Vereinfachung bei der Lösung realer Probleme

Insbesondere bei der Lösung von Suchproblemen ist das Problem vereinfachter Umgebungen, dass die Problemstellungen häufig so stark vereinfacht werden, dass die Schülerinnen und Schüler sie sehr schnell intuitiv lösen können. Das liegt daran, dass der Mensch in der Lage ist, sehr schnell vielseitig vorzugehen, vorausschauend zu denken, zu entscheiden und zu handeln und dabei für die Lösung nicht relevante Alternativen frühzeitig zu verwerfen.

Bei diesem intuitiven Lösungsansatz sind Menschen nicht auf bewusstes algorithmisches Denken angewiesen, Lösungsansätze mit Computereinsatz schon.

Daher muss es das Ziel sein, bei der Vereinfachung eine künstliche Umgebung zu schaffen, die dennoch den algorithmischen Ansatz erkennen lässt.

Hier ist der sonst sehr interessante Ansatz von Gallenbacher[1] nicht ganz schlüssig, weil eigentlich nicht ganz zu erkennen ist, weshalb denn noch ein Algorithmus ausformuliert werden sollte, wenn man ihn denn gar nicht programmieren will.

Gezielte Vorgaben

Wenn für das ganz einfache Rucksackproblem[2] versucht wird, eine Lösung in einer Programmierumgebung zu finden, bei denen die Stücke allein durch einen Wert charakterisiert sind, dann muss man durch die Vorgabe der Werte und Mittel daher sehr deutlich lenken.

Man sollte also als Stückeliste eine dem Wert nach sortierte Liste verwenden. Ist dann durch die Aufgabenstellung vorgegeben, dass demjenigen, der den Rucksack [Container] füllen soll, nicht dessen Größe bekannt ist, kann man davon ausgehen, dass die Schülerinnen und Schüler nach einer "gierigen" Strategie[3] verfahren werden, bei der sie zunächst mit den größten Stücken den Container zu füllen versuchen werden. Der Container, gespielt von einer zweiten Person, kennt natürlich sein Fassungsvermögen und wird Stücke zurückweisen, die nicht mehr hinein passen. Die Person, die den Container füllen will, wird dann das Stück verwerfen und das nächstkleinere versuchen.

Beispiel

Wählt man als Stückeliste beispielsweise '(30 30 30 20 20 20 20 10), die *[für den Füllenden unbekannt*] in einen Container mit Fassungsvermögen 80 passen sollen, dann enthält die Situation die wichtigen beim erfolgreichen Füllen mit der gierigen Strategie auftretenden Fälle. Der zugehörige Algorithmus lässt sich einfach formulieren und das entsprechende Programm lässt sich leicht schreiben.

Das Programm

```
(define
  (fuelle stuecke container kapazitaet)
  (cond
    ((voll? container kapazitaet) container)
    ((null? stuecke) #f)
    ((zu-voll? container kapazitaet)
```

1 Jens Gallenbacher: Abenteuer Informatik

2 Vergleiche die Problemstellung bei Gallenbacher: Dies ist nicht völlig vereinfacht, da die Stücke zwei Eigenschaften haben, Größe und Wert.

3 Bei der gierigen [greedy] Strategie wählt man für die Lösung immer zuerst das größte [teuerste, schwerste,...] Stück aus.

```
     (fuelle stuecke (rest container) kapazitaet))
    (else
     (fuelle (rest stuecke) (cons (first stuecke) container) kapazitaet))))
```

Dazu die Hilfsfunktionen:

```
(define
  (fuellung container)
  (if
   (null? container)
   0
   (+
    (first container)              ; alternativ (car container)
    (fuellung (rest container))    ; alternativ (fuellung (cdr container))
    )))
(define
  (voll? container kapazitaet)
  (= (fuellung container) kapazitaet))
```

```
(define
  (zu-voll? container kapazitaet)
  (> (fuellung container) kapazitaet))
```

Das war's natürlich noch nicht

Durch eine Variation des Fassungsvermögens z.B. auf den Wert 100 wird anschließend auf einfache Weise erkennbar, dass mit diesem Verfahren nicht alle lösbaren Fälle auch wirklich gelöst werden können. Das vorher entwickelte Programm füllt nämlich das dritte Stück 30-er-Stück ein und damit kann eine Füllung von 100 nicht mehr erreicht werden, obwohl mit den Stücken '(30 30 20 20) sehr wohl eine Lösung möglich wäre.

Von greedy zu vollständigen Suchverfahren

Der Schritt zu einem vollständigen Suchverfahren, nämlich der *Tiefensuche mit backtracking* ist nun sehr einfach. Es muss eine Möglichkeit geschaffen werden, das Einfüllen eines Stückes auch dann rückgängig zu machen, wenn es prinzipiell zulässig war und das Programm dazu zu bringen, zu dem Bearbeitungsschritt zurück zu gehen, an dem diese Entscheidung gefällt wurde.

Hier die Erweiterung auf die Tiefensuche mit backtracking:

```
(define
  (fuelle stuecke container kapazitaet)
  (cond
    ((voll? container kapazitaet) container)
    ((null? stuecke) #f)
    ((zu-voll? container kapazitaet)             ; Änderung: #f zurückgeben
     #f)                       ; Schritt in die Tiefe, Einpacken versuchen:
    ((fuelle (rest stuecke) (cons (first stuecke) container) kapazitaet))
    (else                      ; Alternative, ohne das Stueck, versuchen:
     (fuelle (rest stuecke) container) kapazitaet)))
```

backtracking anschaulich machen

Man kann dieses backtracking für die Schülerinnen und Schüler erfahrbar machen, wenn man für die rekursiven Aufrufe der Funktion jeweils einen Aufrufzettel schreibt, der dann weiter gereicht und nach einer erfolglosen Bearbeitung an den Weitergebenden zurück gereicht wird. Derjenige muss dann einen erneuten Versuch des Aufrufs ohne das Stück starten. Führt man das Verfahren mit einer ausreichenden Zahl von Personen durch, dann wird der Suchraum und das Navigieren in diesem Suchraum anschaulich erfahrbar.