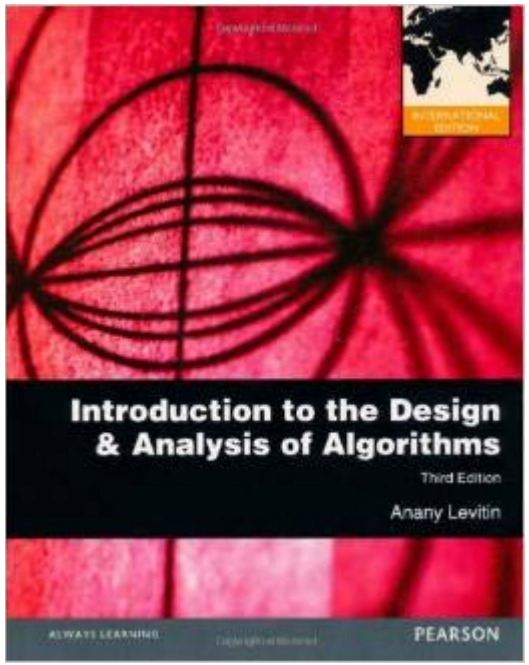
Algorithmen & Datenstrukturen

Brute Force-Algorithmen

Literaturangaben

Diese Lerneinheit basiert größtenteils auf dem Buch „The Design and Analysis of Algorithms“ von Anany Levitin.



In dieser Einheit behandelte Kapitel:
3 Brute Force
3.1 Selection Sort and Bubble Sort
3.2 Sequential Search and Brute-Force String Matching
3.3 Closest-Pair and Convex-Hull Problems by Brute Force
3.4 Exhaustive Search
3.5 Depth-First Search and Breadth-First Search

Brute Force-Ansatz

- Typischerweise relativ **schlichter** Ansatz, **direkt basierend** auf:
  - der Problembeschreibung
  - der Definition der beteiligten Konzepte

- **Beispiele**
  - Berechnung von a^n ($a > 0$, n nicht-negative Ganzzahl)
  - Berechnung von n!
  - Matrizenmultiplikation
  - Schlüsselsuche in einer gegebenen Liste von Werten

Stärken und Schwächen

- **Stärken**
  - Breites Einsatzfeld
  - Einfach zu entwerfen
  - Liefert passable Lösungen für einige wichtige Problemtypen (etwa Matrixmultiplikation, Sortieren, Suchen, Zeichenkettenvergleich)
  - Vergleichsmaßstab für komplexere Algorithmen
- **Schwächen**
  - Führt selten zu den effizientesten Algorithmen
  - Brute Force-Algorithmen für einige Probleme unakzeptabel langsam
  - Nicht so kreativ/lehrreich wie andere Designtechniken

Brute Force Suche: Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Zeiteffizienz? **Speicherplatzbedarf?** **Stabilität?**

Brute Force Suche: Bubble Sort

ALGORITHM *BubbleSort*($A[0..n-1]$)
// Sorts a given array by bubble sort
// Input: An array $A[0..n-1]$ of orderable elements
// Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Zeiteffizienz? **Speicherplatzbedarf?** **Stabilität?**

Brute Force: Zeichenketten-Vergleich

Aufgabe: Finde in einem **Text** eine bestimmtes **Suchmuster**

- **Suchmuster**: Zeichenkette der Länge **m** nach der gesucht wird
- **Text**: (längere) Zeichenkette der Länge **n** in der gesucht wird

Brute Force-Algorithmus:

- **Schritt 1**: Richte das Suchmuster am Textanfang aus
- **Schritt 2**: Beginne links, vergleiche alle Zeichen des Musters mit den korrespondierenden Zeichen des Texts bis:
  - alle Zeichen des Musters gefunden wurden (→ erfolgreiche Suche) oder
  - ein Zeichen nicht übereinstimmt
- **Schritt 3**: Falls das Suchmuster nicht gefunden wurde, verschiebe das Muster um eine Position nach rechts und wiederhole Schritt 2

Beispiele: Brute Force-Zeichenkettenvergleich

1. Suchmuster: 001011
   Text: 10010101101001100101111010

2. Pattern: happy
   Text: It is never too late to have a happy childhood.

Pseudocode und Effizienz

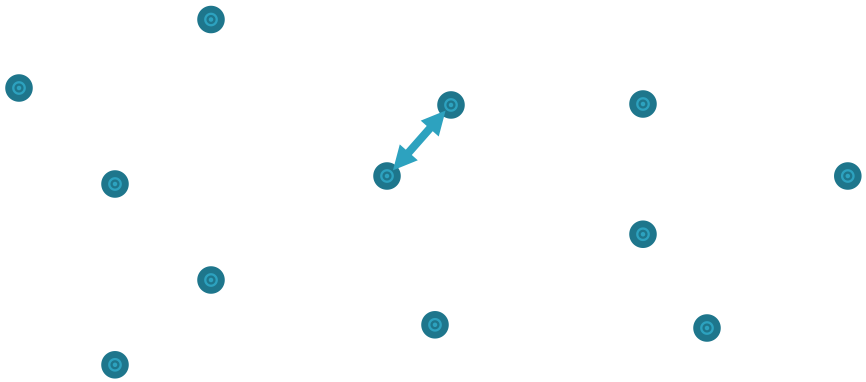
ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Effizienz?

Problem des dichtesten Punktepaars (closest pair problem)

- Finde in einer Menge von n Punkten die beiden Punkte mit dem geringsten Abstand



- **Brute Force Lösung:** Berechne den Abstand zwischen jedem Punktepaar und bestimme das Minimum

Dichtestes Punktepaar: Pseudocode

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices *index*1 and *index*2 of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return *index*1, *index*2

Erschöpfende Suche

Suche nach einem **Element mit speziellen Eigenschaften** aus einer **kombinatorischen Menge** von Objekten. z. B.:

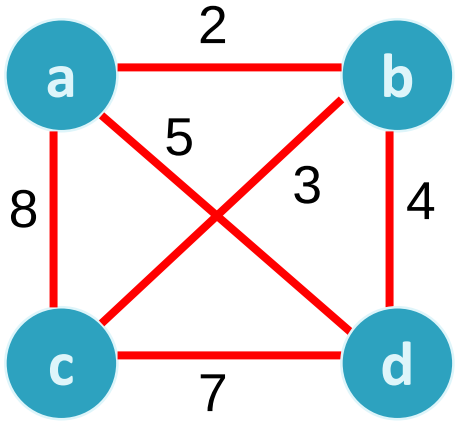
- Permutationen
- Kombinationen
- Menge aller Teilmengen

Methode:

- Generiere systematisch **alle in Frage kommenden Elemente** (= potentielle Lösungen)
- Betrachte **alle** potentiellen Lösungen nacheinander
  - Verwerfe ungültige Lösungen
  - Bei Optimierungsproblemen: Merke dir die bisher beste gültige Lösung
- Wenn alle Elemente untersucht wurden, gebe die **Lösung(en)** aus

Beispiel: Problem des Handelsreisenden

- **Gegeben**: n Städte und die Abstände zwischen allen Städten
- **Gesucht**: Die kürzeste Rundreise durch alle Städte
- **Englischer Name**: Traveling Salesman Problem (TSP)
- **Alternative Formulierung**: Finde den kürzesten Hamiltonkreis eines vollständigen, gewichteten Graphen



Erschöpfende Suche beim TSP

Rundreise	Reiselänge
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Weitere Rundreisen?

Weniger Rundreisen?

Effizienz?

Beispiel: Rucksackproblem

Gegeben: n Gegenstände mit

- den Gewichten (weights) $w_1, w_2, \ldots, w_n$
- den Werten (values) $v_1, v_2, \ldots, v_n$
- ein Rucksack mit der Gewichtskapazität W

Gesucht: Wertvollste Teilmenge, die noch in den Rucksack passt

Englischer Name: Knapsack Problem

Beispiel: Rucksack mit der Kapazität W = 16

Gegenstand	Gewicht	Wert
1	2 kg	20 €
2	5 kg	30 €
3	10 kg	50 €
4	5 kg	10 €

Erschöpfende Suche beim Rucksackproblem

Teilmenge	Gesamtgewicht	Gesamtwert
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	ungültig
{1, 2, 4}	12	60
{1, 3, 4}	17	ungültig
{2, 3, 4}	20	ungültig
{1, 2, 3, 4}	22	ungültig

Beispiel: Zuordnungsproblem

Gegeben:

- **n Personen** und **n Tätigkeiten**
- Alle potentiellen (Arbeits-) **Kosten C(i, j)**: Kosten, die entstehen, wenn Person i Tätigkeit j ausführt

Gesucht: Finde eine **Zuordnung** (eine Tätigkeit pro Person), die die **Kosten minimiert**

Englischer Name: Assignment Problem

Ablauf:

- Erzeuge alle legalen Zuordnungen
- Berechne jeweils die Kosten
- Wähle die preiswerteste Zuordnung

Erschöpfende Suche Zuordnungsproblem

C(i, j)	Tätigkeit 1	Tätigkeit 2	Tätigkeit 3	Tätigkeit 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Kostenmatrix

Liste potentieller Lösungen

Zuordnung (Spaltennummern)	Gesamtkosten
1, 2, 3, 4	9 + 4 + 1+ 4 = 18
1, 2, 4, 3	9 + 4 + 8 + 9 = 30
1, 3, 2, 4	9 + 3 + 8 + 4 = 24
1, 3, 4, 2	9 + 3 + 8 + 6 = 26
1, 4, 2, 3	9 + 7 + 8 + 9 = 33
1, 4, 3, 2	9 + 7 + 1 + 6 = 23
usw.	usw.

Erschöpfende Suche Abschließende Bemerkungen

- Erschöpfende Suche ist aufgrund des hohen Zeitaufwands nur für **sehr kleine Problemgrößen** anwendbar
- Bei manchen Problemen gibt es wesentlich **bessere Alternativen**:
  - Eulerkreise
  - Kürzeste Wege
  - Minimalgerüste
  - Zuordnungsproblem
- In vielen anderen Fällen ist erschöpfende Suche (oder eine Variation davon) allerdings der **einzig bekannte Weg**, eine **genaue Lösung** zu finden

Beispiel: Graphen-Traversierung

- Zur Lösung vieler Probleme müssen **alle Knoten** (bzw. Kanten) **eines Graphen systematisch verarbeitet** werden
- Grundlegende Graphen-Traversierungsalgorithmen:
  - **Depth-first search (DFS)**
    (deutsch: Tiefensuche)
  - **Breadth-first search (BFS)**
    (deutsch: Breitensuche)

Beispiel: Tiefensuche/ Depth-first Search (DFS)

- Besuche alle Knoten eines Graphen
  - Bewege dich **vom letzten besuchten Knoten** zu einem weiteren, noch nicht besuchten Knoten
  - Falls es keinen benachbarten, unbesuchten Knoten gibt, kehre zum vorigen Knoten zurück (**backtracking**)
- Benutzt einen **Stack (Kellerspeicher)**
  - Ein Knoten wird auf den Stack gepackt, wenn er **zum ersten mal** besucht wird
  - Ein Knoten wird vom Stack gelöscht, wenn er **keine unbesuchten Nachbarn mehr** besitzt
- Definiert „**Baumstruktur(en)**“ im Graphen
  - Alle Kanten, die zum Aufsuchen eines unbesuchten Knotens verwendet werden

Pseudocode der Tiefensuche (I)

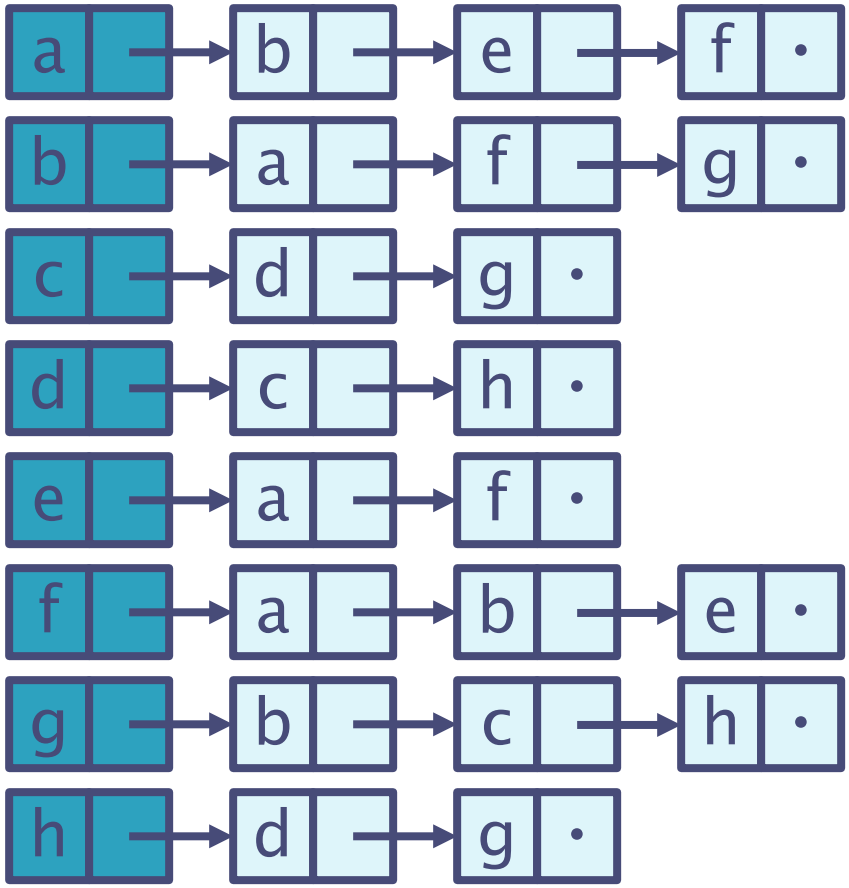
ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

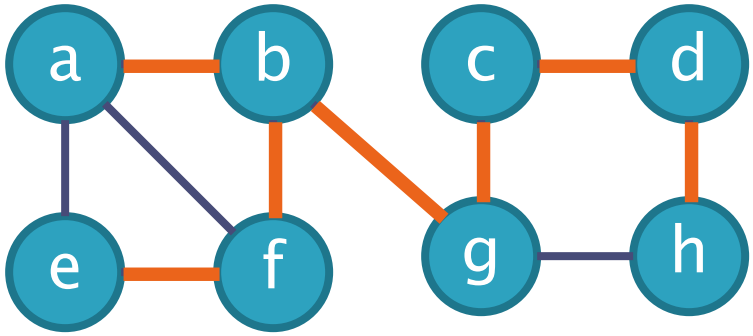
Pseudocode der Tiefensuche (II)

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Beispiel: DFS-Traversierung eines ungerichteten Graphen



Adjazenzliste



Graph

DFS-Traversierungsstack

DFS-Baum:

Anmerkungen zur Tiefensuche

- Tiefensuche kann mit **Adjazenzlisten** und **Adjazenzmatrizen** implementiert werden
  - Adjazenzlisten: $\Theta(|V|+|E|)$
  - Adjazenzmatrix: $\Theta(|V|^2)$
- Es entstehen **zwei unterschiedliche Knotenordnungen**
  - Ordnung, in der die Knoten **entdeckt** werden
  - Ordnung, in der die Knoten „**Sackgassen**“ werden
- **Anwendungen**
  - Prüfung auf **Zusammenhang**, **Zusammenhangskomponenten**
  - Prüfung auf **Zyklenfreiheit**
  - Finden von **Artikulationspunkten**
  - **Durchsuchung des Zustandsraums** nach möglichen Lösungen

Beispiel: Breitensuche/ Breadth-first Search (BFS)

- Besuche alle Knoten eines Graphen
  - Bearbeite aktuellen Knoten und ermittle zunächst **alle noch nicht besuchten Nachbarn**
  - Verfahre so **der Reihe nach** mit allen neu entdeckten Nachbarn
- Benutzt eine **Queue (Warteschlange)**
  - Ein Knoten wird in die Queue eingereiht, wenn er **zum ersten mal** entdeckt wird
  - Ein Knoten wird aus der Queue gelöscht, wenn **alle seine Nachbarn überprüft** wurden
- Definiert „**Baumstruktur(en)**“ im Graphen
  - Alle Kanten, die beim Entdecken eines unbesuchten Knotens verwendet werden

Pseudocode der Breitensuche (I)

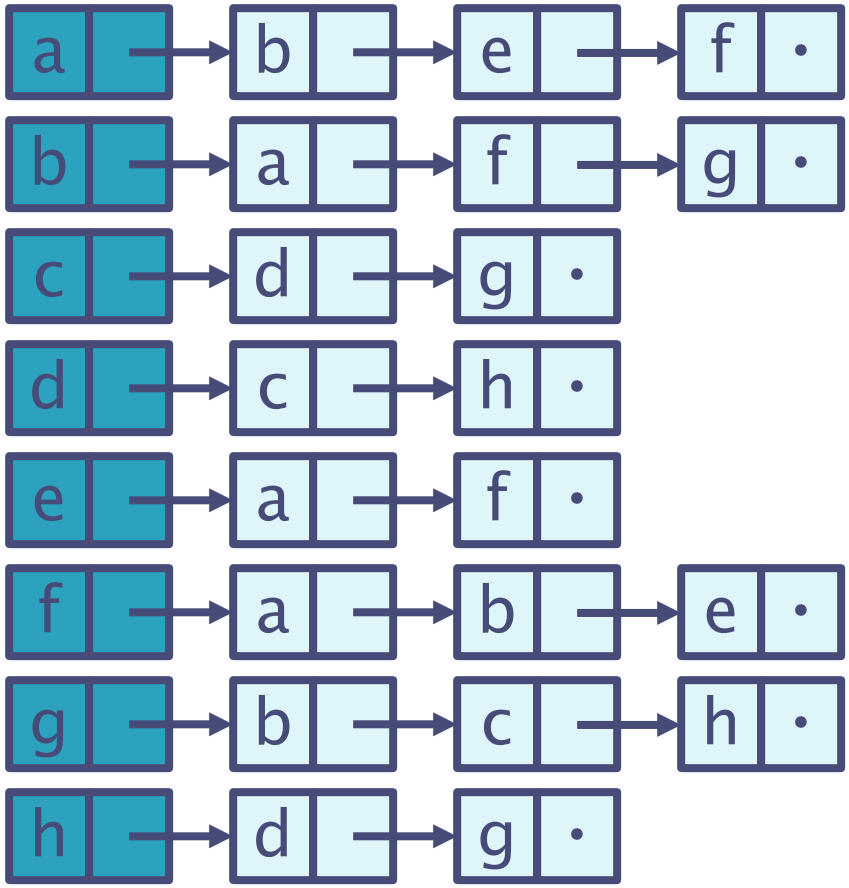
ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
 mark each vertex in V with 0 as a mark of being "unvisited"
 count ← 0
 for each vertex v in V do
     if v is marked with 0
       bfs(v)
```

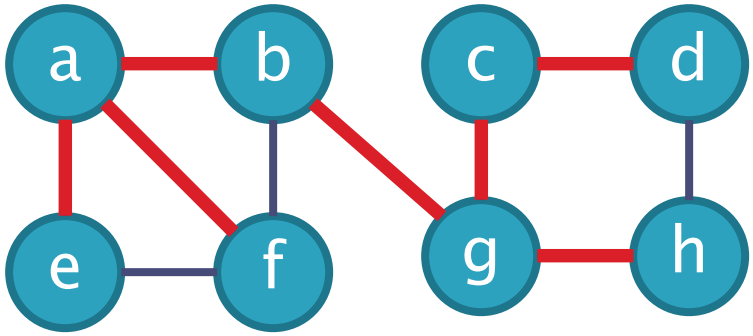
Pseudocode der Breitensuche (II)

```
bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Beispiel: BFS-Traversierung eines ungerichteten Graphen



Adjazenzliste



Graph

BFS-Traversierungsqueue

BFS-Baum:

Anmerkungen zur Breitensuche

- Breitensuche kann mit **Adjazenzlisten** und **Adjazenzmatrizen** implementiert werden
  - Adjazenzlisten: $\Theta(|V|+|E|)$
  - Adjazenzmatrix: $\Theta(|V|^2)$

Äquivalent zur Tiefensuche

- Es entsteht nur **eine Knotenordnungen**
  - Einfüge- und Lösch-Ordnung in der Queue sind identisch
- **Anwendungen**
  - Wie bei der Tiefensuche
  - **Zusätzlich**: Finde die **kürzesten Pfade** (d. h. die Pfade mit den wenigsten Kanten) von einem bestimmten Knoten zu allen anderen