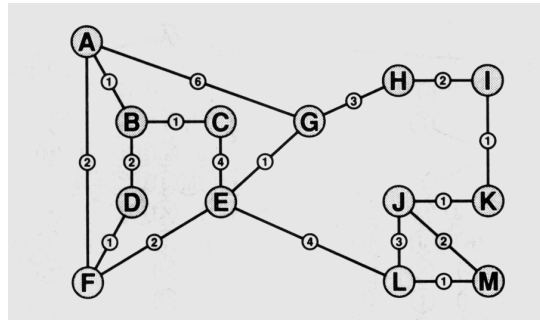


Erschöpfendes Suchen und Backtracking



Inhalt

- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



- Polynomialzeitbeschränkte Algorithmen (Klasse der P-Probleme)
 - Polynomialzeitbeschränkte Algorithmen sind Algorithmen, die eine Komplexität haben, die mit einem Polynom über die Problemgröße n angegeben werden kann
 - bei denen hält sich die Laufzeit noch in Grenzen
 - Beispiel: $O(n^2)$, $O(n^3)$, ...
- Nicht-polynomialzeit-beschränkte Algorithmen (NP-harte Probleme)
 - diese explodieren mit der Problemgröße
 - Beispiel: Algorithmuskomplexität 2^n oder $n!$
- NP-Vollständigkeit bezeichnet die Klasse aller NP-Probleme:
 - Es wurde bewiesen, dass wenn man einen Algorithmus für ein NP-vollständiges Problem findet, damit alle NP-vollständigen Probleme lösen kann (Satz von Cook)



Darstellung polynomiale versus nicht-polynomiale Laufzeitkomplexität

n	n^2	n^4	2^{**n}	4^{**n}	$n!$
2	4	16	4	16	2
3	9	81	8	64	6
4	16	256	16	256	24
5	25	625	32	1.024	120
6	36	1.296	64	4.096	720
7	49	2.401	128	16.384	5.040
8	64	4.096	256	65.536	40.320
9	81	6.561	512	262.144	362.880
10	100	10.000	1.024	1.048.576	3.628.800
11	121	14.641	2.048	4.194.304	39.916.800
12	144	20.736	4.096	16.777.216	479.001.600
13	169	28.561	8.192	67.108.864	6.227.020.800
14	196	38.416	16.384	268.435.456	87.178.291.200
14	196	38.416	16.384	268.435.456	87.178.291.200
16	256	65.536	65.536	4.294.967.296	20.922.789.888.000
17	289	83.521	131.072	17.179.869.184	355.687.428.096.000
18	324	104.976	262.144	68.719.476.736	6.402.373.705.728.000
19	361	130.321	524.288	274.877.906.944	121.645.100.408.832.000
20	400	160.000	1.048.576	1.099.511.627.776	2.432.902.008.176.640.000
21	441	194.481	2.097.152	4.398.046.511.104	51.090.942.171.709.400.000
22	484	234.256	4.194.304	17.592.186.044.416	1.124.000.727.777.610.000.000
23	529	279.841	8.388.608	70.368.744.177.664	25.852.016.738.885.000.000.000
24	576	331.776	16.777.216	281.474.976.710.656	620.448.401.733.239.000.000.000
25	625	390.625	33.554.432	1.125.899.906.842.620	15.511.210.043.331.000.000.000.000
26	676	456.976	67.108.864	4.503.599.627.370.500	403.291.461.126.606.000.000.000.000
27	729	531.441	134.217.728	18.014.398.509.482.000	10.888.869.450.418.400.000.000.000.000
28	784	614.656	268.435.456	72.057.594.037.927.900	304.888.344.611.714.000.000.000.000.000
29	841	707.281	536.870.912	288.230.376.151.712.000	8.841.761.993.739.700.000.000.000.000.000
30	900	810.000	1.073.741.824	1.152.921.504.606.850.000	265.252.859.812.191.000.000.000.000.000.000



- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



- Für viele Problembereiche gibt es keine effizienten Algorithmen
- diese Probleme erfordern ein erschöpfendes Suchen in einem Problemraum
- führen zu NP-vollständigen Algorithmen, d.h., kein Algorithmus existiert, der das Problem in polynomialer Zeit lösen kann
- Typische Beispiele, die erschöpfendes Suchen erfordern:
 - Rundreiseproblem
 - N-Damen-Problem
 - Rucksackproblem
 - Stundenplanerstellung (Job-Shop-Scheduling)
 - Schachzug
 - ...



Erschöpfendes Suchen (2)

- Erschöpfendes Suchen heisst, für ein zu lösendes Problem alle Möglichkeiten zu erzeugen und zu testen und daraus die beste Lösung zu wählen
- 2 Varianten:
 - Aus einer Menge von Elementen finde jene Anordnung der Elemente, die eine optimale Lösung darstellt (Rundreiseproblem)

==> **Permutationen einer Menge erzeugen und testen**

Komplexität: $O(n!)$

- Aus einer Menge von Elementen, finde jene Teilmenge, die eine optimale Lösung darstellt (Rucksackproblem)

==> **Teilmengen von Elementen erzeugen und testen**

Komplexität: mit Wiederholung von Elementen $O\left(\frac{n!}{(n-k)!}\right)$

ohne Wiederholung von Elementen $O\left(\binom{n}{k}\right)$

mit n Anzahl der Elemente der Menge, k Anzahl der Elemente der Teilmenge



Erzeugen aller Anordnungen (Permutationen) einer Menge

- Permutationen einer Menge sind alle möglichen Anordnungen der Elemente
 - Rekursive Entwicklung von Permutationen einer Menge :
 - Stelle jedes Element der Menge an die 1. Stelle der Permutation
 - Hänge daran alle möglichen Permutationen der verbleibenden Menge ohne dem Element an 1. Stelle (*)
- (*) Diese Permutationen werden rekursiv durch das gleiche Vorgehen erzeugt.



Beispiel: Permutationen der Menge $M = \{1, 2, 3, 4\}$

1.Stelle	2. Stelle	3.Stelle	4.Stelle
1	2	3	4
		4	3
	3	2	4
		4	2
	4	2	3
		3	2
2	1	3	4
		4	3
	3	1	4
		4	1
	4	1	3
		3	1
3	1	2	4
		4	2
	2	1	4
		4	1
	4	1	2
		2	1
...



Algorithmus für Permutationen (1)

Erklärungen zum Algorithmus `permutations`

- das Array `object[] perm` diene zur Speicherung der aktuellen Permutation
- Argumente der Methode `permutations`
 - `idx` ist die Position in `perm`, die als nächste besetzt werden soll
 - `perm` enthält bereits Elemente auf Stellen 0 bis `idx-1`
 - `set` enthält die Elemente, die noch nicht in `perm` vorkommen
- an Stelle `idx` werden nun nacheinander die Elemente `elem` der Menge `set` gesetzt (sind alle diejenigen Elemente, die noch nicht in `perm` vorkommen)
- mit den jeweils restlichen Elementen ohne dem Element `elem` (`set \ {elem}`) werden durch den rekursiven Aufruf alle Permutationen von `set \ {elem}` gebildet und an die aktuelle Permutation angehängt



Algorithmus für Permutationen (2)

```
void permutations(Set set, Object[] perm, int idx) {
    if (set.isEmpty()) {
        printOut(perm); // Permutation fertig
    } else {
        for all elem ∈ set {
            perm[idx] = elem;
            permutations(set \ {elem}, perm, idx+1);
            perm[idx] = null;
        }
    }
}
```



Algorithmus für Permutationen: Java-Implementierung

```
public static void permutations(Set set, Object[] perm, int idx) {
    if (set.isEmpty()) {
        System.out.println();
        for (int i = 0; i < perm.length; i++) {
            System.out.print(perm[i].toString());
        }
    } else {
        Iterator it = set.iterator();
        while (it.hasNext()) {
            Object elem = it.next();
            perm[idx] = elem;
            Set setwithoutElem = new TreeSet(set);
            setwithoutElem.remove(elem);
            permutations(setwithoutElem, perm, idx + 1);
            perm[idx] = null;
        }
    }
}
```



Algorithmus für das Erzeugen aller Permutationen: Version 2

- Die 2. Version des Algorithmus für Permutationen verwendet die Markierung `visited` für Elemente der Menge (ähnlich wie bei Graphenalgorithmen)
- Diese Markierung wird bei Verwenden eines Elements in der Permutation für das Element gesetzt und danach wieder rückgesetzt (dieses Zurücknehmen der Verwendung eines Elements nennt man *Backtracking*)

```
void permutations(Set set, Object[] perm, int idx) {
    if (idx == set.size()) {
        printOut(perm);    // Permutation fertig
    } else {
        for all elem ∈ set {
            if (elem not visited) {
                perm[idx] = elem;
                mark elem as visited
                permutations(set, perm, idx+1);
                unmark elem as visited;
                perm[idx] = null;
            }
        }
    }
}
```



Wiederholung: Rekursive Tiefentraversierung bei Graphen

- Rekursive Tiefentraversierung bei Graphen arbeitet mit Markierung von besuchten Knoten.

```
void visit(Node node) {
    node.dooperation();
    mark node as visited
    for each next ∈ node.adjacentNodes() {
        if (next not visited yet ) {
            visit (next);
        }
    }
}
```



2. Algorithmus für Permutationen: Java-Implementierung

```
static Set visited = new TreeSet();

public static void permutations2 (Set set, Object[] perm, int idx)
{
    if (idx == set.size()) {
        System.out.println();
        for (int i = 0; i < perm.length; i++) {
            System.out.print(perm[i].toString());
        }
    } else {
        Iterator it = set.iterator();
        while (it.hasNext()) {
            Object elem = it.next();
            if (! visited.contains(elem)) {
                perm[idx] = elem;
                visited.add(elem);
                permutations2 (set, perm, idx+1);
                visited.remove(elem);
                perm[idx] = null;
            }
        }
    }
}
```



Inhalt

- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



Backtracking-Algorithmen

- Backtracking ist ein allgemeines Problemlösungsverfahren, welches auf gezieltem Entwickeln und Durchprobieren von möglichen Lösungen passiert
- Backtracking ist rekursiv; in den rekursiven Aufrufen gehe folgend vor:
 - Auf Basis einer aktuellen Teillösung wähle einen nächsten Lösungsschritt und erweitere die Teillösung mit dem Lösungsschritt
 - Durch einen rekursiven Aufruf teste, ob mit dieser Teillösung eine Gesamtlösung möglich ist.
 - Wenn ja: ist die Lösung gefunden und gib eine Erfolgsmeldung zurück
 - Wenn nein: nimm die Lösungsschritt zurück und probiere eine Lösung mit einem anderen Lösungsschritt zu finden
 - Mache das mit allen zur Zeit möglichen Lösungsschritten; Kann mit keinem eine Gesamtlösung gefunden werden, melde den Misserfolg zurück



Allgemeine Struktur von Backtracking-Algorithmen

```

boolean FindeLoesung(Lsg teilLoesung, ...) {
  if (teilLoesung ist Lösung) {
    return true;
  } else {
    while (nicht alle Lösungsschritte probiert) {
      Nächsten Lösungsschritt wählen
      teilLoesung mit Lösungsschritt erweitern
      if (FindeLösung (teilLoesung, ...)){
        return true;
      } else {
        Lösungsschritt von teilLoesung zurücknehmen
      }
    }
  } // end while
} // end if
return false; // keine Lösung mit teilLoesung möglich
} // end FindeLoesung

```

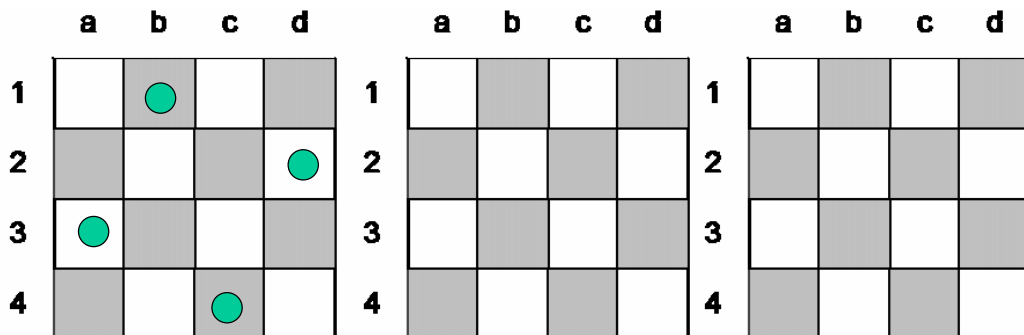


Beispiel: n - Damen Problem

- Gegeben
 - Schachbrett mit $n \times n$ Feldern und n Damen.
- Gesucht:
 - Alle Damen sollen auf dem Schachbrett platziert werden, sodass sie sich gegenseitig nicht bedrohen.
- Lösungsidee:
 - Es gilt, dass Damen alle Figuren bedrohen, die sich in der gleichen Zeile oder Spalte befinden oder in der Diagonale zur Dame stehen.
 - Daraus ergibt sich pro Zeile, Spalte und Diagonale jeweils nur 1 Dame.
- Lösung durch Backtracking-Algorithmus
 - 1) Dame in die erste Zeile setzen
 - 2) In jedem weiteren Schritt in der nächsten Zeile i eine Spalte col suchen, die von keiner der bisher gesetzten Figuren in Zeilen 0 bis $i-1$ bedroht wird.
 - 3) Wenn die Figur nicht positioniert werden kann, den vorherigen Schritt wieder rückgängig machen und andere Spalte für diese Figur suchen (*Backtracking*)



n-Damen-Problem: Arbeitsblatt



Algorithmus n-Damen-Problem (1)

Erklärungen zum Algorithmus:

- Algorithmus verwendet Methode
`int findNextPosition(int col, int line, boolean[][] cb)`
 welche die nächste gültige Spalte nach `col` in Zeile `line` für eine Dame in aktueller Besetzung von Brett `cb` findet (-1 wenn keine gefunden)
- in jeder Zeile `line` wird mit der Methode `findNextPosition` beginnend mit Spalte 0 die nächste Spalte `col` gesucht, deren Zelle nicht bereits bedroht ist.
- Wird eine solche Spalte `col` gefunden, wird eine Dame auf das Feld gesetzt und mit einem rekursiven Aufruf versucht, die restlichen Zeilen zu besetzen.
- Scheitert dies, wird der Zug zurückgenommen und eine nächste Spalte gesucht; dies wird wiederholt, bis alle möglichen Spalten probiert wurden.



Algorithmus n-Damen-Problem (2)

```
boolean queens(int line, boolean[][] chessboard) {
    int col = -1;

    if (line == size) {
        return true;
    } else {
        col = findNextPosition(col, line, chessboard);
        while (col != -1) { // nicht alle
            // Lösungsschritte probiert
            chessboard[line][col] = true;
            if (queens(line+1, chessboard)) {
                return true;
            } else {
                // Sackgasse! Dame wieder entfernen
                chessboard[line][col] = false;
                col = findNextPosition(col, line,
                    chessboard);
            }
        } // end while
    } // end if
}
```



- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- **Rundreiseproblem**
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



Rundreiseproblem (Traveling Salesman Problem)

- Rundreiseproblem: In einem Graph jenen Zyklus über alle Knoten finden, der minimale Distanz hat
- Sind in einem Netzwerk von Knoten alle Knoten miteinander verbunden, ist das Rundreiseproblem im wesentlichen äquivalent dem Erzeugen aller Permutationen der Menge der Knoten
 - Permutation entspricht dem Pfad
 - einen aktuellen Pfad kann man erweitern, indem man alle Knoten versucht, die noch nicht im Pfad enthalten sind
 - Unter allen möglichen Zyklen gibt es einen mit minimaler Distanz; dieser stellt die Lösung des Rundreiseproblems dar.
- Betrachtet man einen beliebigen (zusammenhängenden aber nicht voll verbundenen) Graphen, so ergeben sich folgende neue Aspekte:
 - Versucht man einen aktuellen Pfad mit einem Endknoten v zu erweitern, so sind nur die adjazenten Knoten von v zu betrachten
 - Es ist möglich, dass man in eine Sackgasse läuft, welche zu keinem Zyklus führt



Algorithmus Rundreiseproblem

```
Path shortestCycle = null;

void travelingSalesman(Graph g, Path path) {
    if (path contains all nodes of g) {
        if (shortestCycle == null ||
            path.distance() < shortestCycle.distance()) {
            shortestCycle = path;
        }
    } else {
        Node node = path.end();
        for all next ∈ node.adjacentNodes() {
            if (next not visited) {
                mark next as visited
                travelingSalesman(g, path.extend(next));
                unmark next as visited;
            }
        }
    }
}
```



Effizienzsteigerung beim Erschöpfenden Suchen

- Eine wesentliche Effizienzsteigerung kann man erzielen, wenn bei der Suche nicht zielführende Wege bereits möglichst frühzeitig ausgeschieden werden.
- Es wird getestet, ob eine Teillösung überhaupt zum Ziel führen kann.
- Stellt man fest, dass mit der Teillösung keine oder keine neue Lösung erzielt wird, wird die Teillösung nicht mehr weiter behandelt.
- Dadurch vermeidet man das Untersuchen von ganzen Teilbäumen, was eine erhebliche Effizienzsteigerung bedeuten kann.
- Wesentlich beim Erschöpfenden Suchen ist daher das Finden von Bedingungen, wenn eine Teillösung nicht zum Ziel führen kann.



Effizienzsteigerung beim Rundreiseproblem

Beim Rundreiseproblem kann man Teillösungen ausschließen

1. Erkennen, dass Teillösungen zu keinen Zyklen führen können, weil die Knoten in der Teillösung den Graphen in 2 Subgraphen auftrennen.
2. man betrachtet Zyklen nur in einer Richtung und nicht in die Gegenrichtung

ad 1.) Dies kann man dadurch erreichen, dass man den Teilgraphen ohne die bereits besuchten Knoten betrachtet; zerfällt der Teilgraph in 2 Teile, so kann die Teillösung zu keinem Zyklus führen (erfordert Test auf Zusammenhang).

ad 2.) Dies kann man dadurch erreichen, dass man garantiert, dass 3 ausgewählte Knoten sicher nur in einer gegebenen Reihenfolge im Zyklus besucht werden.



Algorithmus Rundreiseproblem mit Finden von Zyklen

```
Path shortestCycle = null;

void travelingSalesman(↓Graph g, ↓Path path) {
    if (path contains all nodes of g) {
        if (shortestCycle == null ||
            path.distance() < shortestCycle.distance()) {
            shortestCycle = path;
        }
    } else {
        if (cycle with path possible)
            Node node = path.end();
            for all next ∈ node.adjacentNodes() {
                if (next not visited) {
                    if (next has defined predecessor
                        && all predecessor visited) {
                        mark next as visited
                        travelingSalesman(g, path.extend(next));
                        unmark next as visited;
                    }
                }
            }
    }
}
```



- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



- *Branch and Bound* ist eine Erweiterung von Backtracking-Verfahren für Optimierungsverfahren.
- Es werden die Teillösungen bewertet und mit einer bisher besten Lösung verglichen.
- Ergibt sich durch die Bewertung, dass sich durch die Teillösung keine bessere Lösung als die bis dahin gefundene beste Lösung ergeben kann, wird die Teillösung nicht weiter betrachtet (Bound).



Branch and Bound beim Rundreiseproblem

- Einfache Bewertung:
 - Jeder Teilpfad der größere Distanz als der bisher gefundene beste Zyklus hat, muss nicht weiter betrachtet werden.
- Genaue Bewertung:
 - auf Basis der bisher angefallenen Distanz des Teilpfades
 - plus der Distanz des minimalen Spannbaums der noch nicht besuchten Kosten (*)

(*) Der minimale Spannbaum stellt die minimalen Kosten dar, die restlichen Knoten überhaupt verbinden zu können.

gibt es keinen Spannbaum, sind die noch nicht besuchten Knoten nicht verbunden (Backtracking).



Algorithmus Rundreiseproblem mit Branch and Bound

```
Path shortestCycle = null;
```

```
void travelingSalesman(↓Graph g, ↓Path path) {
  if (path contains all nodes of g) {
    if (shortestCycle == null ||
        path.distance() < shortestCycle.distance()) {
      shortestCycle = path;
    }
  } else {
    if (cycle with path possible
        && better cycle than shortestCycle with path possible)
      Node node = path.end();
      for all next ∈ node.adjacentNodes() {
        if (next not visited) {
          if (next has defined predecessor
              && all predecessor visited) {
            mark next as visited
            travelingSalesman(g, path.extend(next));
            unmark next as visited;
          }
        }
      }
    }
  }
}
```



- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- **Dynamisches Programmieren**
- Näherungsverfahren



- Bei diskretem und endlichem Problemraum kann die dynamische Programmierung für viele Optimierungsaufgaben eingesetzt werden
- Funktioniert durch systematisches Durchsuchen aller endlichen Lösungen nach dem *Teile und Herrsche*-Prinzip
 - Lösung eines kleineren Problems
 - darauf aufbauend die Lösung des größeren Problems



Problemstellung

- Gegeben N Klassen von unterschiedlichen Gegenständen die
 - unterschiedliche Größe (size) haben
 - unterschiedlichen Wert (val) repräsentieren
- Gesucht für einen gegebenen Laderaum mit Größe M die optimale Beladung mit Gegenständen der Klassen 1 bis N

Lösungsidee (nach dem Prinzip der dynamischen Programmierung)

- Nehme zuerst nur Gegenstände der Klasse $j = 1$ und versuche für alle Ladungsgrößen $i = 1$ bis M die optimale Ladung zu finden
- Nehme dann Gegenstände der nächsten Klasse $j+1$ und versuche ausgehend von den aktuellen optimalen Ladungen für Ladungsgrößen $i = 1$ bis M durch Austauschen von Gegenständen die derzeit optimalen Lösung zu verbessern
- USW.



- Es gebe $j = 1$ bis N Klassen von Gegenständen mit
 - $size[j]$ ist die Größe des Gegenstands
 - $val[j]$ ist der Wert des Gegenstands
- M sei das maximale Ladungsvolumen
- $cost[i]$ die optimale Lösung für eine Ladung der Größe i
 $best[i]$ der letzte Gegenstand bei der derzeit optimaler Lösung für eine Ladung der Größe i

```
for j = 1 to N {  
  for i = 1 to M {  
    if (i - size[j] >= 0) {  
      if (cost[i] < (cost[i]-size[j] + val[j])) {  
        cost[i] = cost[i]-size[j] + val[j];  
        best[i] = j;  
      }  
    }  
  }  
}
```

Gegenstand j hat Platz

Einfügen von j gibt besseres Ergebnis



Beispiel Dynamische Programmierung des Rucksackproblems

Größe	3	4	7	8	9
Wert	4	5	10	11	13
Bezeichnung	A	B	C	D	E

	k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$j=1$	$cost[k]$	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	$best[k]$			A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
$j=2$	$cost[k]$	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	$best[k]$			A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
$j=3$	$cost[k]$	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	$best[k]$			A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
$j=4$	$cost[k]$	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	$best[k]$			A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
$j=5$	$cost[k]$	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	$best[k]$			A	B	B	A	C	D	E	C	C	E	C	C	D	E	C



Inhalt

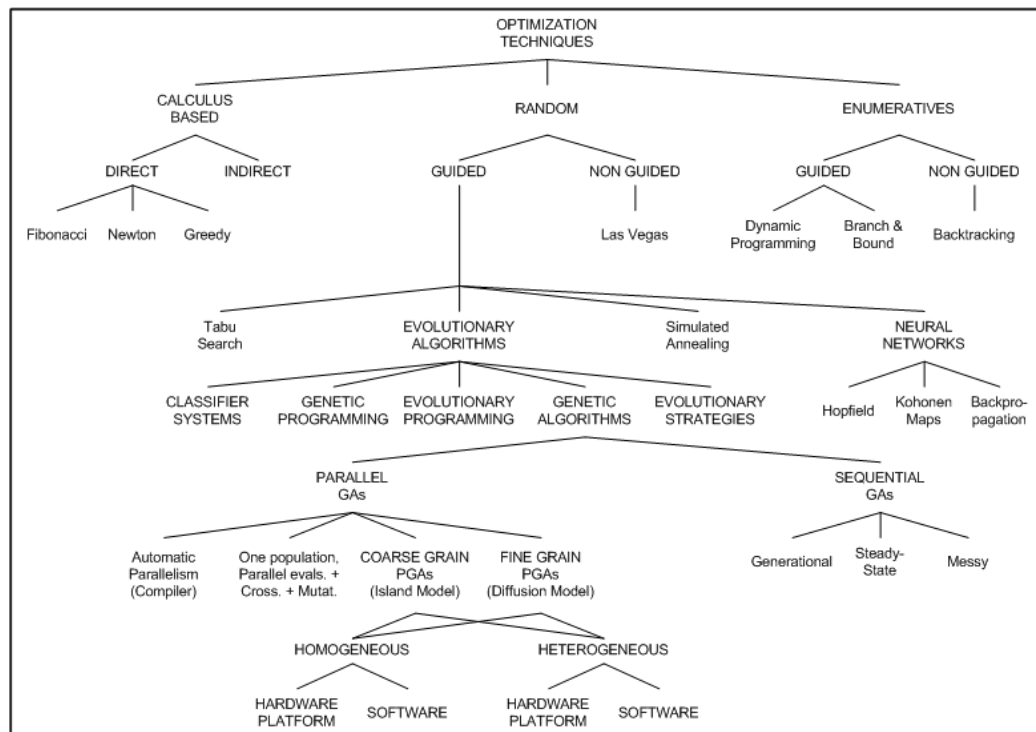
- NP-Vollständigkeit
- Erschöpfendes Suchen
- Backtracking-Algorithmen
- Rundreiseproblem
- Branch and Bound-Verfahren
- Dynamisches Programmieren
- Näherungsverfahren



- Erschöpfenden Suchens ist für viele Optimierungsaufgaben größeren Umfangs nicht anwendbar, wie z.B.
 - geometrische Platzierung von Figuren (Layouting)
 - Finden von optimalen Ladungen
 - Stundenplanerstellung
 - Planung
- Näherungsverfahren bieten hier eine Alternative
 - finden kein Optimum
 - sondern nur eine gute Lösung in den meisten Fällen



Klassifizierung von Suchverfahren



Beispiele von Näherungsverfahren

- Heuristische Suche
- Zufällige Suche (Simulated Annealing)
- Genetische Algorithmen
- Neuronale Netze



Anwendung von zufälliger Suche beim Euklidischen Rundreiseproblem

- Vorgehen (1. Variante)
 - ausgehend von einem beliebigen Zyklus
 - vertausche zwei zufällig ausgewählte benachbarte Knoten
 - ergibt sich dadurch eine Verbesserung, so wird die neue Lösung weiterverfolgt (sonst verworfen)
- Vorgehen (Simulated Annealing)
 - Es wird nicht immer nur die beste Lösung weiterverfolgt, sondern auch einmal eine **etwas** schlechtere, wobei eine **Toleranzgrenze** für die Verschlechterung existiert.
 - Die Toleranzgrenze für die Verschlechterung wird im Laufe des Verfahrens immer weiter gesenkt (Abkühlung)

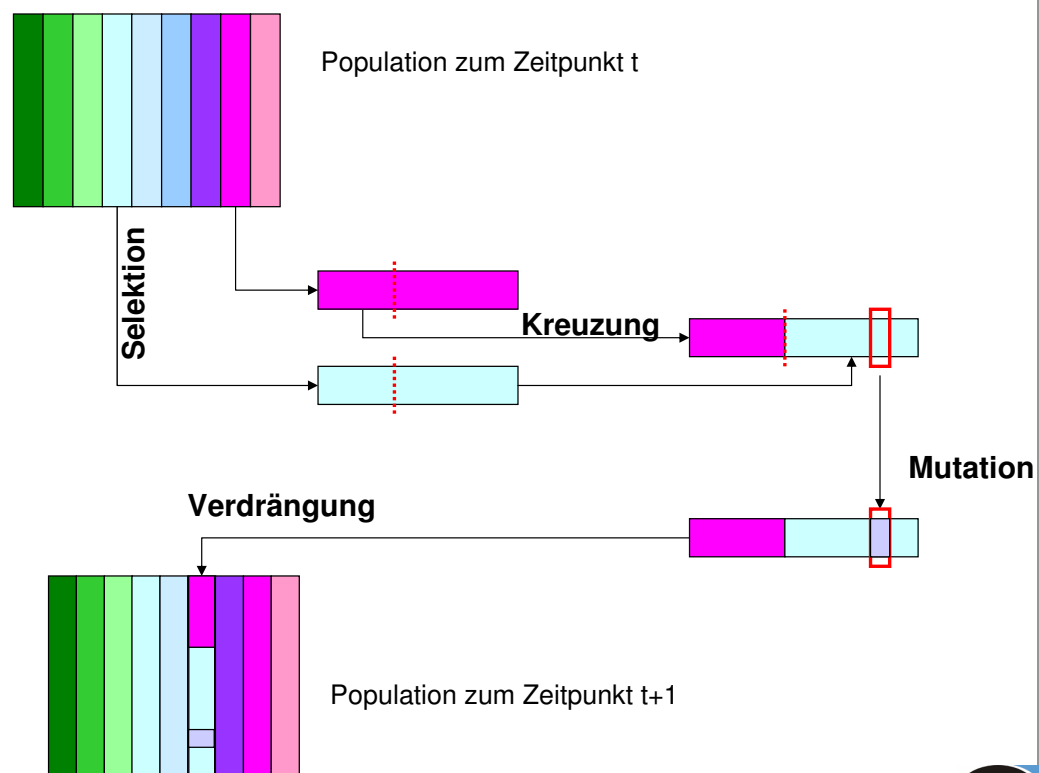


Genetische Algorithmen

- Genetische Algorithmen nehmen sich die Evolutionstheorie als Vorbild und versuchen durch
 - aus einer Population von Lösungen
 - durch gezielte Selektion von guten Lösungen und Ausscheiden (= Aussterben) von schlechten Lösungen
 - durch Kreuzung der selektierten Lösungen (*Crossover*)
 - durch Mutation (= zufällige Veränderung) von Lösungensich einem Optimum anzunähern
- Genetische Verfahren unterscheiden sich durch
 - Bewertungs- und Selektions-Strategien
 - Crossover-Operator
 - Mutations-Verfahren
- Wesentlich ist weiters die Kodierung der Problemlösungen um Kreuzung und Mutation zu ermöglichen (bei genetischen Algorithmen üblicherweise Folge von Bits oder Zahlen)



Prinzip von Genetischen Algorithmen



- Kodierung einer Lösungen: Folge der Städte der Rundreise
 - z.B.: (4, 5, 3, 1, 2, 6)
- Selektion: Auf Basis der Länge der Rundreise
- Kreuzung: Nehme einen Teil einer ersten Lösung und ergänze diesen Teil anhand der restlichen Städte der zweiten Lösung
 - z.B.: (4, 5, 3, 1, 2, 6) + (3, 5, 4, 6, 2, 1) = (4, 5, 3, 6, 2, 1)
- Mutation: Vertauschen von Städten in der Folge
 - z.B.: (4, 5, 3, 6, 2, 1) → (4, 5, 3, 2, 6, 1)



Zusammenfassung

- Für viele Probleme existieren keine effizienten Algorithmen, sondern es ist ein erschöpfendes Suchen notwendig
- Erschöpfendes Suchen ist extrem aufwendig. Es stellt ein NP-vollständiges Problem dar.
- Backtracking-Algorithmen ermöglichen Teillösungen frühzeitig auszuschließen und zu anderen Teillösungen zurückzukehren und weiter zu verfolgen.
- Backtracking-Algorithmen arbeiten mit
 - Test, ob Teillösung zielführend ist
 - Rücknahme der Teillösung und Verfolgung anderer Teillösungen
- Branch and Bound ist Erweiterung von Backtracking für Optimierungsverfahren
 - Bewertung und Ausscheiden von Teillösungen, die garantiert zu keinen optimalen Lösungen führen können.
- Näherungsverfahren finden keine optimale Lösung aber brauchbare Lösung in vernünftiger Zeit
- Heuristische Näherungsverfahren sind ein Forschungsgebiet der Künstlichen Intelligenz

