Systematische Suche

Martin Klemm, Martin Schicht

27.04.2009

Gliederung

1. Rucksackproblem
2. Tiefensuche
3. Breitensuche
4. Beschränkung
5. Nichtdeterminismus

Rucksackproblem

Allgemeines

- engl. Knapsack Problem
- ist ein Optimierungsproblem der Kombinatorik

Begriffe

- nichtleere Menge von n -Gegenständen
- Wert des Gegenstandes w_i
- Gewicht des Gegenstandes g_i
- Kapazität K

Problem

Fülle den Rucksack so, dass die Wertsumme der eingepackten Gegenstände maximal ist und die Kapazitätsschranke eingehalten wird.

Rucksackproblem

Arten des Rucksackproblems

- 0/1 Rucksackproblem
  - Gegenstand wird komplett (1) oder gar nicht (0) eingepackt
- Bruchteilrucksackproblem
  - Gegenstand wird teilweise eingepackt
  - Teil t, $0\% <= t <= 100\%$

Rucksackproblem

Forderung

maximale Wertsumme bei Nichtüberschreiten der Kapazitätsschranke

mathematische Formel

$\sum_{i=1}^{n} x_i w_i \rightarrow max$, wobei $\sum_{i=1}^{n} x_i g_i \leq K$, mit $x_i \in \{0, 1\}$

Rucksackproblem

Teilmengen-Summen-Problem

- engl. subset-sum problem
- Spezialfall des Rucksackproblems
- Gesucht:
  - Teilmenge von der Menge M, deren Elementensumme möglicht an K heran kommt und dieses auch erreichen darf

$M = \{a_i | 1 \leq i \leq n \text{ und } a_i, n \in N\}$
für $g_i = w_i$, mit $g_i, w_i \in N$

Teilmengen-Summen-Problem

Anwendung:

- Transportunternehmen

Rucksackproblem

Lösung des Rucksackproblems

- intuitives Verfahren
  - zufälliges Befüllen des Rucksacks
  - nachträgliches Austauschen von Gegenständen um Gesamtwert zu erhöhen

Rucksackproblem

Lösung des Rucksackproblems

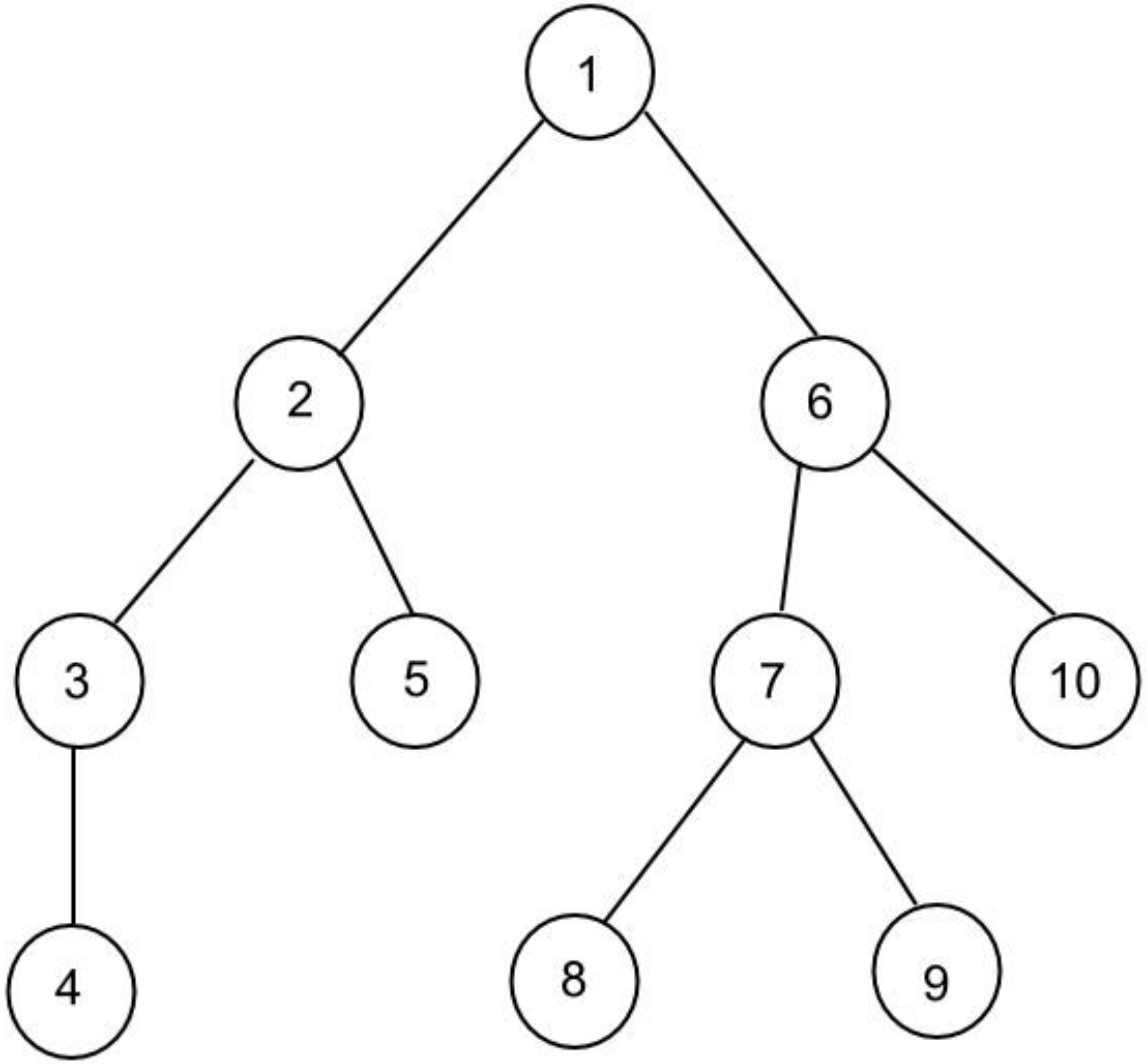
- intuitives Verfahren
  - zufälliges Befüllen des Rucksacks
  - nachträgliches Austauschen von Gegenständen um Gesamtwert zu erhöhen
  - Problem : Man weis nie, ob das Wertmaximum erreicht ist.
- Suchalgorithmen
  - Tiefensuche und Breitensuche

Tiefensuche

Was ist Tiefensuche

- engl. depth-first search
- ein Verfahren zum Suchen eines Knotens in einem Graphen
- zählt zu den uninformierten Suchalgorithmen

Tiefensuche am Beispiel eines Graphen

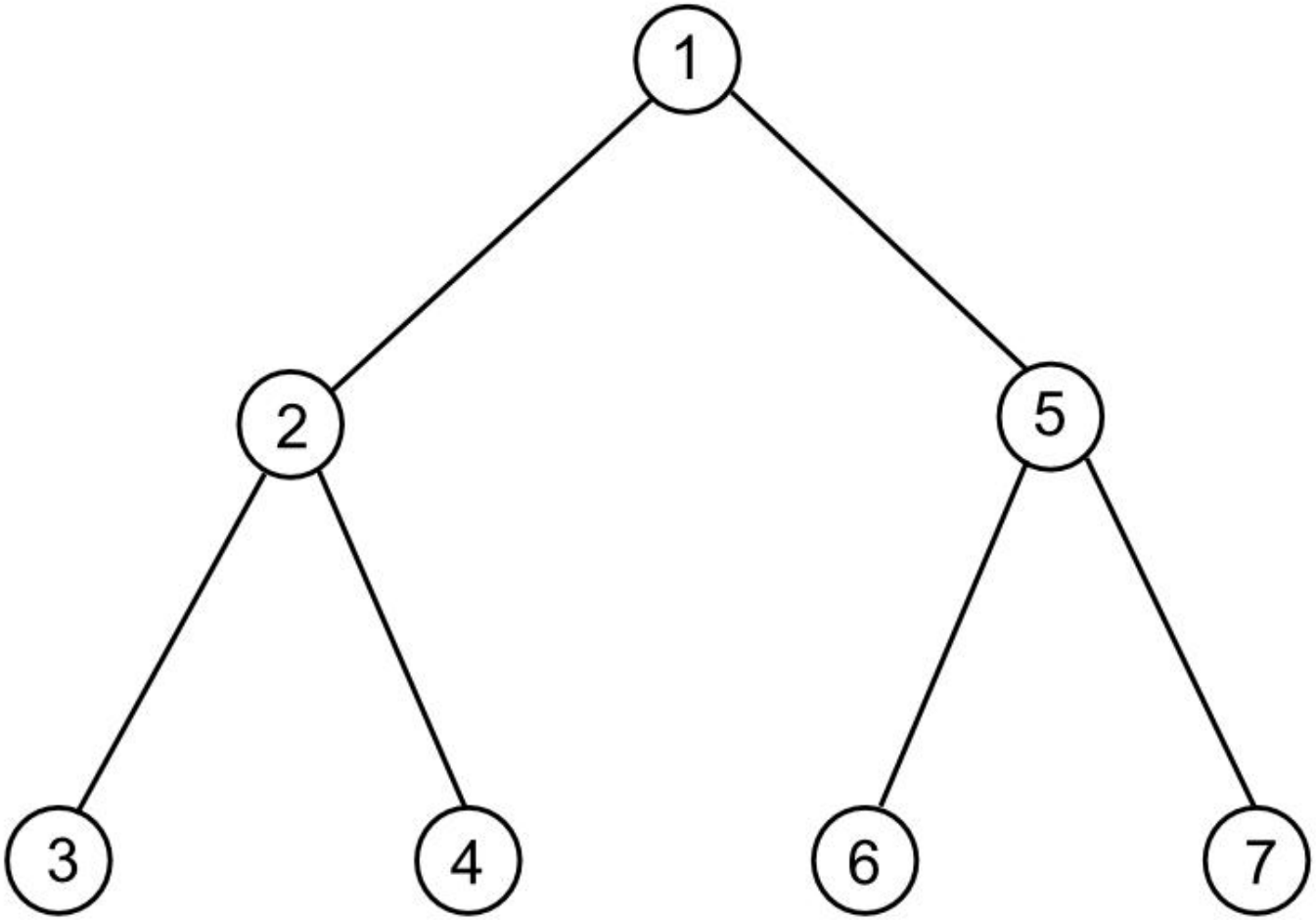


Durchlauf eines Graphen (tree traversal)

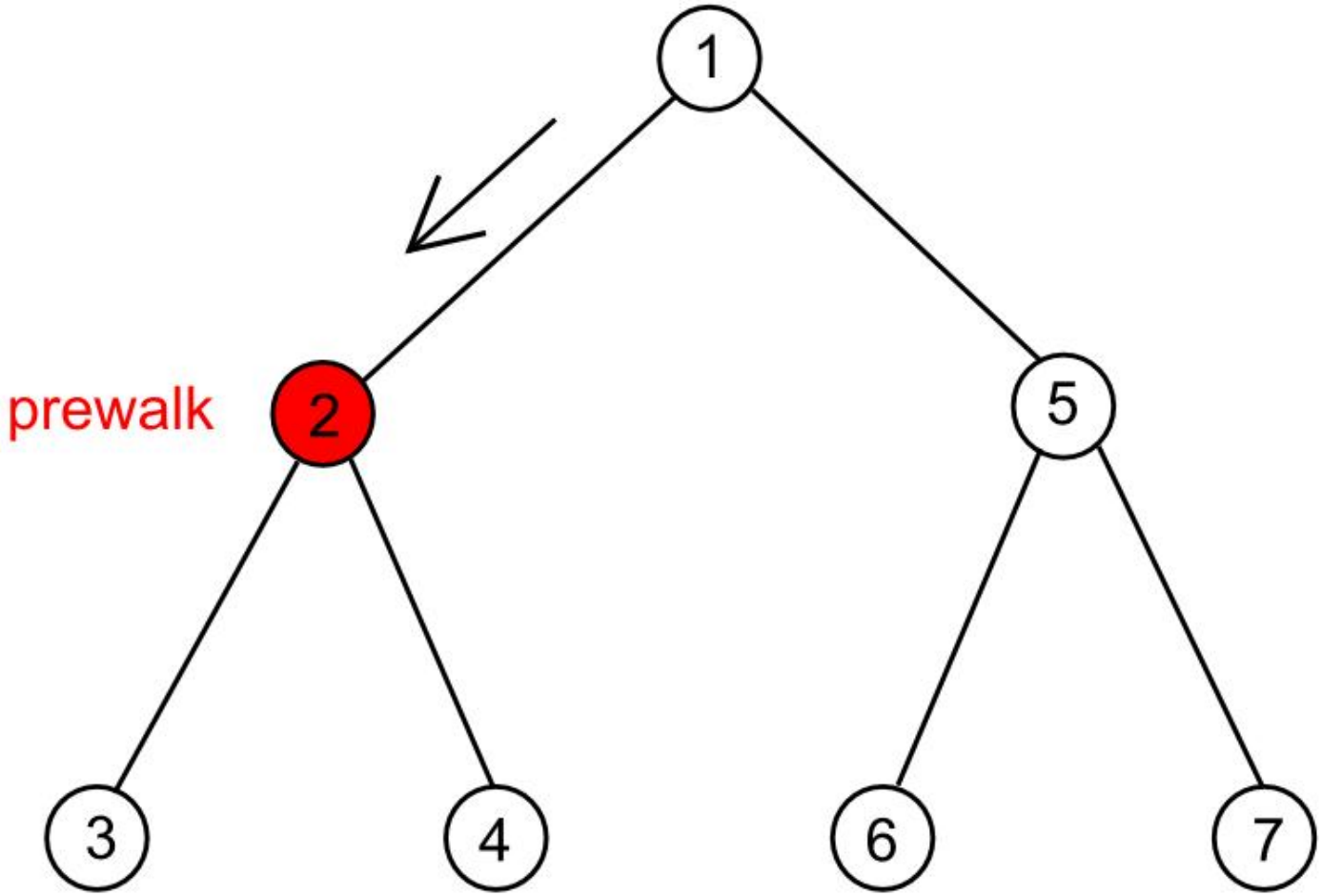
prewalk, centralwalk, postwalk

- prewalk : 1. Berührung eines Knoten
- centralwalk : 2. Berührung eines Knoten
- postwalk : 3. Berührung eines Knoten

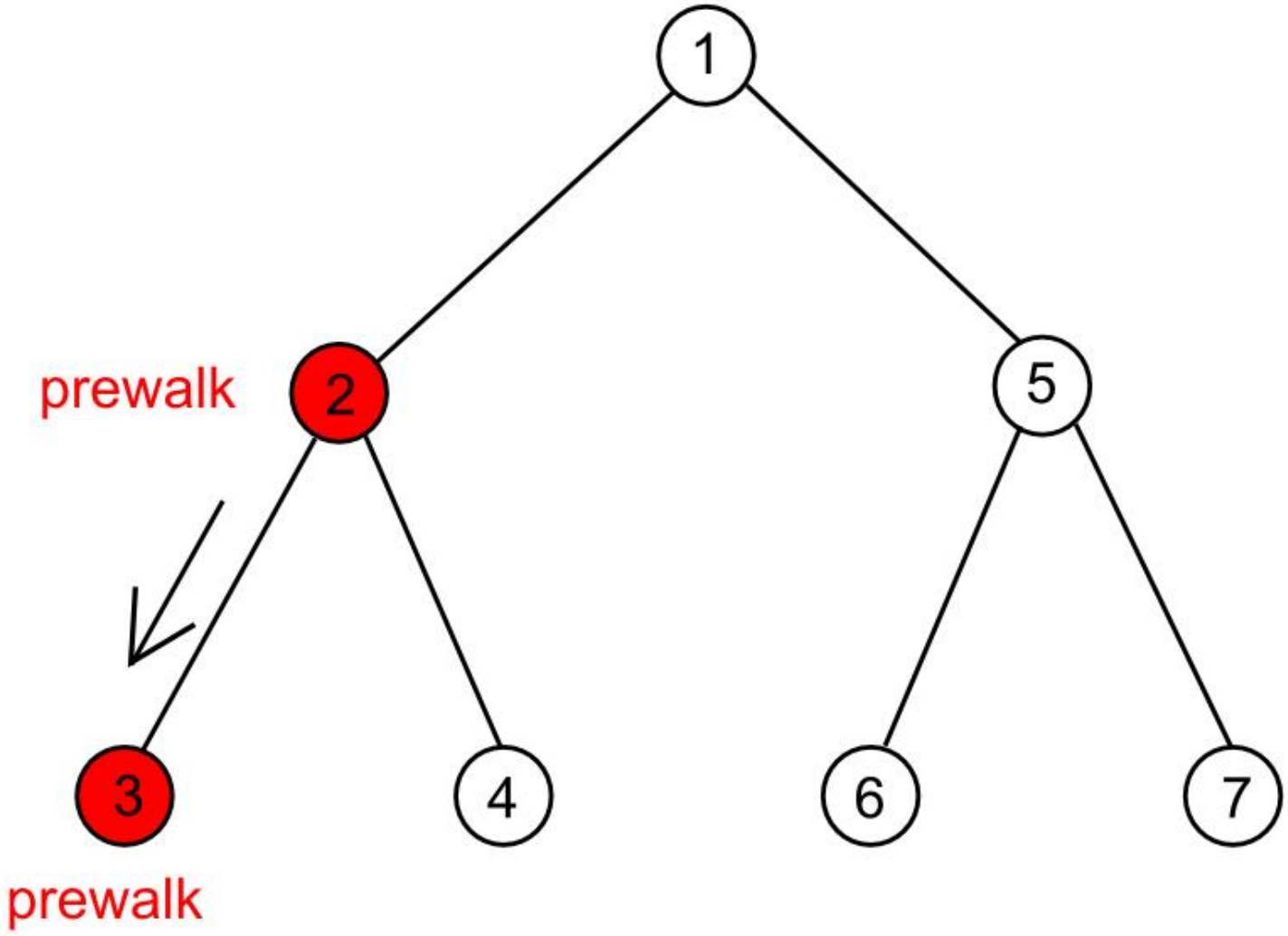
Durchlauf eines Graphen (tree traversal)



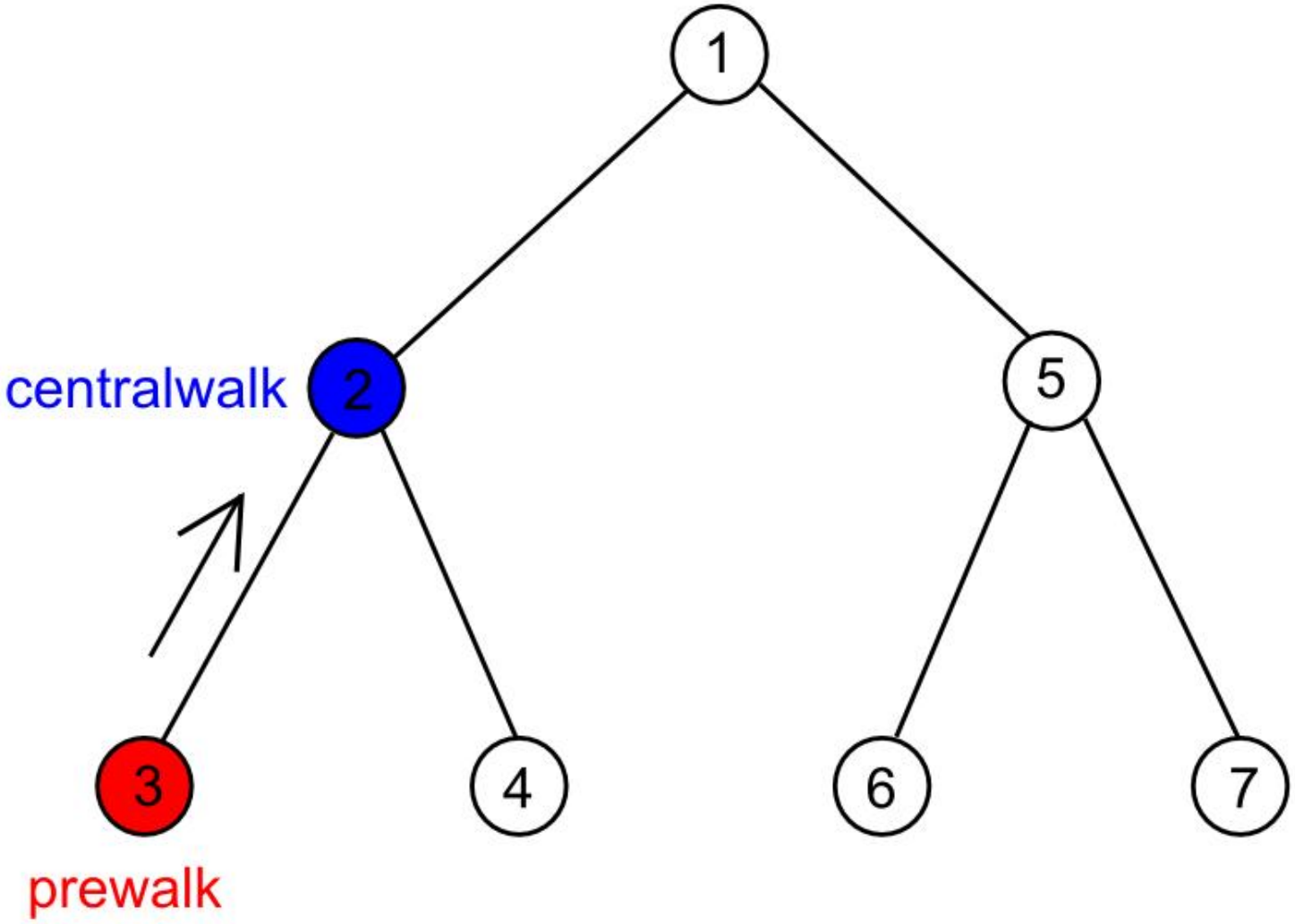
Durchlauf eines Graphen (tree traversal)



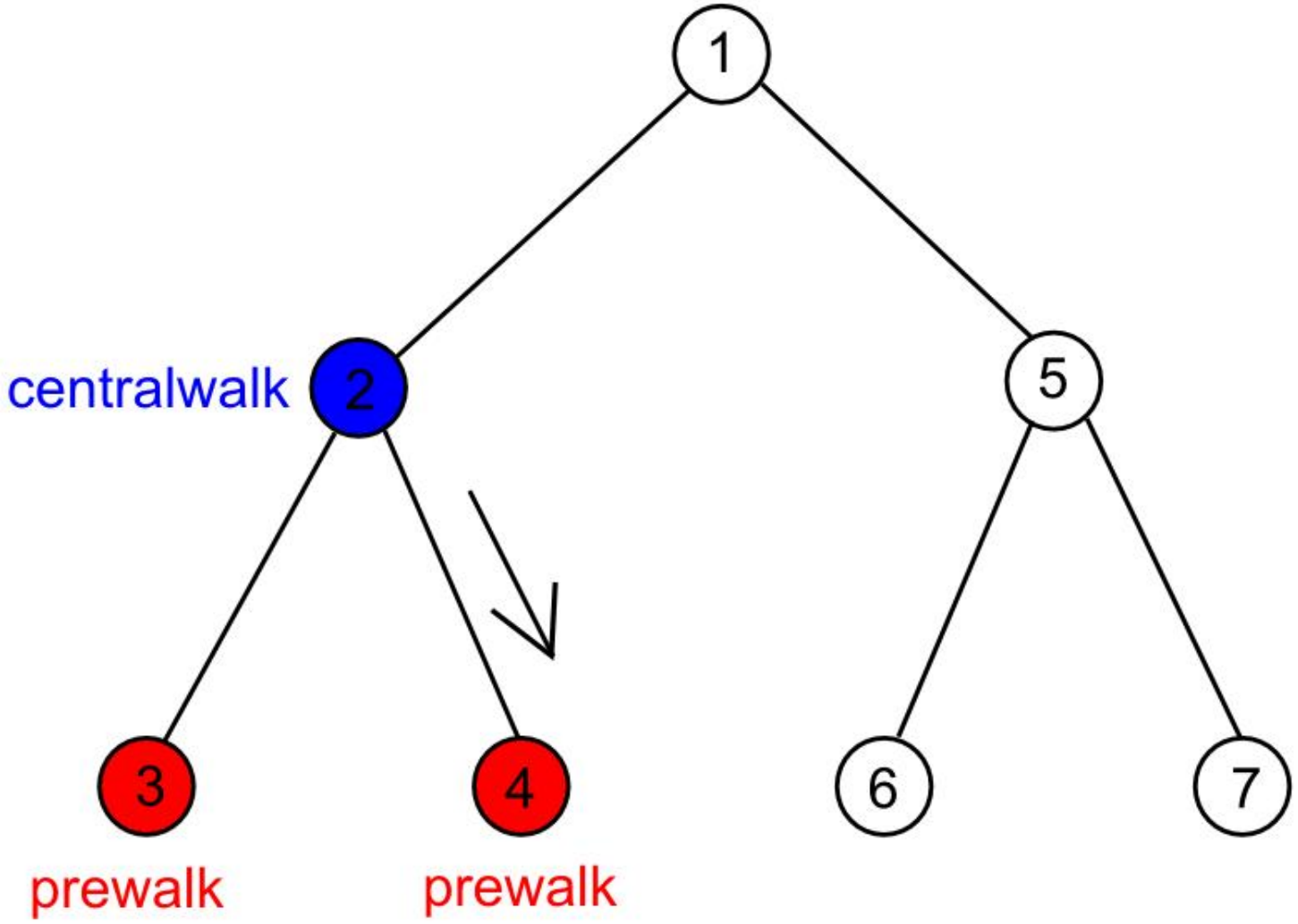
Durchlauf eines Graphen (tree traversal)



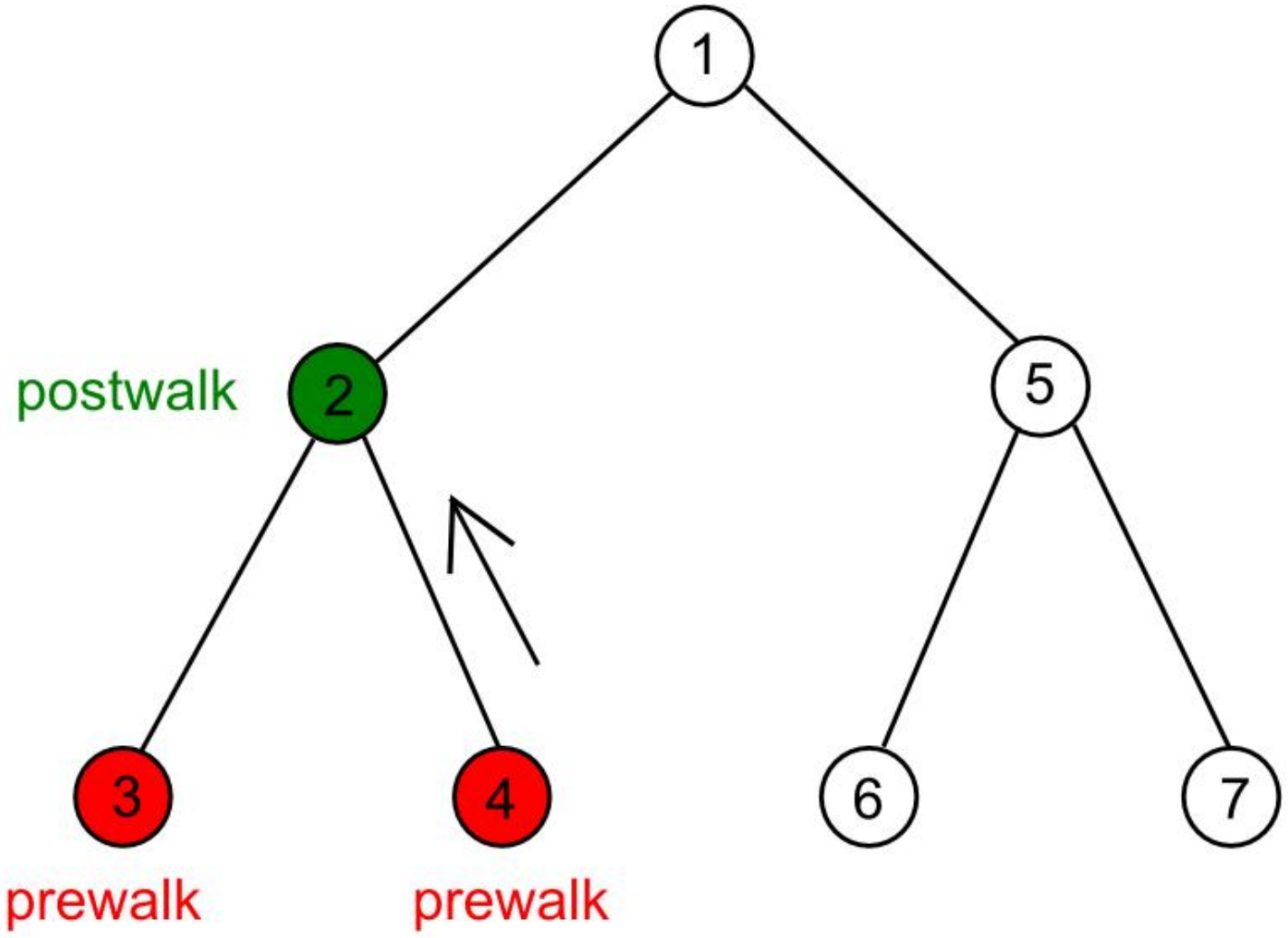
Durchlauf eines Graphen (tree traversal)



Durchlauf eines Graphen (tree traversal)



Durchlauf eines Graphen (tree traversal)



Algorithmus der Tiefensuche

- Bestimme den Knoten mit dem die Suche beginnen soll.
- Betrachte die erste Kante.
- Wenn der nächste Knoten erweiterbar ist wird wieder expandiert.
- Kann nicht expandiert werden gehe mittels backtracking zum darüber liegenden Knoten zurück und betrachte eine alternative Kante.
- Ist kein Knoten mehr expandierbar wurden alle Ecken bereits besucht.

Graph der Tiefensuche

Warum Binärbaum?

- man kann aus jedem Graphen der Tiefensuche einen Binärbaum umformen
- Reihenfolge egal, Kombination ist wichtig
- Suchbaum ist Uniform : konstanter Verzweigungsgrad, einheitliche Tiefe
- Zielknoten liegt mit gleicher Wahrscheinlichkeit in der Maximaltiefe d (ganz unten im Suchbaum)

Aufbau des Graphen

- Anzahl der Knoten
  - $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$
- Anzahl der Blätter
  - 2^n
- Anzahl der Kanten
  - $|E| = |V| - 1$
    $|V|$ = Anzahl der Knoten

Tiefensuche

Entscheidungsbaum

- JA/Nein-Entscheidungen
- Baum wird von links her aufgebaut
- wird als eine Art Protokoll generiert
- praktisch heißt das, jeder Gegenstand wird eingepackt, nimmt den jeweils letzten wieder heraus und entscheidet sich in jedem Knoten anders als vorher (Backtracking)

Tiefensuche

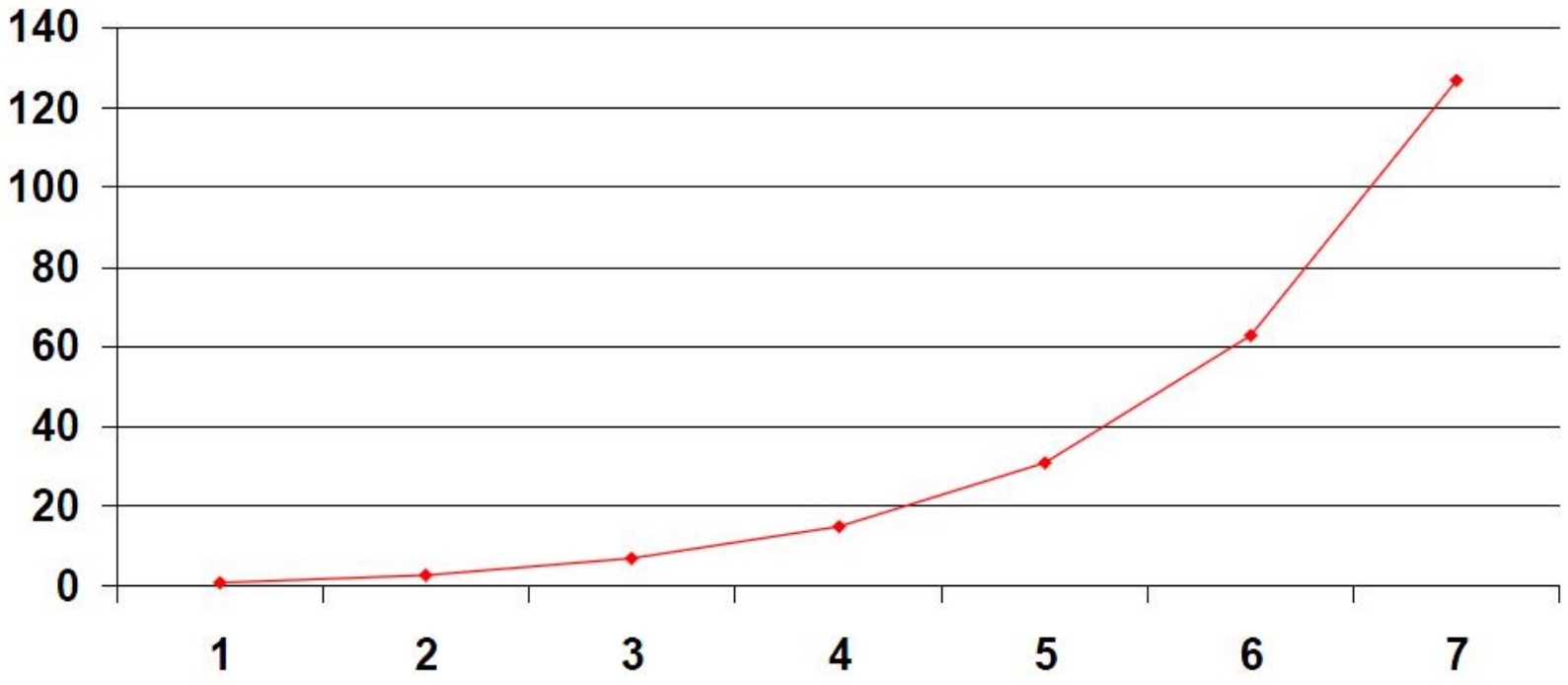
Eigenschaften

- Komplexität $= \mathcal{O}(|V| + |E|)$
- $|V|$ = Anzahl der Knoten
- $|E|$ = Anzahl der Kanten
- im worst case werden alle Knoten betrachtet.
  - m = größtmögliche Tiefe
  - b = Breite (Alternative Wege).
- Optimalität: liefert auch nicht optimale Lösungen
- Speicherplatzbedarf ist linear
- Ein Ergebnis kann schneller gefunden werden als bei der speicheraufwendigen Breitensuche

exponentieller Aufwand

Aufwand

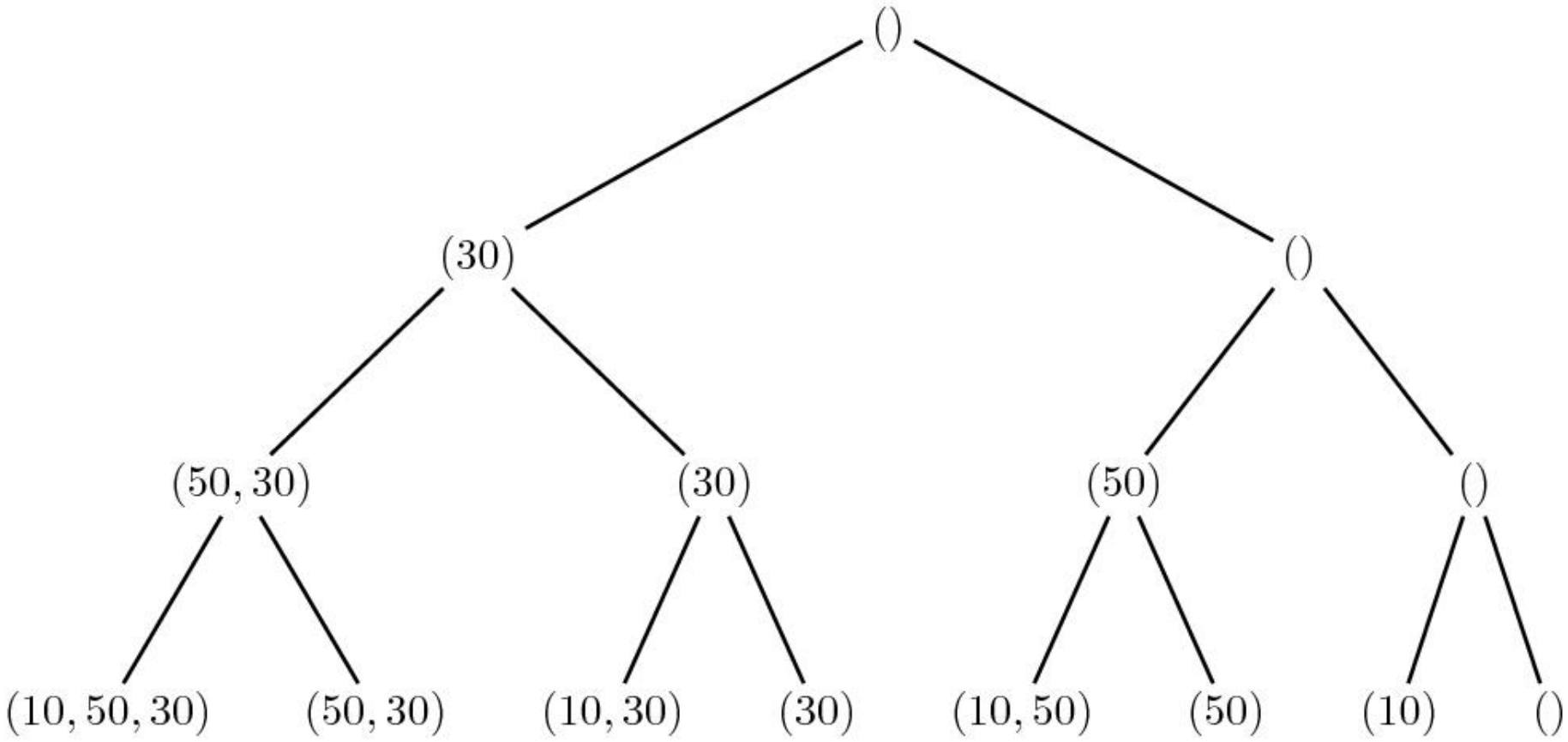
Für die Erzeugung eines gesamten Baumes mit n Gegenständen hat man einen exponentiellen Aufwand von $\mathcal{O}(2^n)$



Tiefensuche in Scheme

```
(define rucksack1
(lambda (g-ls s-ls)
(if (null? g-ls)
s-ls
(begin
(rucksack1 (cdr g-ls) (cons (car g-ls) s-ls))
(rucksack1 (cdr g-ls) s-ls)))))
```

Tiefensuche am Beispiel (30,50,10)



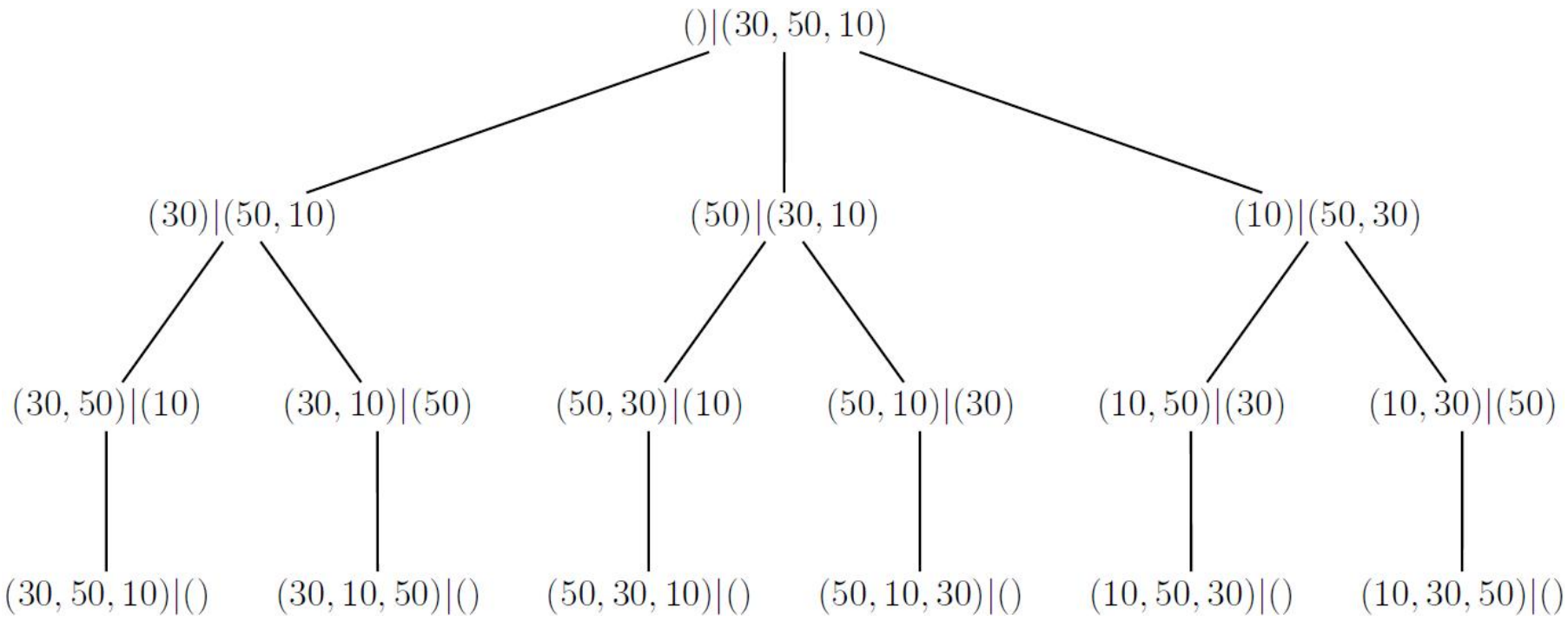
Tiefensuche in Scheme

```
> (rucksack1 '(30 50 10) '())
| (rucksack1 (30 50 10) ())
| (rucksack1 (50 10) (30))
| | (rucksack1 (10) (50 30))
| | (rucksack1 () (10 50 30))
| | (10 50 30)
| | (rucksack1 () (50 30))
| | (50 30)
| | (rucksack1 (10) (30))
| | (rucksack1 () (10 30))
| | (10 30)
| (rucksack1 () (30))
| (30)
```

Tiefensuche in Scheme

```
| (rucksack1 (50 10) ())
| (rucksack1 (10) (50))
| | (rucksack1 () (10 50))
| | (10 50)
| (rucksack1 () (50))
| (50)
| (rucksack1 (10) ())
| (rucksack1 () (10))
| (10)
| (rucksack1 () ())
| ()
()
```

Breitensuche



Algorithmus

1. Setzte Startknoten und speichere ihn in einer Warteschlange ab.
2. Entnehme ersten Knoten
   - Zielknoten gefunden $\rightarrow$ Beende Suche mit Erfolg
   - else $\rightarrow$ hänge alle unmarkierten Nachfolger dieses Knotens an das Ende der Warteschlange
3. Wiederhole 2.
4. Warteschlange leer $\rightarrow$ Beende Suche mit Misserfolg

Anzahl der Knoten

$$1+n+n(n-1)+n(n-1)(n-2)+\ldots$$
$$\ldots+n(n-1)(n-2)(n-3)(n-(n-1))$$

$$=\frac{n!}{0!}+\frac{n}{1!}+\frac{n!}{2!}+\ldots+\frac{n!}{n!}=\sum_{k=0}^{n}\frac{n!}{k!}$$

Analyse: Speicheraufwand

- Repräsentierung als Liste
- Alle Knoten einer Ebende befinden sich in der Liste, bevorerster knoten untersucht wird
- $\Rightarrow Speicher(B) = b^d \rightarrow exponentiell$
- Die Knoten der Ebene davor wurden alle durch ihre Nachfolgeknoten ersetzt.

Analyse: Zeitaufwand

- best case: erste Knoten der nächsten Ebene der Zielknoten,
  aber es müssen alle Knoten der oberen Ebene (d-1) überprüft worden sein
- worst case: Letzte Knoten der letzten Ebene der Zielknoten

Kürzeste Pfade

- jeweils vom Start- zum Zielknoten
- kürzester Abstand = minimale Anzahl von Kanten über alle Pfade von s nach v
- kürzester Pfad = Pfad der Länge $\delta(s, v)$

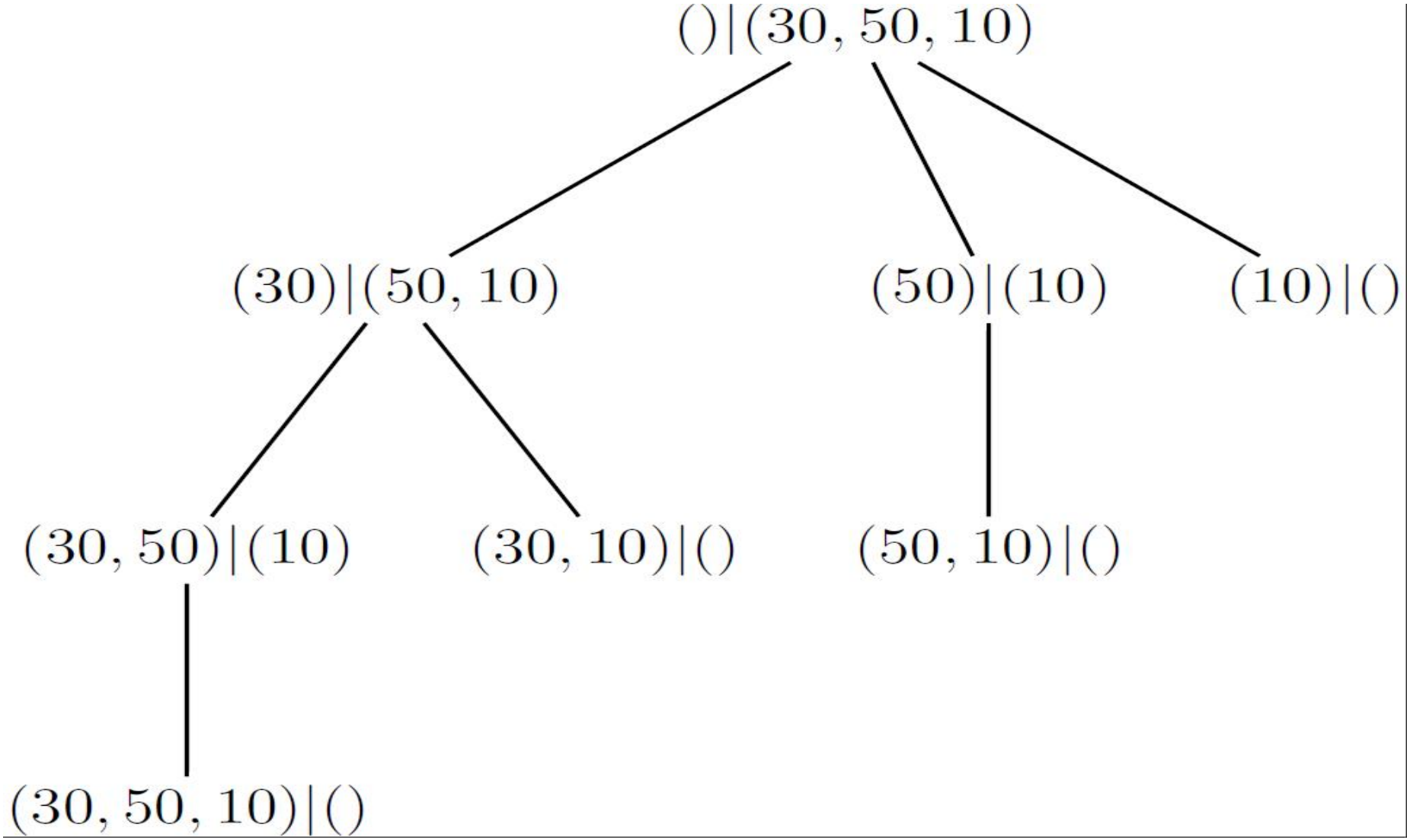
Vollständigkeit / Fairness

= Wenn eine Lösung existiert, wird diese auch gefunden.
Falls der Graph unendlich ist, divergiert die Breitensuche und die Kanten werden *fair* durchsucht.

reduzierter Beitensuchbaum

- Reihenfolge hat für Lösungsmenge keinen Einfluss
- Problem: doppelte Zustände
- Lösung: Basismenge wird verbrauchend gelesen
- Ergebnis: reduzierter Breitensuchbaum
  $\Rightarrow$ 8 Knoten entsprechen 8 Blättern der Tiefensuche

reduzierter Breitensuchbaum



Vergleich : Tiefen- und Breitensuche

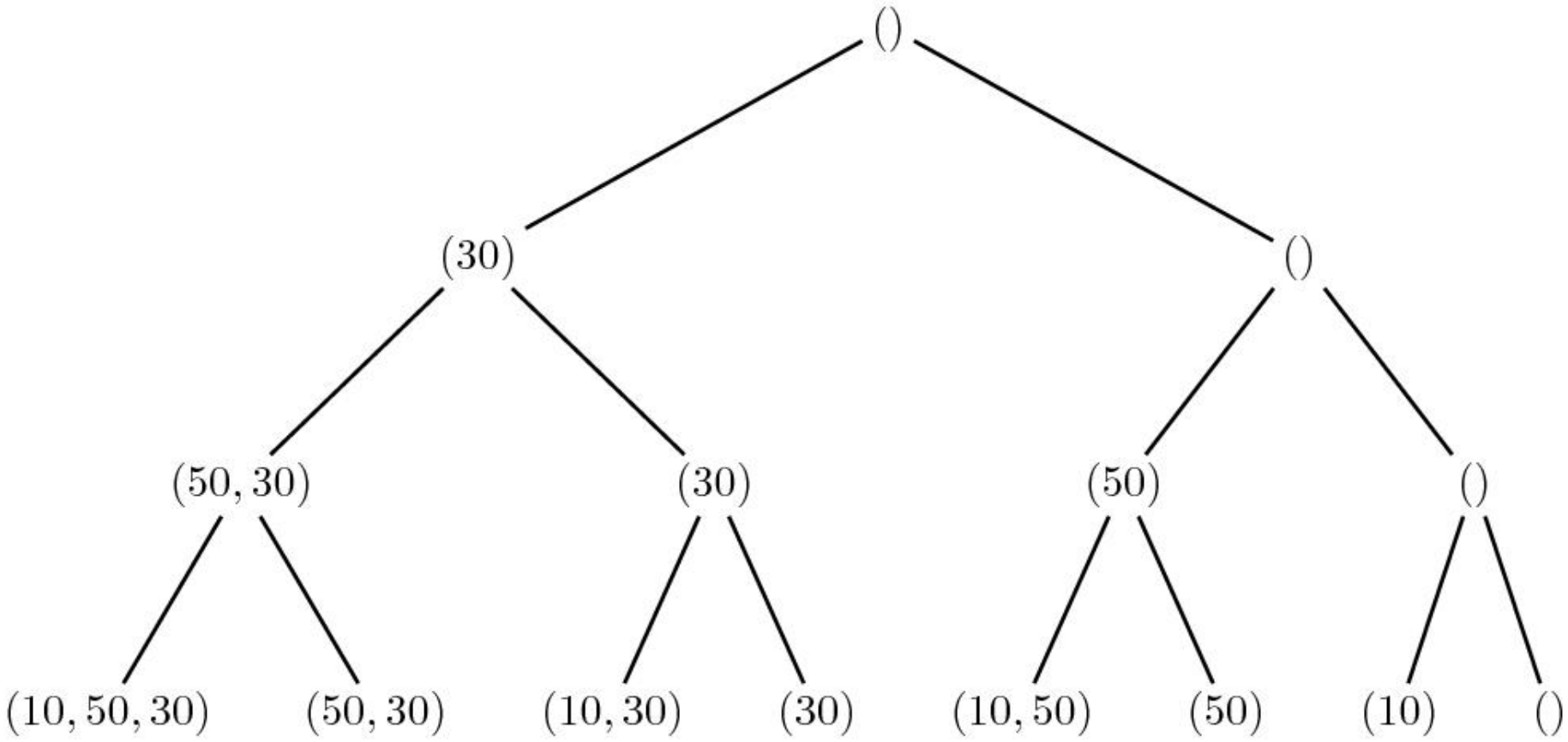
Kriterium	**Tiefensuche**	**Breitensuche**
Schnelligkeit	links	rechts
Vollständigkeit	nur, wenn Suchbaum endlich	ja
Optimal bzgl. Pfadlänge	nein, liefert auch nicht-optimale Lösungen	ja, da alle Lösungen immer minimale Tiefe haben
Speicher	nur für einen Pfad (+ anhängende Knoten)	ganze bisher besuchte Graph
Anwendung	zuerst gefundene Lösung	Ermittlung aller Lösungen
Beispiel	8-Puzzle	Graphentheorie

Beschränkte Tiefensuche

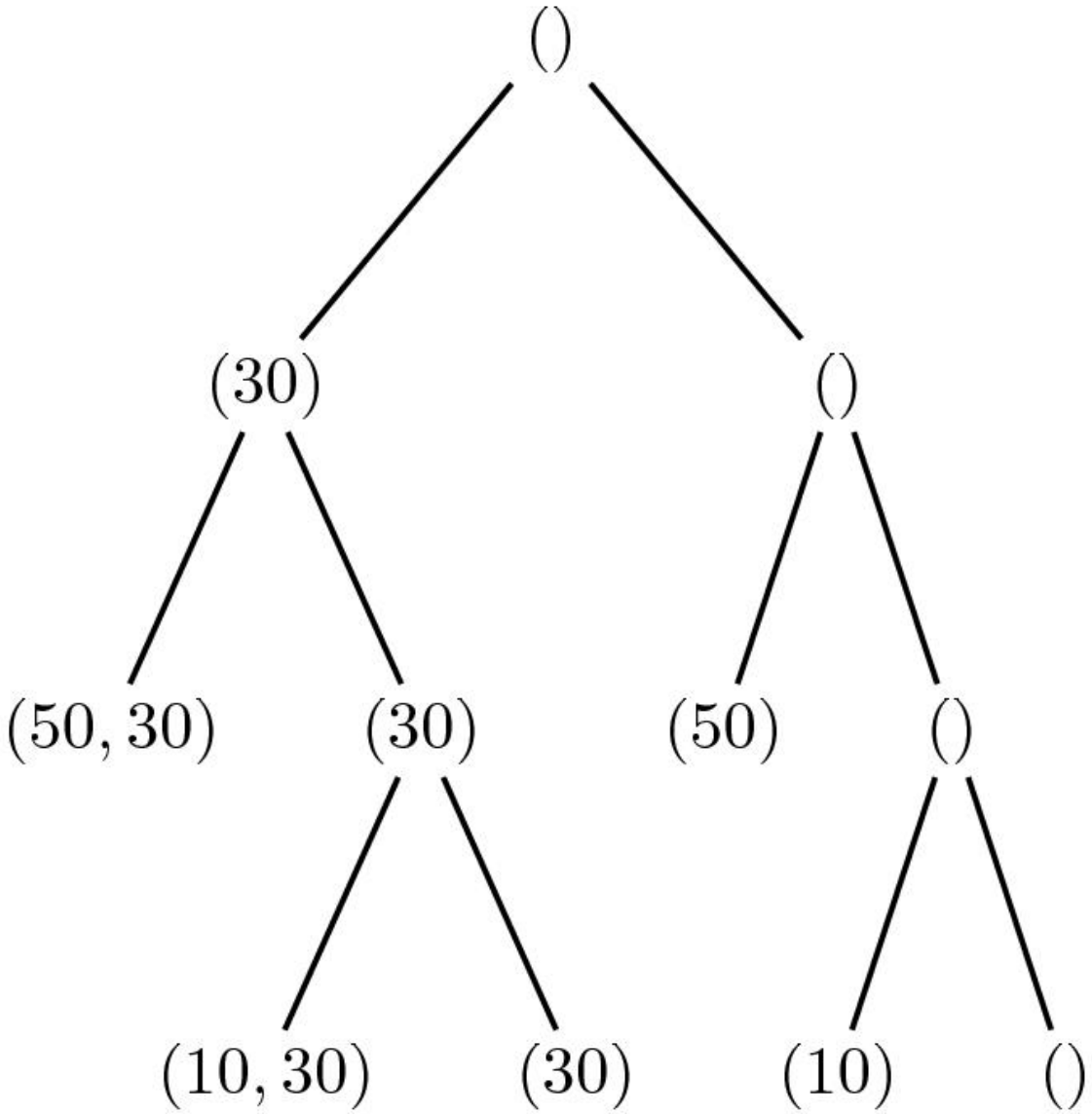
Kapzitätsbeschränkung

- Kapazität K
- einschränken der Suche bei überschreiten von K

ohne Beschränkung



mit Beschränkung K=42



beschränkte Tiefensuche in Scheme

```
(define rucksack2
(lambda (g-ls s-ls)
(if (null? g-ls)
s-ls
(begin
(if (hoechstens-maxgewicht? s-ls 42)
(rucksack2 (cdr g-ls) (cons (car g-ls) s-ls)))
(if (hoechstens-maxgewicht? s-ls 42)
(rucksack2 (cdr g-ls) s-ls))))))

(define hoechstens-maxgewicht?
(lambda (ls k)
(<= (apply + ls) k)))
```

beschränkte Tiefensuche in Scheme

```
(rucksack2 '(30 50 10) '())
|(rucksack2 (30 50 10) ())
|(rucksack2 (50 10) (30))
||(rucksack2 (10) (50 30))
|| #<void>
|(rucksack2 (10) (30))
||(rucksack2 () (10 30))
||(10 30)
|(rucksack2 () (30))
|(30)
```

```
|(rucksack2 (50 10) ())
|(rucksack2 (10) (50))
| #<void>
|(rucksack2 (10) ())
|(rucksack2 () (10))
|(10)
|(rucksack2 () ())
|()
()
```

Beschränkte Tiefensuche mit Ergebnisliste

```
(define rucksack3
(lambda (g-ls k)
(let ((max 0) (max-s-ls '())))
(letrec
((helfer
(lambda (g-ls s-ls)
(display s-ls)(newline)
(if (null? g-ls)
(let ((gesamtgewicht (apply + s-ls)))
(if (and
(> gesamtgewicht max)
(hoechstens-maxgewicht? s-ls k))
```

```
(begin
(set! max gesamtgewicht)
(set! max-s-ls s-ls)))
(cons max-s-ls max))
(begin
(if (hoechstens-maxgewicht? s-ls k)
(helfer (cdr g-ls) (cons (car g-ls) s-ls)))
(if (hoechstens-maxgewicht? s-ls k)
(helfer (cdr g-ls) s-ls)))))))
(helfer g-ls '()))))))
```

Beschränkte Tiefensuche mit Ergebnisliste für (30,50,10) mit K=42

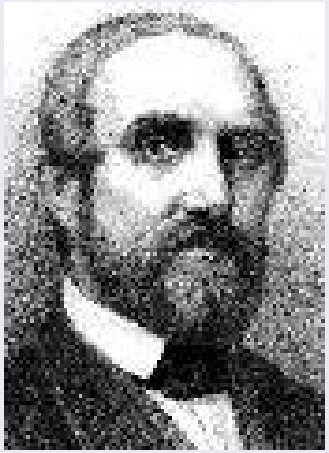
```
()
(30)
(50 30)
(30)
(10 30)
(30)
()
(50)
()
(10)
()
((10 30) . 40)
```

8-Damenproblem

Das Problem

- 8 Damen auf ein 1 Schachbrett
- Jede Dame spielt für sich
- Damen dürfen sich nicht gegenseitig schlagen
  Wie viele mögliche Aufstellungen gibt es?

Geschichte



- 1848 von Max Bezzel(I.)
- 1850 Franz Nauck:
  92 Möglichkeiten
- 1874 James W. L.Glaisher :
  Beweis

$n \times n$ Damenproblem

Der unaufhörliche Drang zur Verallgemeinerung

- nicht tiviale Lösungen
- naiver Brute Force-Algorithmus nicht effizient

$n \times n$ Damenproblem

n	1	2	3	4	5	6	7	8	9	10
eindeutig	1	0	0	1	2	1	6	12	46	92
insgesamt	1	0	0	2	10	4	40	92	352	724

n	16	18	22
eindeutig	1.846.955	83.263.591	336.376.244.042
insgesamt	14.772.512	666.090.624	2.691.008.701.644

n	24	25 (2005)
eindeutig	28.439.272.956.934	275.986.683.743.434
insgesamt	227.514.171.973.736	2.207.893.435.808.352

Begriffe

Rekursion

- implizierte Definition von Funktionen
- Bäume als Beschreibungsmittel für die Suche

deskriptive Programmierung

- Beschreibung des Ergebnisses
- einzelne Arbeitssschritte werden nicht angegeben

Nichtdeterminismus

- Annahme, dass das gewünschte Resultat zielgerichtet gefunden wird
- Akzeptanz von nichtdeterministischen Algorithmen : Wenn es eine Lösung gibt, wird sie auch gefunden.

Hilfsmittel : Scheme λ

> (load "'ambiguous.ss"')

initialize-amb-fail

- Initialisierung des Ambiguous-Systems
- nullstellige Prozedur (thunk)

choice

- nimmt Liste und gibt "richtiges" Element für Gesamtausdruck aus

```
> (< (choice '(1 2 3 4 5 6)) 4)
#t
> (< (choice '(3 4 5 6)) 3)
#f
```

Hilfsmittel : Scheme λ

bag-of

Gibt eine Liste mit Elementen, die zum Erfolg führen zurück

```
> (bag-of (< (choice '(1 2 3 4 5 6)) 4))
(\#t \#t \#t \#f \#f \#f)
```

Definition: summanden

```
(define summanden
(lambda ()
(list (choice '(0 1 2 3 4 5)) (choice
'(0 1 2 3 4 5)))))
```

Definition: summandensuche

```
(define summandensuche
(lambda (summe)
(let ((kandidatenliste (summanden)))
(if (= (apply + kandidatenliste) summe)
kandidatenliste
(choice '()))))))
```

Aufrufbeispiel

```
>(initialize-amb-fail)
>(summandensuche 6)
(1 5)
> (bag-of (summandensuche 6))
((1 5) (2 4) (3 3) (4 2) (5 1))
```

multichoice

- nimmt eine Liste und eine Zahl n
  und gibt n ”richtige” Elemente aus der Liste zurück
- Beispiel: 6 aus 49

randomchoice

nimmt Zahl n und liefert

```
(choice '(1 2 3 ... n))
```

Definition: rucksack4

```
(define rucksack4
(lambda (gls k)
(let ((sls (multichoice gls (randomchoice
(length gls)))))
(if (<= (apply + sls) k)
sls
(choice '()))))))
```

Aufrufbeispiel

```
> (initialize-amb-fail)
> (rucksack4 '(30 50 10) 42)
(30 10)
> (bag-of (rucksack4 '(30 50 10) 42))
((30 10) (10 30) (30) (10))
```

rucksack5(Auszug)

```
(map
(lambda (sls)
(let ((slssumme (apply + sls)))
(if (> slssumme max)
(begin
(set! max slssumme)
(set! resultat sls)))))
alle-sls))
```

Aufrufbeispiel

```
> (rucksack5 '(30 50 10) 42)
((30 10) . 40)
```

```
Program [redacted]
var n: integer;

function [redacted](n: integer): longint;
var i, f0, f1, f2: longint;
begin
 f0 := 0; f1 := 1; f2 := 1;
 i := 0;
 while i < n do
   begin
     i := i+1;
     f0 := f1;
     f1 := f2;
     f2 := f1 + f0;
   end;
  [redacted] := f0;
end;

begin
  writeln('[redacted] ');
  write('n = '); readln(n);
  writeln [redacted](n));
end.
```

```
Program fibonacci_zahlen_iterativ;
var n: integer;

function fib(n: integer): longint;
var i, f0, f1, f2: longint;
begin
 f0 := 0; f1 := 1; f2 := 1;
 i := 0;
 while i < n do
   begin
    i := i+1;
    f0 := f1;
    f1 := f2;
    f2 := f1 + f0;
   end;
  fib := f0;
end;

begin
  writeln('Fibonacci-Zahlen (iterativ) ');
  write('n = '); readln(n);
  writeln(fib(n));
end.
```

kognitive Effizienz

- = wie einfach ist ein Algorithmus geistig zu erfassen
- höhere Komplexität möglich
- programmiertechnische Hilfsmittel : Rekursion, logische Programmierung, genetische Algorithmen
- Vorteile:
  - höheres Verständnis des Algorithmus
  - kleinerer Zeitaufwand bei Implementierung
  - Übersichtlichkeit, Fehlererkennung
- Nachteil:
  - keine brauchbare Metriken