Kapitel 5 Algorithmische Methoden und Techniken

Für die Lösung der bisher betrachteten Probleme konnten stets Algorithmen gefunden werden, die eine Laufzeit von $O(n)$, $O(n \cdot \log n)$ oder $O(n^2)$ besaßen.

Es gibt jedoch eine Reihe von Problemen, zu deren Lösung wesentlich aufwendigere Algorithmen notwendig sind. Ihre Bearbeitung erfordert speziell bei ‘großer’ Eingabemenge einen derart immensen algorithmischen Aufwand, dass er auch bei Ausführung auf den schnellsten Rechnern jeglichen zeitlichen Vorstellungsrahmen sprengt.

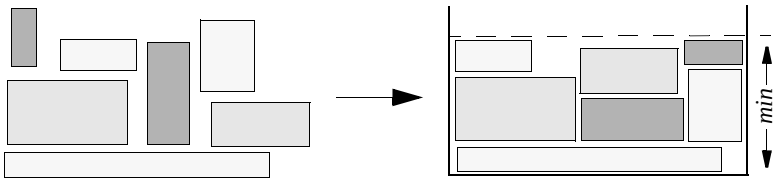
→ Entwicklung *allgemeiner algorithmischer Prinzipien*, die für viele schwierige Probleme zu ‘effizienten’ Lösungsverfahren führen.

Beispielproblem 1: *Das Packmusterproblem*

Gegeben sei eine Menge von *n* quaderförmigen Paketen.

Problem: finde ein Stapelmuster (Rotation der Pakete möglich) derart, dass die Höhe des auf eine Palette aufgeschichteten Stapels möglichst niedrig bleibt.

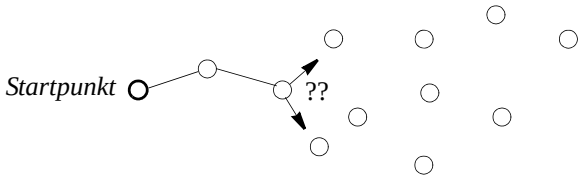
Veranschaulichung für den 2-dimensionalen Fall:



Beispielproblem 2: *Das Problem eines Handelsreisenden (Travelling-Salesman-Problem)*

Gegeben seien *n* Punkte (Kunden), die Entfernung (Länge des Weges) zwischen je zweien dieser Punkte und ein bestimmter Startpunkt.

Problem: finde den kürzesten Weg, der von dem Startpunkt ausgehend alle Punkte besucht und wieder zum Ausgangspunkt zurückkehrt.



Dieses Problem kann als markierter Graph dargestellt werden, wobei die Kunden durch Knoten und ihr Abstand durch Markierungen von Kanten zwischen den Knoten repräsentiert sind. Ausgehend von einem vorgegebenen Startknoten ist dann der kürzeste Hamilton-Kreis zu finden.

Auch wenn diese beiden Beispielprobleme zunächst einfach klingen, so sind sie doch Exemplare der Klasse der von ihrer Zeitkomplexität her aufwendigsten Probleme, den *NP-vollständigen Problemen*. Für NP-vollständige Probleme gibt es keinen (deterministischen) Algorithmus mit polynomialer Laufzeit. In den folgenden Abschnitten stellen wir eine Reihe grundlegender Strategien für Algorithmen zur Lösung solcher und ähnlicher Probleme vor.

Bemerkungen:

① Nicht für jedes Problem kann auf Basis jedes der im folgenden vorgestellten Paradigmen die wirklich optimale Lösung berechnet werden. In der Regel liefern diese jedoch zumindest eine nahezu optimale Lösung.

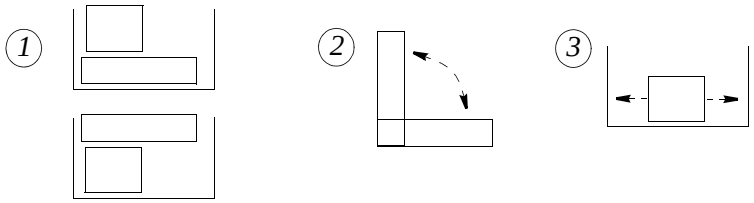
② Für einzelne Klassen von Problemen sind nur bestimmte Algorithmen geeignet.
→ Auswahl einer für das jeweilige Problem geeigneten Lösungsmethode

5.1 Erschöpfende Suche

Paradigma: Berechne die komplette Menge aller zulässigen Lösungen des gegebenen Problems. Hierin ist auch die optimale Lösung enthalten. Diese kann durch einen einfachen Vergleich der Kosten bestimmt werden. Dieses Paradigma wird im Englischen als *exhaustive search* bezeichnet.

Im Beispielproblem 1:

Die Menge aller möglichen Problemlösungen erhalten wir durch
(1) Betrachtung aller $n!$ verschiedenen Einfügesequenzen der Objekte.
(2) Berücksichtigung der 3 (1) Rotationsachsen für Quader (Rechtecke).
(3) Einbeziehung aller möglichen Translationen auf der Palette.



Für jede dieser Lösungen ist die Höhe des resultierenden Stapels zu berechnen.

Im Beispielproblem 2:

Gehen wir davon aus, dass es eine Kante von jedem Knoten zu jedem anderen Knoten gibt. Die möglichen Lösungen (d.h. alle möglichen Touren) ergeben sich durch die Betrachtung aller unterschiedlichen Knoten, die an *i*-ter Stelle ($2 \le i \le n$) besucht werden können. Nachdem der Startknoten fest vorgegeben ist, können zu Beginn also *n-1* Knoten, dann noch *n-2*, ..., und schließlich nur noch ein Knoten besucht werden. Die Anzahl der möglichen Lösungen ist also $(n-1)!$.

Das folgende Algorithmus *´Hamilton_routes´* generiert systematisch alle möglichen Routen des Handelsreisenden auf Basis einer erschöpfenden Suche auf einem als Adjazenzmatrix *a* vorliegenden Graphen. Die Suchstrategie (nicht jedoch die Suche selbst!) ist ähnlich zu einer Tiefensuche, die rekursiv über die Methode *´visit´* realisiert wird:

```
class Hamilton_ routes{
        protected int position []; // als wievielter Knoten besucht ?
        protected int id, startknoten, totalNodes;

        protected void visit (int k) {
            position[k] = id; id++;
            if (id == totalNodes) {
                    if (a[k][startknoten])
                       -->Hamilton´scher Weg gefunden }
            else {    // alle möglichen Folgeknoten betrachten
                    for (int t=0; t < totalNodes; t++) {
                       if (a[k][t]) // Eintrag der Adjazenzmatrix: true/ false
                           if (position[t] == 0) visit(t);   // Tiefensuche
                       } // end for
                } // end else
            id--; position[k] = 0;   // Rücksetzen auf die Ausgangssituation
            }

        public Hamilton_routes (int sKnoten, int n) {
                startknoten = sKnoten; // sKnoten liegt zwischen 0 und (n-1)
                totalNodes = n;
                position = new int [n];
                for (int i = 0; i < n; i++) position[i] = 0;
                id = 0;
                visit (startknoten);
                }
        }
```

Der Bearbeitungsaufwand ist proportional zur Anzahl der Aufrufe der Prozedur ´*visit*´ und damit zur Anzahl der Wege im Graphen. Existieren Kanten zwischen allen Paaren von Knoten, so beträgt dieser $O(n^{n-1})$ (ohne Beweis).

Entscheidendes Problem

Insbesondere für ´große´ Probleme (großes *n*) mit vielen möglichen Lösungen ist die erschöpfende Suche extrem aufwendig. Man sagt, das *Problem* besitzt einen *exponentiellen Aufwand*, wenn die Anzahl der möglichen Lösungen, unabhängig von ihrer Berechnung, exponentiell ist ($n! = O(n^n)$).

Bereits für Größenordnungen von $n \approx 50$ ist dieser Lösungsweg daher nicht mehr praktikabel.

Da eine erschöpfende Suche die Grenzen der Rechnerleistung sprengt, müssen wir versuchen, den Suchraum, d.h. die Menge der explizit berechneten Lösungskandidaten, einzuschränken.

5.2 Lokal-optimierende Berechnung

Der einfachste und (meist) effizienteste Ansatz besteht darin, mit Hilfe geeigneter Heuristiken die optimale Lösung auf direktem Wege zu generieren.

Prinzip: Jeder Schritt in Richtung der Problemlösung wird auf Basis eines lokalen Optimalitätskriteriums (*Heuristik*) ausgeführt. Es werden nicht alle Lösungskandidaten, sondern nur ein einzige Lösung konstruiert.
Dieser Ansatz ist anwendbar, wenn aus einer Folge von optimalen Einzelschritten eine (nahezu) optimale Lösung des Gesamtproblems resultiert.

Algorithmen, die dem Paradigma der lokal-optimierenden Berechnung folgen, werden auch als ‘*greedy algorithms’* bezeichnet.

Bereits bekannte Beispiele:

- Dijkstra’s Algorithmus für kürzeste Pfade in Graphen
- Kruskal’s Algorithmus für minimale Spannbäume

Beispielproblem 1:

Eine mögliche Heuristik, um eine möglichst optimale Stapelung zu finden, besteht in folgendem Ansatz:

Wähle in jedem Schritt des Algorithmus dasjenige Paket aus, das bei optimaler Positionierung (bzgl. Rotation und Translation) die Höhe des Stapels am geringsten anwachsen lässt.

Ein Algorithmus zur heuristischen Lösung von Beispielproblem 2:

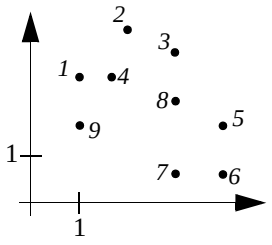
Anwendung des Algorithmus von Kruskal zur Bestimmung minimaler Spannbäume. Abweichendes Kriterium:

statt: Betrachten von Kanten, die keinen Zyklus zur Folge haben

jetzt: • *Ein* Zyklus ist erlaubt.
• Kein Knoten darf einen Grad von ≥ 3 besitzen.

Beispiel: Das *Euklidische travelling-salesman-Problem*

$$Kosten(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$



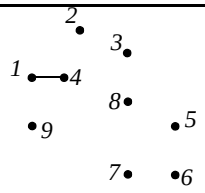
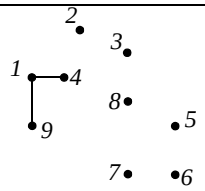
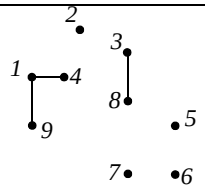
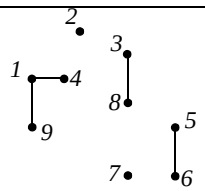
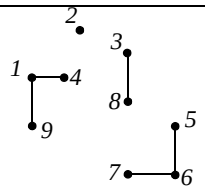
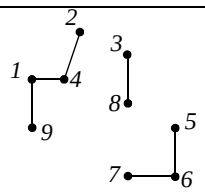
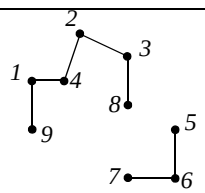
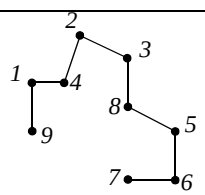
Koordinatenwerte:

1	(1, 2.5)
2	(2, 3.5)
3	(3, 3)
4	(1.6, 2.5)
5	(4, 1.5)
6	(4, 0.5)
7	(3, 0.5)
8	(3, 2)
9	(1, 1.5)

Kostenmatrix (symmetrisch):

	1	2	3	4	5	6	7	8	9
1	0.0	1.4	2.1	0.6	3.2	3.6	2.8	2.1	1.0
2		0.0	1.1	1.1	3.2	3.6	3.2	1.8	2.2
3			0.0	1.5	1.8	2.7	2.5	1.0	2.5
4				0.0	2.6	3.1	2.4	1.5	1.2
5					0.0	1.0	1.4	1.1	3.0
6						0.0	1.0	1.8	3.2
7							0.0	1.5	2.2
8								0.0	2.1
9									0.0

Ablauf des Algorithmus:

Schritt	ausgewählte Kante	aktuelle Knotenmengen	Situation im Graphen
1	(1,4): Länge 0.6	{1, 4}, {2}, {3}, {5}, {6}, {7}, {8}, {9}	
2	(1,9): Länge 1.0 nicht eindeutig!	{1, 4, 9}, {2}, {3}, {5}, {6}, {7}, {8}	
3	(3,8): Länge 1.0 nicht eindeutig!	{1, 4, 9}, {2}, {3, 8}, {5}, {6}, {7}	
4	(5,6): Länge 1.0 nicht eindeutig!	{1, 4, 9}, {2}, {3, 8}, {5, 6}, {7}	
5	(6,7): Länge 1.0	{1, 4, 9}, {2}, {3, 8}, {5, 6, 7}	
6	(2,4): Länge 1.077	{1, 4, 2, 9}, {3, 8}, {5, 6, 7}	
7	(2,3): Länge 1.118 nicht eindeutig!	{1, 4, 2, 3, 8, 9}, {5, 6, 7}	
8	(5,8): Länge 1.118	{1, 4, 2, 3, 5, 6, 7, 8, 9}	

Im Gegensatz zu minimalen Spannbäumen ➞ dies ist noch keine Lösung!

Betrachten der weiteren Kanten nötig:

➞ alle Kanten bis auf (7,9) verletzen die Bedingung: “Grad der Knoten < 3)“. Sie werden ignoriert.

Kante (7,9) mit einer Länge von 2.236 komplettiert die konstruierte Lösung.

Analyse der Laufzeit

• die Kanten werden gemäß ihrer Kosten in einer Heap-Struktur organisiert
➞ Minimum der Kosten (Wurzel des Heap) nicht notwendigerweise eindeutig!

• Die Knotenmengen werden mit Hilfe einer Union-Find-Datenstruktur verwaltet. Anfangs bilden die einzelnen Knoten jeweils einelementige Teilmengen der Knotenmenge.

Die Lauzeitkomplexität des Algorithmus von Kruskal mit diesen Datenstrukturen beträgt $O(e \cdot \log e)$. Für das Euklidische Travelling-Salesman-Problem gilt $e = O(n^2)$.

Dieser konstruktive Ansatz verringert die Laufzeitkomplexität also von $O(n^n)$ auf $O(n^2 \cdot \log n^2)$ und damit auf eine praktikable Größenordnung.

Entscheidendes Problem

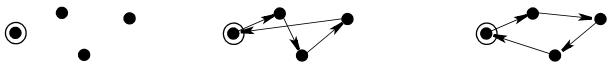
Ein lokal optimierender Algorithmus findet nicht immer die (eine) wirklich optimale Lösung des Gesamtproblems.

➞ nur anwendbar, falls eine ‘gute’ Lösung ausreicht
(wie z.B. für einen Handelsreisenden).

Im Beispiel:

• ein kürzester Hamilton’scher Weg verläuft nicht notwendig über die kürzesten Verbindungskanten der Knoten.

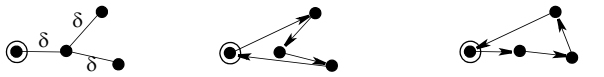
Beispiel:



Situation *Heuristische Lösung* *optimaler Weg*

• die Reihenfolge der Bearbeitung von Kanten mit identischer Markierung kann sich stark auf die Qualität des gefundenen Weges auswirken!

Beispiel:



Situation *mögliche heur.Lösung* *optimaler Weg*

5.3 Backtracking und ´branch-and-bound´-Verfahren

Für viele Probleme ist keine geeignete Heuristik bekannt, um eine tatsächlich optimale Lösung zu finden. In diesem Falle liefern *Backtracking* und *branch-and-bound*-Verfahren grundsätzliche Konzepte, die erschöpfende Suche effizienter zu gestalten.

Grundlegende Idee beider Verfahren

Systematische Einschränkung des Lösungsraumes, indem Lösungen nicht berechnet werden, von denen bereits frühzeitig (d.h. vor ihrer expliziten Berechnung) bekannt ist, dass sie nicht optimal sein können.

Wir gehen dabei von folgender Darstellung aus:

- alle möglichen Lösungen des Problems stellen die Blätter des sogenannten *Lösungsbaumes* dar,
- die Schritte des Algorithmus repräsentieren Pfade zu diesen Blättern und
- die auf dem Weg zu einer Lösung durchlaufenen ´Situationen´ (Teillösungen) entsprechen den inneren Knoten des Baumes.

Lösungsprinzip

Statt eines kompletten Durchlaufs des Lösungsbaumes (*erschöpfende Suche*) versucht man, Zweige (Teilbäume) zum frühest möglichen Zeitpunkt abzuschneiden und nicht mehr zu besuchen.

Abschneiden eines Teilbaumes, sobald bekannt ist, dass

- keine der Lösungen innerhalb dieses Teilbaumes optimal sein kann oder
- feststeht, dass im restlichen Teil des Baumes ebenfalls eine optimale Lösung zu finden ist.

Bemerkungen

① die Verfahren stellen sicher, dass stets eine (global) optimale Lösung des Problems gefunden wird.

② ein bei vielen Problemstellungen wichtiger Aspekt ist die Beseitigung von Symmetrien innerhalb des Lösungsbaumes, deren Bearbeitung oftmals einen sehr großen zusätzlichen Aufwand darstellt.
(z.B.: für den Handelsreisenden ist die Orientierung des optimalen Weges sekundär, da stets auch der umgekehrte Weg die optimale Länge besitzt)

③ für sehr umfangreiche Bäume ist die Einsparung (besonders nahe der Wurzel) so beträchtlich, dass ein recht hoher Aufwand gerechtfertigt ist, um zu untersuchen, ob der Besuch eines Teilbaumes wirklich relevant ist.

Beispiel des Handelreisenden:

➞ es ist gerechtfertigt, minimale Spannbäume der noch nicht betrachteten Knotenmenge zu berechnen, um darin kürzeste Wege abzuschätzen und aufgrund dieser Abschätzung Teile des Baumes abzuschneiden.

5.3.1 Backtracking (*Zurückverfolgung*)

Grundlage: (eingeschränkter) Tiefendurchlauf durch den Lösungsbaum.

Prinzip: Ausgehend von der Wurzel des Lösungsbaumes werden sukzessive vervollständigte Teillösungen (innere Knoten des Lösungsbaumes) in Richtung einer optimalen Gesamtlösung des Problems, d.h. in die Tiefe des Baumes, konstruiert.
Wird an einer Stelle erkannt, dass die darunterliegenden Teilbäume keine optimale Lösung enthalten können (weil z.B. bereits die Teillösung höhere Kosten besitzt als eine zuvor gefundene Gesamtlösung), so steigt der Algorithmus eine Stufe zurück (Rückkehr der Rekursion) und setzt die Suche in einem anderen Teilbaum fort.

Die folgende Methode *´Hamilton_backtrack´* generiert eine optimale Route des Handelsreisenden (ausgehend vom *Startknoten*) auf Basis des *Backtracking*-Prinzips. Bei Terminierung des Algorithmus befindet sich das Ergebnis in '*opt_path*', wobei *opt_path[i]* die Position von Knoten *i* innerhalb des kürzesten Weges angibt.

Das Problem sei dabei durch einen Graphen gegeben, der über seine Kostenmatrix *a* definiert ist. Nicht-existierende Kanten seien durch '∞'-Werte charakterisiert.

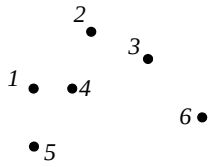
```
class Hamilton_backtrack {
        protected int position [];
        protected int opt_path [];
        protected int id, totalNodes, startknoten;
        protected double min_path_length;

        protected void visit (int k, double akt_length) {
            position[k] = id;
            id++;
            if (id == totalNodes) {
                if ((akt_length + a[k][startknoten]) < min_path_length) {
                        // kürzerer Hamilton´scher Weg gefunden
                        for (int i=0; i < totalNodes; i++)
                            opt_path[i] = position[i];   // kürzesten Weg speichern
                        min_path_length = akt_length + a[k][startknoten];
                    }
            else {    // alle möglichen Folgeknoten betrachten
                for (int t=0; t < totalNodes; t++) {
                        if (((akt_length + a[k][t]) < min_path_length)
                            && (position[t] == 0))
                             // der Teilbaum kann einen kürzeren Weg enthalten
                            visit(t, akt_length + a[k][t]);   // Tiefensuche
                        }
                    } // end else
            id--;               // Rücksetzen auf die Ausgangssituation
            position[k] = 0;   // Rücksetzen auf die Ausgangssituation
            }

        public Hamilton_backtrack (int sKnoten, int n) {
            position = new int [n];
            opt_path = new int [n];
            for (int i=0; i < n; i++) position[i] = 0;
            id = 0;
            totalNodes = n;
            min_path_length = Double.MAX_VALUE;
            startknoten = sKnoten;
            visit ( startknoten, 0.0 );
        }
```

Beispiel

Gegeben sei das (etwas veränderte) *Euklidische travelling-salesman-Problem* von oben:



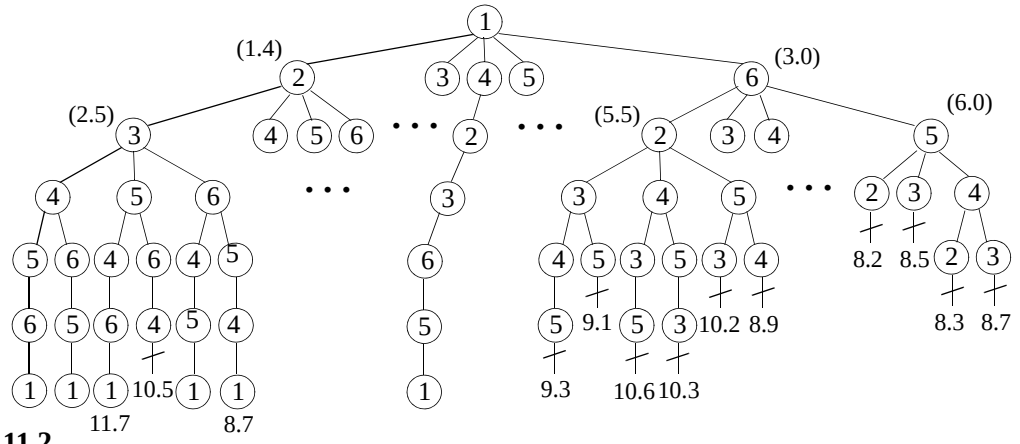
Koordinatenwerte:

1	(1, 2.5)
2	(2, 3.5)
3	(3, 3)
4	(1.6, 2.5)
5	(1, 1.5)
6	(4, 2)

Kostenmatrix (symmetrisch):

	1	2	3	4	5	6
1	0.0	1.4	2.1	0.6	1.0	3.0
2		0.0	1.1	1.1	2.2	2.5
3			0.0	1.5	2.5	1.4
4				0.0	1.2	2.5
5					0.0	3.0
6						0.0

Der zugehörige Lösungsbaum besitzt das folgende Aussehen, wobei die explizit dargestellten Teile bereits die Wirkungsweise des Backtracking-Prozesses wiedergeben:



min. Kosten: **11.2** **10.5** **8.6** **8.2**
(min_path_length)

Der Backtracking-Prozess gliedert sich dabei in zwei wesentliche Teile:

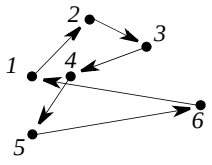
(1) finden einer ersten möglichen Lösung über einen Tiefendurchlauf

(2) eigentlicher Backtracking-Prozess zum Finden der optimalen Lösung

➞ Zurücksteigen in der Rekursionshierarchie; Test alternativer Verzweigungen.

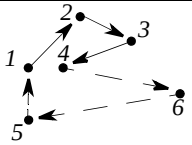
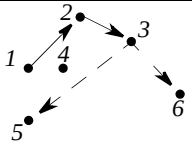
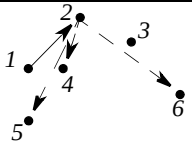
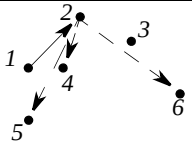
zu (1): Ausgehend von Knoten ‘1’ wird zunächst ‘in die Tiefe’ gesucht, bis eine erste gültige Lösung (Pfad: 1, 2, 3, 4, 5, 6, 1) ermittelt ist:

Länge des Pfades: **11.2**



zu (2): Die rekursive Baumsuche des Tiefendurchlaufs wird nun sukzessive wieder von unten nach oben aufgelöst. Dabei werden auf jeder Rekursionsebene alle alternativen Lösungen mit kürzerer Pfadlänge gesucht, was wiederum (lokale) Tiefendurchläufe zur Folge hat, solange Aussicht auf Erfolg besteht (d.h. der aktuelle Teilweg kurz genug ist).

Die folgende Tabelle zeigt das rekursive Zurücksteigen und die Suche nach alternativen Pfaden.

Rücksteigen bis:		nächster Knoten	restlicher Pfad	Pfadlänge	min. Pfadlänge
1, 2, 3, 4		6	5, 1	10,5	**10,5**
1, 2, 3		5 6	4, 6, 1 6, 4 → Abbruch 4, 5, 1 5, 4, 1	11,7 10,5 8,6 8,7	**10,5** **8,6**
1,2		4 5 6	... ...	... ...	**8,6**
1		3 4 5 6	... ...	... 8,2	**8,6** **8,2**

5.3.2 Branch-and-bound

Grundlage: (eingeschränkter) Breitendurchlauf durch den Lösungsbaum.

Prinzip: Die Bearbeitung startet wiederum mit der Wurzel des Lösungsbaumes.
Beim Durchlauf werden jedoch zunächst *alle* von einem Knoten ausgehenden Kanten des Lösungsbaumes betrachtet. Die Bearbeitung der Söhne erfolgt erst, wenn der Knoten selbst komplett bearbeitet ist. Um eine möglichst geringe Anzahl von Pfaden zu durchlaufen, wird aufgrund zum Teil aufwendiger Untersuchungen und Abschätzungen ermittelt, welcher der Söhne tatsächlich besucht werden muss.

Die folgende Methode *´Hamilton_branch&bound´* benutzt ein allgemeines Verfahren, um Pfade des Baumes geeignet auszugrenzen: Für jeden Knoten (*s*), der auf einem Pfad betrachtet wird, wird eine obere Grenze (*s.upper*) und eine untere Grenze (*s.lower*) für den kürzesten Folgeweg ermittelt (Methoden '*schätze_min_Weg*' und '*schätze_max_Weg*'). Ein Teilbaum braucht nicht betrachtet zu werden, falls dessen untere Schranke größer ist als eine bereits für einen anderen Teilbaum bekannte obere Schranke; in diesem Fall kann der Teilbaum keinen kürzesten Weg enthalten.

Ausgehend vom *Startknoten* wird eine optimale Route des Handelsreisenden auf Basis des *branch-and-bound*-Prinzips generiert. Wiederum sei das Problem durch einen Graphen, definiert über seine Kostenmatrix *a*, gegeben:

```
void Hamilton_branch&bound (int Startknoten, int n) {
    int opt_path [] = new int [n];
    double min_path, min_upper;
    Schlangenelement r = new Schlangenelement(n);
    Schlangenelement s = new Schlangenelement (n);
    Queue q = new Queue();
    min_path = Double.MAX_VALUE; min_upper = Double.MAX_VALUE;
    for (int t = 0; t < n; t++)  s.position[t] = 0;
    s.id = 0;  s.Knoten = Startknoten;  s.path_length = 0.0; s.lower = 0.0;
    q.enqueue(s);
    while (!q.isEmpty()) {
       q.dequeue (s);   (* nächsten Knoten bearbeiten *)
       if  ((s.path_length + s.lower) < min_upper) { //  inzwischen evtl. kleineres min_upper
            s.position[s.Knoten] = s.id;  s.id++;  // Pfad weitersetzen
            if (s.id == n) { // Hamilton´scher Weg gefunden
                if ((s.path_length + a[s.Knoten][Startknoten]) < min_path) {
                    // neuer kürzester Hamilton´scher Weg gefunden
                    for (int t = 0; t < n; t++)
                        opt_path[t] = s.position[t];   // Weg speichern
                    min_path = s.path_length + a[s.Knoten][Startknoten];
                    }
                }
            else { // alle Folgeknoten betrachten
                    for (int t = 0; t < n; t++)  {
                        s.lower = schätze_min_Weg ( t, s.position); // Vorausschau von t
                        s.upper = schätze_max_Weg ( t, s.position); // Vorausschau von t
                        if ((s.position[t] == 0)
                            && (s.path_length + s.lower < min_upper)) {
                            // der Teilbaum kann einen kürzeren Weg enthalten
                            if (s.path_length + s.upper < min_upper)
                                min_upper = s.path_length + s.upper;
                            r = s.clone();
                            r.path_length = s.path_length + a[s.Knoten][t];
                            r.Knoten = t;
                            q.enqueue (r);
                            } // end if
                        } // end for
                } // end else
            } // end if s.lower < min_upper
       } // end while !q.isEmpty
    }
class Schlangenelement {
    int id, Knoten; // id: die aktuelle Länge des Pfades, Knoten: der letzte Knoten des Pfades
    int position [];
    double path_length, upper, lower;
    public Schlangenelement (int n) {position = new int [n];}
    }
```

Bemerkung

schätze_min_Weg(t, s.position) bzw. *schätze_max_Weg*(t, s.position) liefern eine Schätzung für die untere bzw. obere Schranke der resultierenden Pfadlänge, wenn man den durch *s.position* definierten Pfad um den Knoten *t* erweitert.

Betrachten wir wieder das obige Beispiel des *Euklidischen Travelling-Salesman-Problems*:

Beginnend mit der Wurzel wird der Lösungsbaum ebenenweise von oben nach unten durchlaufen.
Zu Beginn liegt (im Gegensatz zum *Backtracking*) kein Lösungskandidat als Vergleichswert vor. Um die relevanten Verzweigungen innerhalb des Lösungsbaumes zu reduzieren, ist man daher auf Schätzwerte für kürzeste Wege angewiesen (Vorausschau 'in die Tiefe'!), deren Qualität die Effizienz des Algorithmus wesentlich beeinflusst.

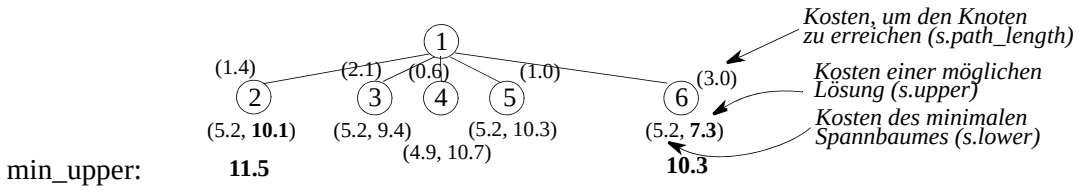
Als unterer (*s.lower*) bzw. oberer (*s.upper*) Schätzwert für die Lösung des aktuellen Teilproblems kann z.B. gewählt werden:

- *s.lower*: Kosten des minimalen Spannbaums der bisher nicht betrachteten Knoten, des Startknotens sowie des aktuellen Knotens.
- *s.upper*: Kosten einer beliebigen Lösung innerhalb des aktuellen Teilbaumes (z.B. linkester oder zufälliger Ast des Lösungsbaumes ➞ leicht zu ermitteln)

  im folgenden: linkester Ast aus Gründen der Nachvollziehbarkeit
  allgemein: zufälliger Ast oder Minimum aus mehreren (2 oder 3) Ästen ➞ meist besseres Ergebnis

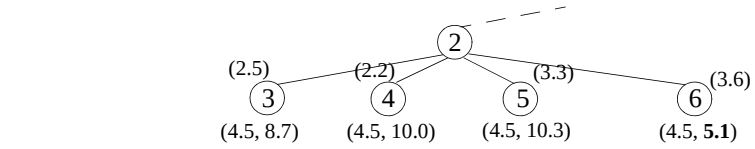
Zusätzlich benötigen wir eine Variable '*min_upper*', um die Kosten des bisher minimalen Lösungskandidaten festzuhalten; sie wird jeweils als aktueller Vergleichswert beim Abschneiden von Teilbäumen herangezogen.

Betrachten wir unter diesen Voraussetzungen den sukzessiven Durchlauf des Lösungsbaumes im obigen Beispiel:

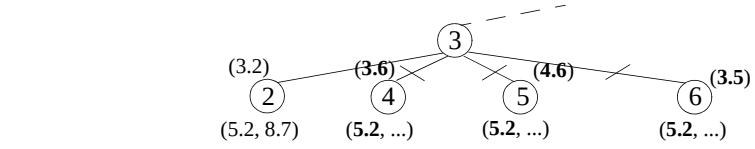


➞ auf oberster Ebene des Baumes müssen alle 5 Verzweigungen betrachtet werden.

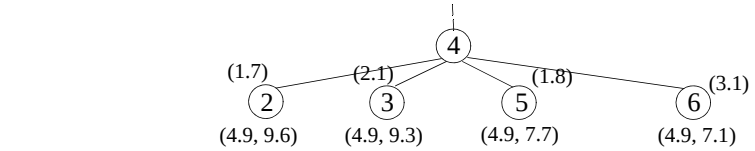
Im nächsten Schritt werden die (jeweils 4) von diesen Knoten ausgehenden möglichen Wege betrachtet:



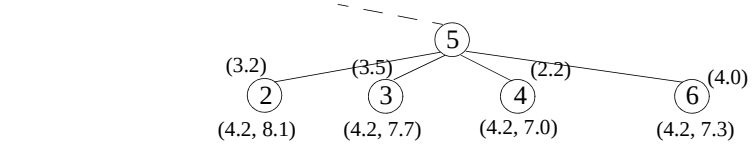
min_upper: **10.3** **8.7**



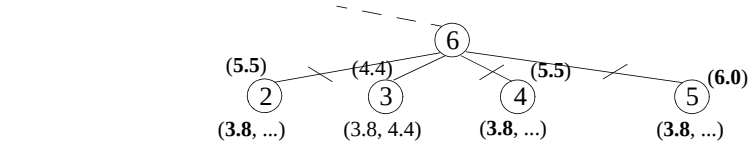
min_upper: **8.7**



min_upper: **8.7**



min_upper: **8.7**



min_upper: **8.7**

➞ bereits auf dieser Ebene können 6 der 20 möglichen Teilbäume abgeschnitten werden (wobei jeder einzelne Teilbaum einer Menge von 6 verschiedenen Lösungen entspricht!).

Begründung: wir verwenden im Gegensatz zum *Backtracking* bereits in den oberen Ebenen des Baumes eine (approximative) Vorausschau in *alle* Teilbäume.

➞ sofern eine gute Schätzfunktion bekannt ist, verhält sich das *branch-and-bound*-Verfahren in der Regel besser als *Backtracking*, da bereits weiter oben im Baum abgeschnitten werden kann.

Jedoch können weder *branch-and-bound* noch *Backtracking* den exponentiellen Bearbeitungsaufwand grundsätzlich verringern (*Worst Case*: $O(n^n)$), unabhängig von den Kriterien, die zur Beschränkung der Baumsuche herangezogen werden. Andererseits kann die in akzeptabler Zeit lösbare Problemgröße mit heuristischen Ansätzen in der Regel (*Average Case*) um eine Größenordnung und mehr gesteigert werden.

5.4 Divide-and-Conquer - Verfahren

Das wohl am häufigsten und allgemeinsten angewendete Verfahrensprinzip zur Konstruktion effizienter Problemlösungen und Algorithmen für umfangreiche Probleme ist das Prinzip des *Divide-and-Conquer* (*lat.: divide et impera*).

Prinzip: 'Große' Probleme werden in geeigneter Weise in (wenige) kleinere Teilprobleme zerlegt:

- sind die Teilprobleme klein genug, so werden sie direkt gelöst
- andernfalls werden sie in der Regel ihrerseits nach dem gleichen Prinzip bearbeitet (➞ rekursive Anwendung des Verfahrens).

Die Lösungen der Teilprobleme werden schließlich zu einer Lösung des Gesamtproblems zusammengesetzt.

Die allgemeine Kontrollstruktur von *Divide-and-Conquer*-Algorithmen besitzt somit das folgende Aussehen:

ALGORITHM Divide-and-Conquer;
  IF Problemgröße klein genug
    THEN löse das Problem direkt *(* mit einfacher Methode *)*
    ELSE
      Divide: Zerlege das Problem in Teilprobleme möglichst gleicher Größe;
      Conquer: Löse jedes Teilproblem für sich;
      Merge: Berechne aus den Teillösungen die Gesamtlösung
  END Divide-and-Conquer;

Bereits bekannte Beispiele:

- einige ´gute´ Sortierverfahren (*Quick_Sort* und *Merge_Sort*).
- binäre Suche

Problem: Das Verfahren liefert nur dann eine analytisch nachweisbare algorithmische Verbesserung, falls das Ausgangsproblem gleichmäßig auf Teilprobleme verteilt werden kann, diese also eine 'effektiv geringere' Problemgröße besitzen.
➞ Balancierung der Teilprobleme nötig! (*oftmals schwierig*)

Nach der Analyse von Abschnitt 3.1.3 gilt:

Der Algorithmus besitzt eine Laufzeit von $O(n \cdot \log n)$, falls:

- *Divide*- und *Merge*-Schritt nicht länger als $O(n)$ Zeit benötigen,
- die Bearbeitung 'kleiner' Probleme in konstanter ($O(1)$) Zeit erfolgt und
- eine balancierte Unterteilung des Ausgangsproblems garantiert ist.

Hieraus folgt:

Divide-and-Conquer ist nur dann gewinnbringend einsetzbar, wenn das Gesamtproblem in voneinander unabhängige Teilprobleme zerlegt werden kann, diese also (weitgehend) isoliert gelöst und die Ergebnisse in einfacher Weise zusammengesetzt werden können (➞ einfacher *Merge*-Schritt).
Ist dies nicht der Fall, so besitzt bereits der *Merge*-Schritt die Komplexität des Ausgangsproblems und ein algorithmischer Gewinn ist nicht zu erwarten.

Aus diesem Grund führen *Divide-and-Conquer*-Verfahren nicht zu einer algorithmischen Verbesserung der beiden Beispielprobleme dieses Kapitels (Packmusterproblem bzw. Travelling-Salesman-Problem).

5.5 Dynamische Programmierung

Häufig kann ein vorgegebenes Problem nicht oder nur mit großem Aufwand in eine kleine Menge unabhängiger Teilprobleme zerlegt werden, deren Teilergebnisse sich zu einer Gesamtlösung verknüpfen lassen. Ein insbesondere im Bereich des *Operations Research* weit verbreiteter Ansatz ist es daher, 'alle möglichen' kleinen Teilprobleme (optimal) zu lösen, deren Lösungen zu speichern und sie 'von untern nach oben' zu einer optimalen Lösung des Gesamtproblems zusammenzusetzen.

Prinzip: 'Große' Probleme werden geeignet in viele (einfache) Basisprobleme zerlegt, die über ein direktes Verfahren gelöst und deren Lösungen innerhalb einer globalen Lösungstabelle gespeichert werden.
Auf dem Weg zu einer Lösung des Gesamtproblems werden aufgrund dieser Lösungstabelle sukzessive Lösungen komplexerer Teilprobleme berechnet und ihrerseits in die Tabelle eingetragen. Die Gesamtlösung wird auf diese Weise 'von unten nach oben' aus den jeweiligen Teillösungen zusammengesetzt.

Dabei kommt das folgende **Optimalitätsprinzip** zum Tragen:

Die Lösung eines Teilproblems der Größe k ist optimal, falls sie aus einer optimalen Zusammensetzung der optimalen Lösungen der Teilprobleme der Größe $< k$ hervorgeht. Als optimal erkannte Teillösungen bleiben demnach optimal beim Zusammensetzen zu Lösungen für größere Probleme.

➞ um die (alle) optimale(n) Lösung(en) des Gesamtproblems zu ermitteln, müssen nicht alle möglichen Lösungen jeder Problemgröße $k < n$, sondern nur 'lohnende', weil in obigem Sinne optimale Teillösungen berechnet werden.

Bereits bekannte Beispiele für dieses Lösungskonzept:

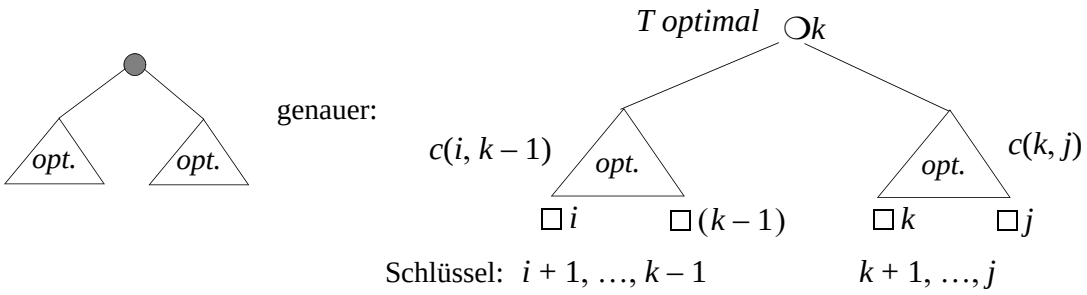
- Aufbau optimaler binärer Suchbäume (vgl. Abschnitt 2.3)

  Optimalitätskriterium: Alle Teilbäume eines optimalen Suchbaumes sind optimal.

  ➞ Idee: ausgehend von einzelnen Knoten als minimalen Teilbäumen werden systematisch "immer größere" optimale Teilbäume konstruiert.

  '*Bottom-up*' - Methode:

  Ein neuer optimaler Teilbaum ergibt sich aus einer geeigneten Wurzel und zwei optimalen Teilbäumen (*bereits berechnet*) dieser Wurzel.



Seien $c(i, j)$, $0 \le i \le j \le n$, die Kosten eines optimalen Teilbaumes mit den Gewichten $p_{i+1}, \ldots, p_j$; $q_i, \ldots, q_j$; sei weiterhin: $w(i, j) = \sum_{k=i+1}^{j} p_k + \sum_{l=i}^{j} q_l$.

Dann können die Werte $c(i, j)$ nach folgendem Rekursionsschema berechnet werden:

$c(i, i) = 0$ ➞ der Baum besteht aus $\square\, i$ (*leerer Baum*).

$$c(i, j) = w(i, j) + \min_{i < k \le j} (c(i, k-1) + c(k, j)) \qquad \text{für } i < j .$$

Denn: das minimale Gewicht des Teilbaums mit Wurzel $\bigcirc k$ und den Grenzen i bzw. j ist gegeben durch: $w(i, j) + c(i, k-1) + c(k, j)$.

- der Algorithmus von *Floyd* zur Lösung des 'all pairs shortest path'-Problems sowie der Algorithmus von *Warshall* zur Berechnung der transitiven Hülle in gewichteten Graphen (das aktuelle Teilproblem wird dabei unter Rückgriff auf bereits berechnete Teilprobleme gelöst)

Probleme bei der Anwendung der *dynamischen Programmierung*:

- es ist nicht immer möglich, die Lösung eines komplexen Problems aus der Lösung kleinerer, unabhängiger Einzelprobleme zu kombinieren.
- die Anzahl der zu lösenden Teilprobleme und damit der für die Teilergebnisse benötigte Speicherplatz kann unverhältnismäßig groß sein (im Extremfall: exponentiell).

→ es gibt viele '*schwierige*' Probleme, für die das Verfahren nicht anwendbar oder kein entscheidender Gewinn in der Laufzeit zu erwarten ist (z.B. das Problem des Handelsreisenden), aber auch eine Reihe '*leichter*' Probleme, für die Standardalgorithmen effizienter sind.

Die *dynamische Programmierung* kann als Verallgemeinerung von *Divide-and-Conquer* sowie der *erschöpfenden Suche* betrachtet werden:

- *Divide-and-Conquer*:
  Berechnung von Informationen zur Lösung einzelner kleinerer Teilprobleme, die in bereits zuvor bestimmter Weise ('*Divide*-Schritt') zur Lösung des Gesamtproblems zusammengesetzt werden.
- *erschöpfende Suche*:
  Berechnung der kompletten Menge von Informationen über alle Lösungsmöglichkeiten des Gesamtproblems und Auswahl der optimalen Lösung.

→ die *dynamische Programmierung* berechnet Informationen zur Lösung von Teilproblemen und setzt diese 'in geeigneter Weise' nach dem Optimalitätsprinzip zu den zur Lösung größerer Probleme benötigten Informationen und schließlich zur Lösung des Gesamtproblems zusammen.

Für das Prinzip der *dynamischen Programmierung* gibt es ein sehr breites Spektrum von Anwendungsmöglichkeiten; sie definiert gleichsam eine 'natürliche' Methode zur Entwicklung von Lösungskonzepten.