\rightsquigarrow$ Merken
7	2356	-6	+1,-2,+6	235	32	17	
8	235	-5	-2,+6,+5	23	24	12	erstmalig: zwei Gegenstände entfernt
9	23	+4	+6,+5,-4	234	35	20	Startwert

An dieser Stelle erreicht man zwar wieder (234), jedoch sieht die TL anders aus. Daher gerät man hier noch nicht ins Kreisen. Der `loop` besteht aus einer vorgegebenen Anzahl von Iterationen.
Der beschriebene Algorithmus ist an vielen Stellen verbesserbar, die zugrundeliegende Idee bleibt aber erhalten. Erweiterungen können sein, den Tabustatus einzelner Elemente temporär aufzuheben (Aspirationskriterien) oder die TD dynamisch zu gestalten. Weitere Informationen dazu findet man in den im Literaturverzeichnis angegebenen Quellen.

Zusammenfassung der Aufgabe

Implementieren Sie Tabu Search. Lassen Sie dabei den Benutzer Ihres Programmes den Datensatz (es existieren drei Testdateien), die Tabudauer und die Anzahl von Iterationen wählen und fragen sie, ob und nach wie vielen Iterationen er eine Ausgabe der laufenden Iteration wünscht.

Natürlich sollten Sie am Ende der gewählten Anzahl Iterationen das gefundene Optimum, die Gesamtmasse und den Gewinn sowie die TL ausgeben. In den Dateien mit den Testdaten steht in der ersten Zeile die Anzahl der Gegenstände. Die zweite Zeile enthält die maximale zulässige Gesamtmasse des Rucksacks. Alle nachfolgenden Zeilen bestehen aus Gewinn und Masse der Gegenstände. Es gibt drei Dateien: `data.1`, `data.2` und `data.3`. Davon enthält `data.1` das obige Beispiel.

Um keine Energie in die Konvertierung verschiedener Vektortypen zu investieren, existieren diesmal keine Testroutinen. Allerdings gibt es das Programm `a7_lsg`, welches genau die Funktionalität besitzt, die programmiert werden soll. Durch die Konfiguration der Ausgabe des Programmes (Ausgabe nur jede m'te Iteration) können die eigenen Werte auch bei vielen Daten verglichen werden.

C++-Implementierung

Die Vektorklasse aus Aufgabe 4, die in Aufgabe 5 zu einem Klassentemplate erweitert wurde, eignet sich z.B. als Container für die Daten. Sie können darüber z.B. einen Vektor für einen Verbund von Daten anlegen. Ein solcher Verbund ist z.B. eine Klasse der Form[5].

```
struct element {
  int   m_nMoney, m_nWeight, m_nTouched;
  bool  m_bx;
};
```

Man sollte an dieser Stelle bedenken, dass die im Klassentemplate bereitgestellten Operatoren für ein solches Struct wenig Sinn machen bzw. gar nicht definiert sind, d.h.: benutzt man diese Operatoren, so wird dies zu Fehlermeldungen führen.

Weiter müssen Sie entscheiden, wie Sie sich die Lösung x und die TL merken. Eine Möglichkeit ist, den jeweiligen Gegenstand zu markieren (z.B. `m_bx` in `element`). Eine andere Möglichkeit besteht darin, einen Vektor nur mit den jeweiligen Elementen zu führen. Gleiches gilt für die TL. Hier müssen Sie zusätzlich dafür sorgen, daß die Elemente nach TD Schritten wieder freigegeben sind. Allerdings können Sie auch hier eine Form der Markierung wählen. Legt man in einem Feld (z.B. `m_nTouched` in `element`) pro Gegenstand die aktuelle Iterationsnummer i ab, wenn der der Gegenstand bewegt wird (in- oder aus dem Rucksack), so ist dieser Gegenstand tabu, solange $i \leq \text{touched} + TD$ gilt.

Beispiele

In Abbildung 1 (a), (b) und (c) ist der maximale Gewinn der drei Beispiele jeweils in Abhängigkeit der TD aufgetragen. Man sieht, daß es große Bereiche gibt, in denen die gefundene Lösung *relativ* nah an dem Optimum des Verfahrens liegt.[6] Die Schwierigkeit liegt also darin, diese Bereiche zu finden...

Literatur

[1] Domschke, W. und A. Drexl: *Einführung in Operations Research*, Kapitel 6. Springer Verlag, Heidelberg, 4. Auflage, 1998.

[2] Domschke, W., R. Klein und A. Scholl: *Taktische Tabus, Tabu Search: Durch Verbote schneller optimieren.* c't - Magazin für Computertechnik, 12:326–333, 1996. `http://www.vwl.tu-darmstadt.de/bwl3/forsch/projekte/tabu/index.htm`.

[3] Glover, F. und M. Laguna: *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, 1997.

[5] `struct` entspricht einer Klasse, in der alle Variablen und Funktionen `public` sind.

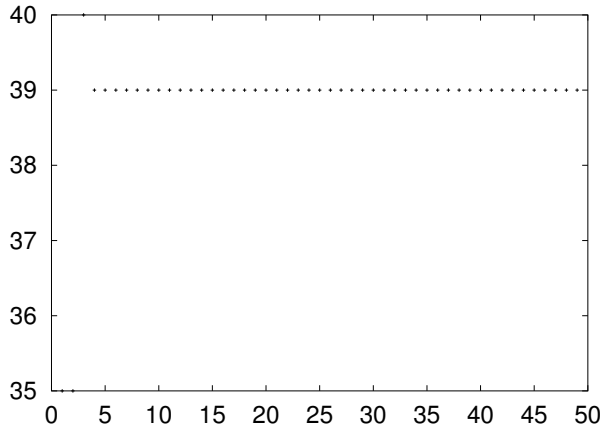
[6] Vielleicht hat jemand Lust, das wirkliche Optimum, z.B. mit einem *Branch-and-Bound-Verfahren* [7], zu ermitteln.

[4] Hromkovič, J.: *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics.* Texts in Theoretical Computer Science. Springer Verlag, Heidelberg, 2. Auflage, 2003.

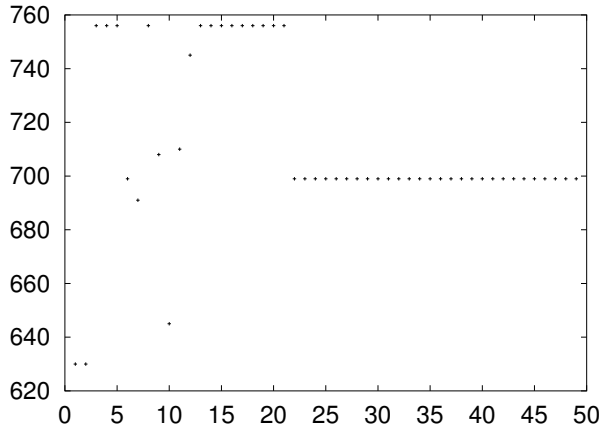
[5] Martello, S. und P. Toth: *Knapsack Problems: Algorithms and Computer Implementations.* Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1990. `http://www.or.deis.unibo.it/knapsack.html`.

[6] Michalewicz, Z. und D. B. Fogel: *How to solve it: modern heuristics.* Springer Verlag, Heidelberg, 2. Auflage, 2004.

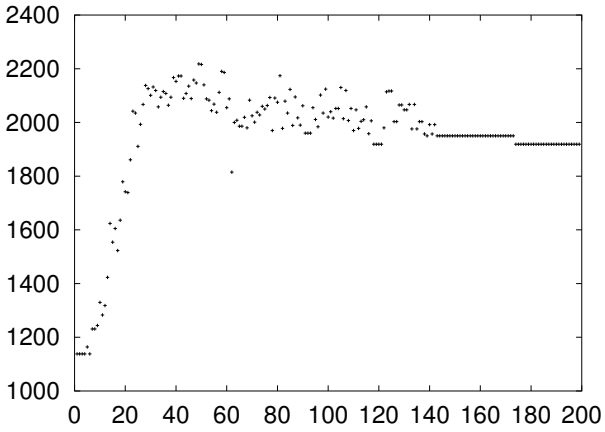
[7] Scholl, A., G. Krispin, R. Klein und W. Domschke: *Besser beschränkt, Clever optimieren mit Branch and Bound.* c't - Magazin für Computertechnik, 10:336–345, 1997. `http://www.vwl.tu-darmstadt.de/bwl3/forsch/projekte/bb/index.htm`.



(a) 6 Elemente, 1000 Iterationen



(b) 25 Elemente, 1000 Iterationen



(c) 500 Elemente, 10000 Iterationen

Abbildung 1: Beispielgrafiken: Es sind jeweils die ermittelten Gewinne in Abhängigkeit von der Tabudauer dargestellt.