Abschnitt 4.3

Algorithmen in prozeduraler Formulierung

Unterabschnitt 4.3.1

Einführung in die Algorithmenanalyse

Abstraktere Effizienzbegriffe:

Statt ein installiertes Programm zu betrachten, sieht man üblicherweise von Details ab und betrachtet Effizienz nur näherungsweise. (Anwendungen...)

Vereinfachungen:

1. Betrachte nicht die Eingabewerte selbst, sondern nur ihre Größe (z.B. Länge von Listen, Stellenanzahl bei Zahlen).
2. Vernachlässige die Häufigkeitsverteilung der Daten.
3. Vernachlässige konstanten Aufwand, also Aufwand, der unabhängig von den Eingabedaten entsteht.

Abstraktere Effizienzbegriffe: (2)

4. Betrachte nur das Wachstum von Laufzeit/Speicherplatz, vernachlässige konstante Faktoren (Abstraktion von der Leistung eines Rechners und den Implementierungseigenschaften einer Programmiersprache).
5. Betrachte nur obere und untere Schranken, oder Durschschnitt.

Beispiel: (Analyse der Laufzeit)

Wir analysieren eine prozedurale Implementierung des Algorithmus *Sortieren durch Auswahl* (*engl. selection sort*) .

Eingabe: Feld f von ganzen Zahlen.

Aufgabe: Sortiere das Feld f aufsteigend.

Algorithmische Idee:

- Bestimme eine Komponente mit Index `ixmin` von f, die ein minimales Element von f[0] ... f[f.length-1] enthält.
- Vertausche f[ixmin] und f[0].
- Sortiere dann den Bereich f[1] ... f[f.length-1] analog.

Iterative Fassung des Sortierens durch Auswahl:

```
void sortieren(int[] f ) {
   for( int i = 0; i<f.length-1; i++ ) {
      // bestimme Komponente mit kleinstem
      // Element in f[i] ... f[f.length-1]
      int ixmin = i;
      for( int j = i+1; j<f.length; j++ ) {
         if( f[j] < f[ixmin] ) {
            ixmin = j;
         }
      }
      // vertauschen der Elemente i und ixmin
      int temp = f[i];
      f[i] = f[ixmin];
      f[ixmin] = temp;
   }
}
```


Analyse der Laufzeit von `sortieren`:

In Abhängigkeit von der Größe N des Feldes schätzen wir die Anzahl $A(N)$ der Operationen/ Rechenschritte für den ungünstigsten Fall ab.

Eine Operation ist:

- ein Vergleich
- eine Zuweisung
- eine Addition/Subtraktion

Vereinfachende Annahme:

- Alle Operationen brauchen die gleiche Zeit.
- Andere Aspekte der Ausführung werden vernachlässigt (Speicherverwaltung, Sprünge)

Analyse der Laufzeit von `sortieren`: (2)

Aufwand B(i,N) der inneren Schleife:

$$B(i,N) \leq (2+1) + (N-i-1) * (2+2)$$

Aufwand $A(N)$ des gesamten Rumpfes für $N \geq 2$:

$$\begin{aligned} A(N) &= 3 + \sum_{i=0}^{N-2} (1 + B(i,N) + 3 + 2) \\ &\leq 3 + \sum_{i=1}^{N-1} (9 + (N-i) * 4) = 3 + (N-1) * 9 + 4 * \sum_{i=1}^{N-1} i \\ &= 9 * N - 6 + 2 * N * (N-1) = 2 * N^2 + 7 * N - 6 \end{aligned}$$

$A(0) = A(1) = 3$

O-Notation

Häufig interessiert man sich nur für die *Größenordnung* des Wachstums der ***Aufwandsfunktion*** $A(N)$, die den Aufwand an Speicher bzw. Zeit in Abhängigkeit von der ***Problemgröße*** N beschreibt.

Begriffsklärung: (obere Schranke)

Eine Funktion $f : Nat \to \mathbb{R}^+$ heißt ***obere Schranke*** einer Aufwandsfunktion A, wenn gilt:

Es gibt c, d in Nat, sodass für alle N in Nat gilt:

$$A(N) \leq c * f(N) + d$$

Man sagt auch, A wächst wie f bzw. ist von der Größenordnung f.

Die Menge aller Funktionen von der Größenordnung f bezeichnet man mit $O(f)$:

$$O(f) = \{g \mid \exists c, d \text{ in } Nat : \forall N \text{ in } Nat : g(N) \leq c * f(N) + d\}.$$

Bemerkung:

- Entsprechend definiert man auch untere Schranken.
- Vertiefung in ”Entwurf und Analyse von Algorithmen”
- Meist schreibt man $O(N)$ statt $O(\lambda N.N)$, $O(N^2)$ statt $O(\lambda N.N)^2$, $O(logN)$ statt $O(log)$, usw.

Beispiel: (Bestimmung oberer Schranken)

Der Zeitaufwand A vom Sortieren durch Auswahl ist

$$A(N) \leq 2 * N^2 + 7 * N - 6$$

und damit in $O(N^2)$; denn mit $c = 3$ und $d = 6$ gilt:

$$A(N) \leq 3 * N^2 + 6$$

Wichtige Komplexitätsklassen:

Kompl.klasse	Bezeichnung	Beispiel
$O(1)$	*konstant*	Hashverfahren
$O(log\ N)$	*logarithmisch*	binäre Suche in Bäumen
$O(N)$	*linear*	sequentielle Suche
$O(N * log\ N)$	*n log n*	gute Sortierverfahren
$O(N^2)$	*quadratisch*	einfache Sortierverfahren
$O(N^3)$	*kubisch*	Matrixmultiplikation
$O(2^N)$	*exponentiell*	Optimierungsverfahren

Algorithmen mit einem Aufwand in $O(N^k)$, $k \geq 2$, nennt man *polynomisch* oder *engl. polynomial*.

Diskussion der O-Notation:

- Die O-Notation liefert eine grobe Klassifikation.
- In der Praxis können konstante Faktoren und Programmiersprachen-spezifische Aspekte entscheidend sein (siehe Beispiel unten).
- Weitere Aspekte für die Effizienzbetrachtung:
  - Antwortzeiten interaktiver Softwaresysteme
  - Kommunikationszeiten

Vorgehen

Wir betrachten jetzt auch noch die prozeduralen Versionen von

- Sortieren durch Einfügen
- Quicksort

Vorgehen (2)

Bei allen Algorithmen gehen wir davon aus, dass die zu sortierenden Daten in einem Feld vorliegen, das verändert werden darf.

(Heapsort kommt später noch einmal vor.)

Alle Datensätze sind vom Typ `int[]`.

Bemerkung:

Beachte bei den folgenden Beispielen:

1. Die algorithmische Grundidee ist unabhängig vom verwendeten Programmierparadigma.
2. Die Verwendung von Feldern statt Listen kann die Komplexität ändern.

Sortieren durch Einfügen

Algorithmische Grundidee:

Sortiere zunächst eine Teilliste (Terminierungsfall: leere Liste). Füge dann die verbleibenden Elemente nacheinander in die bereits sortierte Teilliste ein.

Funktionale Fassung:

```
insertionsort :: [Int] -> [Int]
insertionsort []     = []
insertionsort (x:xl) = insert x (insertionsort xl)

insert :: Int -> [Int] -> [Int]
insert x []     =  [x]
insert x (y:yl) =  if (x <= y)
                   then  x : (y:yl)
                   else  y : (insert x yl)
```


Sortieren durch Einfügen (2)

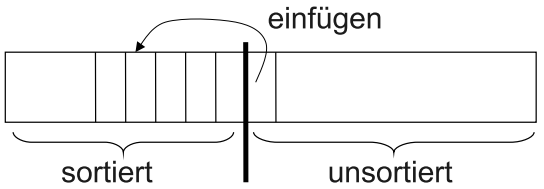
Nachteil der rekursiven Fassung:

- Aufwand durch Listendarstellung
- Aufwand durch rekursive Aufrufe

Sortieren durch Einfügen (3)

Ideen zur prozeduralen Realisierung:

- Speichere die Datensätze in einem Feld
- Realisiere das Einfügen durch schrittweises Verschieben (ausgehend vom größten Element)
- Eliminiere die Rekursion durch Beginn mit der einelementigen Liste in die nacheinander Elemente eingefügt werden.



Prozedurale Fassung in Java:

```
// Assume that the subarray f[0]...f[n-1] is sorted
// and that f[n] is empty. Insert the element x into
// the array so that the subarray f[0]...f[n] is sorted
// afterwards.

void insert(int[] f,int n,int x) {
    while(n>0) {
        if(x>f[n-1]) break;
        f[n] = f[n-1];
        n--;
    }
    f[n] = x;
}
```


Prozedurale Fassung in Java: (2)

```
// The array consists of a sorted portion, f[0]...f[i-1]
// and an unsorted portion, f[i]...f[f.length-1].
// Each iteration, take element i out of the array and then
// call insert to insert it into the portion f[0]...f[i-1]

void sort(int[] f) {
    int tmp;
    for(int i=0;i<f.length;i++)
        insert(f,i,f[i]);
}
```


Prozedurale Fassung in Java: (3)

```
void sort0(int[] f) {
    int tmp;
    for(int i=0;i<f.length;i++)  {
        // expanded inline: insert(f,i,f[i])
        int x = f[i];
        int n = i;
        while(n>0) {
            if(x>f[n-1]) break;
            f[n] = f[n-1];
            n--;
        }
        f[n] = x;
    }
}
```


Prozedurale Fassung in Java: (4)

```
// A small test with random numbers.

import java.util.Random;
random = new Random();

int[] a = new int[10];
for(int i=0;i<a.length;i++)
    a[i] = random.nextInt(10);

print(a);
sort(a);
print(a);
```

Laufzeitabschätzung

Wir betrachten die Anzahl der Schlüsselvergleiche C und der Zuweisungen M von Datensätzen in Abhängigkeit von der Anzahl N der Datensätze.

Günstigster Fall:

Liste ist bereits aufsteigend sortiert.

→ pro Schleifendurchlauf ein Schlüsselvergleich

→ pro Durchlauf zwei Datensatzzuweisungen

Schlüsselvergleiche:

$$C_{min}(N) = N - 1;$$

Datensatzzuweisungen:

$$M_{min}(N) = 2 * (N - 1);$$

Laufzeitabschätzung (2)

Ungünstigster Fall:
Liste ist absteigend sortiert.

→ pro Schleifendurchlauf i Schlüsselvergleiche

→ pro Durchlauf $(i + 2)$ Datensatzzuweisungen

Schlüsselvergleiche: $C_{max}(N) = \sum_{i=1}^{N-1} i \in O(N)^2$

Datensatzzuweisungen: $M_{max}(N) = \sum_{i=1}^{N-1} (i + 2) \in O(N^2)$

Durchschnitt:
Im Durchschnitt ergibt sich quadratische Komplexität.

Quicksort

Algorithmische Grundidee:

- Wähle einen beliebigen Datensatz mit Schlüssel k aus, das sogenannte Pivotelement.
- Teile die Liste in zwei Teile:
  - 1. Teil enthält alle Datensätze mit Schlüsseln $< k$
  - 2. Teil enthält die Datensätze mit Schlüsseln $\geq k$
- Wende quicksort rekursiv auf die Teillisten an.
- Hänge die resultierenden Listen und das Pivotelement zusammen.

Quicksort (2)

Funktionale Fassung:

```
qsort :: [Int] -> [Int]
qsort []       = []
qsort (p:rest) =
  let lo = filter (<p) rest
      hi = filter (>=p) rest
  in (qsort lo) ++ [p] ++ (qsort hi)
```

Quicksort (3)

Umsetzung in prozedurale Fassung:

Quicksort (4)

Prozedurale Fassung in Java:

Achtung!

- Im Prinzip ganz analog zur funktionalen Fassung, aber...
- Wir benutzen zum Speichern nur das Eingabefeld.
- Die Listen “lo” und “hi” von oben werden auf Unterbereiche abgebildet.

Quicksort (5)

```
void swap(int[] f,int i,int j) {
    int temp = f[i];
    f[i] = f[j];
    f[j] = temp;
}
```


Quicksort (6)

```
void quicksort(int[] f,int start,int end) {
    if(end-start<=1) return;
    int middle = split(f,start,end);
    quicksort(f,start,middle);
    quicksort(f,middle+1,end);
}
```

- `f[middle] = q = pivot`
- `f[start]...f[middle-1] = lo`
- `f[middle+1]...f[end-1] = hi`

Quicksort (7)

```
void quicksort(int[] f) {
    quicksort(f,0,f.length);
}
```


Quicksort (8)

Algorithmische Idee für `split`:

- benutze das letzte Element des Bereiches als Pivot
- für die restlichen Elemente...
- gehe von links und rechts nach innen, bis...
- wir zwei Element finden, die getauscht werden müssen...
- oder wir uns überkreuzen
- am Ende bewege das Pivot-Element in die Mitte
- gib die Position des Pivot-Elements zurück

Quicksort (9)

```
int split(int[] f,int start,int end) {
    int pivot = f[end-1];
    int i = start;
    int j = end-2;
    for(;;) {
        while(f[i]<pivot) i++;
        while(j>=i && f[j]>=pivot) --j;
        if(i>j) break;
        swap(f,i,j);
    }
    swap(f,i,end-1);
    return i;
}
```


Quicksort (10)

```
[13 14 6 (27) 34 76 42 29 73 31 89 59 52 98 38 64 44 52 51 77]
[(6) 14 13] 27 34 76 42 29 73 31 89 59 52 98 38 64 44 52 51 77
6 [(13) 14] 27 34 76 42 29 73 31 89 59 52 98 38 64 44 52 51 77
6 13 14 27 [34 76 42 29 73 31 51 59 52 52 38 64 44 (77) 89 98]
6 13 14 27 [34 38 42 29 31 (44) 51 59 52 52 76 64 73] 77 89 98
6 13 14 27 [29 (31) 42 34 38] 44 51 59 52 52 76 64 73 77 89 98
6 13 14 27 29 31 [34 (38) 42] 44 51 59 52 52 76 64 73 77 89 98
6 13 14 27 29 31 34 38 42 44 [51 59 52 52 64 (73) 76] 77 89 98
6 13 14 27 29 31 34 38 42 44 [51 59 52 52 (64)] 73 76 77 89 98
6 13 14 27 29 31 34 38 42 44 [51 (52) 52 59] 64 73 76 77 89 98
6 13 14 27 29 31 34 38 42 44 51 52 [52 (59)] 64 73 76 77 89 98
6 13 14 27 29 31 34 38 42 44 51 52 52 59 64 73 76 77 [89 (98)]
```

Grobe Laufzeitabschätzung von Quicksort:

Seien C, M und N definiert wie auf Folie 723.

Vorüberlegung:

Betrachte die Ebenen gleicher Tiefe im Aufrufbaum von quicksort. Das Zerlegen aller Teillisten auf einer Ebene verursacht schlimmstenfalls linearen Aufwand:

$$\begin{aligned} C_{split}(N) &= O(N) \\ M_{split}(N) &= O(N) \end{aligned}$$

Grobe Laufzeitabschätzung von Quicksort: (2)

Ungünstigster Fall:

Beim Zerlegen der Listen ist jeweils eine der Teillisten leer. Dann hat der Aufrufbaum die Tiefe N, also gilt:

$$\begin{aligned} C_{max}(N) &= N * C_{split}(N) = O(N^2) \\ M_{max}(N) &= N * M_{split}(N) = O(N^2) \end{aligned}$$

Grobe Laufzeitabschätzung von Quicksort: (3)

Günstigster Fall:

Beim Zerlegen der Liste entstehen jeweils zwei etwa gleich große Teillisten. Dann hat der Aufrufbaum die Tiefe $logN$, also gilt:

$$\begin{aligned} C_{min}(N) &= log\, N * C_{split}(N) = O(N\, log\, N) \\ M_{min}(N) &= log\, N * M_{split}(N) = O(N\, log\, N) \end{aligned}$$

Bemerkung:

- Die mittlere Laufzeit von Quicksort ist auch von der Größenordnung $O(N\, log\, N)$ (siehe Ottmann, Widmayer: Abschn. 2.2).
- Die vorgestellte Quicksort-Fassung arbeitet schlecht auf schon sortierten Listen.
- Verbesserungen der vorgestellten Variante ist möglich durch geeignetere Auswahl des Pivotelementes und durch Elimination der Rekursion.

Bemerkung: (2)

```
int split(int[] f,int start,int end) {
    int m = random.nextInt(end-start)+start;
    swap(f,m,end-1);
    int pivot = f[end-1];
    int i = start;
    int j = end-2;
    for(;;) {
        while(f[i]<pivot) i++;
        while(j>=i && f[j]>=pivot) --j;
        if(i>j) break;
        swap(f,i,j);
    }
    swap(f,i,end-1);
    return i;
}
```


Unterabschnitt 4.3.2

Algorithmenklassen & -entwicklung

Algorithmenklassen & -entwicklung

Dieser Abschnitt skizziert:

- wichtige weitere Problem- und Algorithmenklassen
- einen Weg zur Entwicklung von Algorithmen anhand eines Beispiels

Problem- und Algorithmenklassen

Neben dem klassischen Bereichen des Sortierens und Suchens von Datensätzen gibt es eine Vielzahl von Algorithmen für unterschiedliche Aufgaben- und Problembereiche.

Beispiele: (Algorithmische Probleme)

- Optimaler Einsatz der Flugzeugflotte einer Fluggesellschaft.
- Ermittlung der Schnittfläche zweier Flächen gegeben durch ihre Punkte
- Erfüllbarkeit/Allgemeingültigkeit logischer Formeln.
- Auffinden aller Web-Seiten, die eine Menge von Schlüsselwörter enthalten.

Klassifikation von Algorithmen

Algorithmen lassen sich nach unterschiedlichen Aspekten klassifizieren, zum Beispiel gemäß:

- verwendeter Datenstrukturen
- algorithmischer Kriterien (z.B. Art der Parallelität)
- spezieller Aufgabenbereiche

Klassifikation nach Datenstrukturen

Mengen, Listen, Warteschlangen, etc.:

Ziele:

Effiziente Speicherung und effiziente Operationen zum Einfügen, Suchen und Löschen.

Zeichenreihen, Textsuche:

Ziele:

Effiziente Suche von Wort- oder Textmustern in Texten.

Beispiel:

Finde alle Vorkommen von "S%Haffner" in den letzten 5 Jahrgängen der Frankfurter Allgemeinen Zeitung.

Begriffsklärung: (Graph)

Ein ***gerichteter Graph*** (engl. ***digraph***) $G = (V, E)$ besteht aus

- einer endlichen Menge V von ***Knoten*** (engl. ***vertices***)
- einer Menge $E \subseteq V \times V$ von ***Kanten*** (engl. ***edges***)

Ist (va, ve) eine Kante, dann nennt man

- va den *Anfangs-* oder *Startknoten* oder die *Quelle*
- ve den *Endknoten* oder das *Ziel*

der Kante. ve heißt von va *direkt erreichbar* und *Nachfolger* von va; va *Vorgänger* von ve.

Graphen bieten für eine große Klasse von Problemen ein geeignetes abstraktes Modell.

Beispiele:

- Was ist die beste Verbindung von A nach B?
- Wie transportiere ich Waren von mehreren Anbietern am billigsten zu mehreren Nachfragern?
- Wie gestalte ich einen Arbeitsablauf mit mehreren Maschinen und Arbeitskräften optimal?
- Welche Wassermenge kann maximal durch die Kanalisation von KL abgeleitet werden?

Speicherdarstellungen von Graphen:

Sei $G = (V, E)$ ein gerichteter Graph, $V = \{1, ..., n\}$. G lässt sich speichern als:

- Adjazenzmatrix: boolesche $n \times n$-Matrix, wobei das Element (x, y) true ist genau dann, wenn es in G eine Kante von x nach y gibt.
- Adjazenzlisten: Speichere für jeden Knoten die Liste der durch eine Kante erreichbaren Knoten.

Klassifikation nach algorithmischen Kriterien

Wir haben bisher nur sequentielle Algorithmen betrachtet, deren Daten alle im Hauptspeicher Platz finden. In der Praxis sind häufig komplexere Anforderungen zu berücksichtigen:

- Daten auf anderen Speichermedien ohne wahlfreies Zugriffsverhalten
- Parallelisierung für gegebene Rechner, um akzeptable Antwortzeiten zu erhalten bzw. große Datenmengen rechnen zu können.
- Arbeiten mit verteilten, sich dynamisch entwickelnden Daten

Klassifikation nach Aufgabenbereichen

Viele Teilgebiete der Informatik und anderer Fächer haben mittlerweile für ihre speziellen Aufgaben umfangreiches algorithmisches Wissen erarbeitet.

Zwei Beispiele:

- Algorithmische Geometrie
- Übersetzertechnik/Compilerbau

Beispiele: (Algorithmische Geometrie)

Beispielproblem:

Gegeben eine Menge von Rechtecken; ermittle alle Paare von Rechtecken, die sich schneiden.

Anwendungsbereiche:

- Computergraphik, Visualisierung
- Geometrische Modellierung, CAD
- Schaltungsentwurf
- Wegeplanung von Robotern

Beispiele: (Übersetzertechnik)

Aufgabenbereiche:

- Parsen gemäß einer kontextfreien Grammatik:
  - Eingabe: Zeichenreihe (Programm)
  - Ausgabe: Syntaxbaum
- Optimierende Übersetzung, zum Beispiel:
  - Konstante Ausdrücke zur Übersetzungszeit berechnen
  - Prozeduraufrufe durch ihre Rümpfe ersetzen
  - Speicherbedarf verringern

Beispiele: (Übersetzertechnik) (2)

Konstantenfaltung:
Übersetze das Programmfragment:

```
int a = 7;
int b = a * 3;
int c = a + b;
```

so als hätte der Programmierer geschrieben:

```
int a = 7;
int b = 21;
int c = 28;
```


Beispiele: (Übersetzertechnik) (3)

Escape-Analyse:
Ermittele die Objekte, die auf dem Keller alloziert werden können, da ihre Referenzen den Methodenaufruf, der sie erzeugt hat, nicht verlassen.

Beispielfragment:

```
void m( String s ) {
  String t = "" + s ;   // neuer Verbund
  t = doSomething(t);   // Modifikation von t
  println(t);
}
```

Der/das von *t* referenzierte Verbund/Objekt könnte auf dem Keller verwaltet werden.

Ziel:

Entlastung der Speicherbereinigung.

Algorithmenentwicklung

Abschließend zu 4.3 betrachten wir wichtige Phasen der Algorithmenentwicklung an einem Beispiel.

Phasen der Algorithmenentwicklung:

1. Problemabstraktion und -formulierung
2. Entwickeln einer algorithmischen Idee
3. Ermitteln wichtiger Eigenschaften des Problems
4. Grobentwurf eines Algorithmus‘ mit Abstützung auf existierende Teillösungen
5. Entwickeln bzw. Festlegen der Datenstrukturen
6. Ausarbeiten des Algorithmus

Algorithmenentwicklung an einem Beispiel:

Wir erläutern die Phasen der Algorithmenentwicklung an einem Beispiel (vgl. Phasen der Softwareentwicklung).

0. Problem:

Routenplaner für Fahrradfahrer in einer Großstadt:
Wie ist die beste Verbindung zwischen zwei Straßenkreuzungen?

Algorithmenentwicklung an einem Beispiel: (2)

1. Problemabstraktion und -formulierung:

Modelliere die Straßen und Wege durch einen gerichteten Graphen mit bewerteten Kanten:

- Straßenkreuzungen entsprechen Knoten
- Kante entspricht einer *direkten* Straßenverbindung zwischen Kreuzungen A und B, die von A nach B befahrbar ist (ggf. auch Kante für umgekehrte Richtung).
- Jede Kante bekommt als Bewertung die Zeit in Sekunden, die man im Durchschnitt für den Weg von A nach B braucht.

Algorithmenentwicklung an einem Beispiel: (3)

Die Bewertung ist eine Funktion $c : E \to \mathbb{R}^+$
Bewertete gerichtete Graphen nennt man *Distanzgraphen*.

Damit lässt sich das Problem wie folgt formulieren:

- Gegeben ein Distanzgraph, der die Straßenverbindungen modelliert, sowie zwei Knoten s und z.
- Gesucht ist ein Weg $s, v_1, \ldots, v_n, z$ mit minimaler Länge lg:

$$lg = c((s, v_1)) + c((v_1, v_2)) + \cdots + c((v_n, z))$$

Algorithmenentwicklung an einem Beispiel: (4)

2. Entwickeln einer algorithmischen Idee:

Eine verbreitete *Strategie* zur Algorithmenentwicklung versucht ein Problem auf ähnlich geartete Teilprobleme zu reduzieren. Hier:

Reduziere die Suche des kürzesten Wegs von s nach z auf kürzeste Wege zwischen anderen Knotenpaaren.

Ansatz:

(a) Errechne schrittweise Knotenmengen B, sodass der kürzeste Weg von s zu allen Knoten von B bekannt ist. Anfangs ist $B = \{s\}$.

Algorithmenentwicklung an einem Beispiel: (5)

(b) Betrachte alle Knoten R außerhalb von B, die von Knoten in B direkt erreichbar sind. (R wird meist der Rand von B genannt.)

(c) Bestimme den kürzesten Weg zu einem Knoten in R und erweitere B entsprechend.

Unter welchen Bedingungen lassen sich (a)-(c) algorithmisch lösen? Was sind die Einzelschritte?

Sei $r \in R$ ein Randknoten und $w_1, \ldots, w_r \in B$ alle Knoten mit $(w, r) \in E$. Lässt sich damit der kürzeste Weg von s nach r bestimmen und seine Länge $spl(s, r)$?

Algorithmenentwicklung an einem Beispiel: (6)

3. Ermitteln wichtiger Eigenschaften des Problems:

Um unseren Ansatz umsetzen zu können, brauchen wir eine Eigenschaft, die es uns ermöglicht, B schrittweise um Randknoten zu erweitern.

Verschärfung des Ansatzes:

1. Bestimme für jeden Knoten r des Randes einen Vorgänger w_r in B, so dass $d(r) = spl(s, w_r) + c((w_r, r))$ minimal ist.
2. Wähle unter allen Knoten r von R denjenigen mit minimalem $d(r)$ aus. Sei dieser Knoten mit p bezeichnet. Erweitere B um p.

Algorithmenentwicklung an einem Beispiel: (7)

Behauptung:

$spl(s, w_p) + c((w_p, p)) = spl(s, p)$, d.h. der kürzeste Weg von s zu p wurde gefunden.

Beweis:

Mit Induktion über den kürzesten Weg (siehe Vorlesung).

Algorithmenentwicklung an einem Beispiel: (8)

4. Grobentwurf eines Algorithmus‘ mit Abstützung auf existierende Teillösungen:

Wir setzen die obigen Ansätze in einen Grobentwurf um, der auf Dijkstra zurückgeht (vgl. Ottmann, Widmayer: 8.5.1):

Jeder Knoten erhält drei zusätzliche Komponenten:

pred:	Vorgänger auf dem kürzesten "Rückweg" zu s.
dist:	die kürzeste bisher ermittelte Entfernung zu s.
inB:	Ist genau dann true, wenn Knoten in der Menge ist, für die der kürzeste Weg bekannt ist.

Algorithmenentwicklung an einem Beispiel: (9)

Algorithmus:
Berechnet kürzeste Wege in bewerteten Graphen $G = (V, E)$ mit Bewertungsfunktion c und Startknoten ist s.

// Initialisieren der Knoten und von B:

```
for all v ∈ V\{s} do {
  v.pred = null;
  v.dist = ∞;
  v.inB  = false ;
}
s.pred = s ;
s.dist = 0 ;
s.inB  = true ;
```


Algorithmenentwicklung an einem Beispiel: (10)

// Initialisieren des Randes R:
$R = \emptyset$;
// R initialisieren, d.h. die Nachfolger von s eintragen:

```
ergaenzeRand(s,R);
```

// Auswählen von Knoten aus R und R ergänzen:

```
while R != ∅ do {
  // waehle naechst gelegenen Randknoten aus:
  waehle v ∈ R mit v.dist minimal ;
  entferne v aus R ;
  v.inB  = true ;
  ergaenzeRand(v,R);
}
```


Algorithmenentwicklung an einem Beispiel: (11)

```
where
procedure ergaenzeRand( v, R ) {
  for all (v,w)∈E do {
    if  not w.inB  and
        ( v.dist + c((v,w)) < w.dist ){
      // w ist (kuerzer) ueber v erreichbar
      w.pred = v ;
      w.dist = v.dist + c((v,w)) ;
      R = R ∪ {w} ;
    }
  }
}
```

Algorithmenentwicklung an einem Beispiel: (12)

5. Entwickeln bzw. Festlegen der Datenstrukturen:
In dieser Phase ist zu entscheiden, welche Datenstrukturen für die Realisierung

- des Graphen und
- des Randes

benutzt werden sollen.

Benötigte Operationen auf der Graphdatenstruktur:

- Iterieren über die Knotenmenge
- Iterieren über die Kantenmenge zu einem Knoten
- Bewertung der Kanten auslesen

Algorithmenentwicklung an einem Beispiel: (13)

Benötigte Operationen auf dem Rand:

- Rand als leer initialisieren
- Prüfen, ob Rand leer ist
- Wählen des Knotens mit minimaler Entfernung
- Entfernen eines Knotens aus dem Rand
- Knoten zum Rand hinzufügen bzw. Knoten im Rand modifizieren

Als Graphdatenstruktur könnten z.B. Adjazenzlisten verwendet werden. Der Rand kann als Heap realisiert werden.

Algorithmenentwicklung an einem Beispiel: (14)

6. Ausarbeiten des Algorithmus:

Aus den Entscheidungen der 5. Phase entsteht ein Feinentwurf, der präzise zu formulieren und, wo möglich, zu optimieren ist.

Schließlich kann der Feinentwurf ausprogrammiert und getestet werden.

Bemerkung:

- Bis auf den letzten Schritt sind alle Phasen der Algorithmenentwicklung unabhängig von Programmiersprachen.

  Üblicherweise rechnet man die Algorithmenimplementierung auch nicht mehr zum Bereich Algorithmen und Datenstrukturen.
- Softwareentwicklung im Allg. hat viele Parallelen zur Algorithmenentwicklung. Auch hier hat die Programmierung eine nachgeordnete Bedeutung.

  Dafür liegt der Schwerpunkt nicht so sehr auf der Lösung gut eingrenzbarer schwieriger Probleme, sondern stärker auf der Bewältigung der vielen Aspekte und des Umfangs der Aufgabenstellung.