LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Algorithmen und Datenstrukturen

Kapitel 6: Algorithmische Methoden und Techniken

Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2015

Ludwig-Maximilians-Universität München

basierend auf dem Skript von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Allgemein I

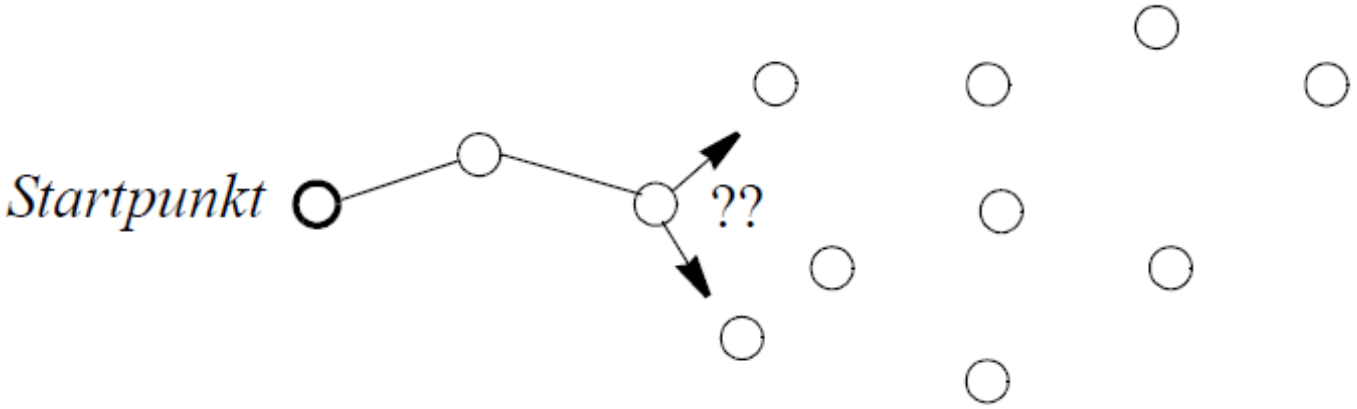
- Für die Lösung der bisher betrachten Probleme: Algorithmen mit Laufzeit $O(n)$, $O(n \cdot \log n)$ oder $O(n^2)$
- Aber: Für eine Reihe von Problemen sind für deren Lösung wesentlich aufwendigere Algorithmen notwendig
- Speziell bei großer Eingabemenge sprengt der immense algorithmische Aufwand (auch bei Ausführung auf den schnellsten Rechnern) jeglichen zeitlichen Vorstellungsrahmen

→ Entwicklung allgemeiner **algorithmischer Prinzipien**, die für viele schwierige Probleme (Problemklassen) zu ‚effizienten' Lösungsverfahren führen.

Allgemein II

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

- Graphproblem: Gegeben seien n Punkte (Kunden)→Knoten, die Entfernung (Länge des Weges) zwischen zweien dieser Punkte (→Kanten) und ein bestimmter Startpunkt.
- **Problem:** Finde den kürzesten Weg, der von dem Startpunkt ausgehend alle Punkte besucht und wieder zum Ausgangspunkt zurückkehrt.



Allgemein III

- Lösungsparadigmen die im Folgenden betrachtet werden:
  - Erschöpfende Suche (exhaustive search)
  - Lokal-optimierende Berechnung (Greedy) → ‚oft' nicht opt. Lösung
  - Backtracking / Branch-and-Bound
  - Divide-and-Conquer
  - Dynamische Programmierung

Bemerkungen:

- Nicht für jedes Problem kann auf Basis jedes der oben genannten Paradigmen die wirklich optimale Lösung berechnet werden. In der Regen liefern diese jedoch zumindest eine nahezu optimale Lösung.
- Für einzelne Klassen von Problemen sind nur bestimmte Algorithmen geeignet.

→ Auswahl einer für das jeweilige Problem geeigneten Lösungsmethode

Erschöpfende Suche I

Paradigma:

- Berechnung der kompletten Menge aller zulässigen Lösungen des gegebenen Problems.
- Hierin ist auch die optimale Lösung enthalten.
- Bestimmung der optimalen Lösung durch einfachen Vergleich der Kosten.
- Dieses Paradigma wird im Englischen als **exhaustive search** bezeichnet.

Erschöpfende Suche II

In unserem Beispielsproblem (Travelling-Salesman-Problem)**:**

- Annahme: Es gibt von jedem Knoten zu jedem anderen Knoten eine Kante.
- Die möglichen Lösungen (d.h. alle Touren) ergeben sich durch die Betrachtung aller unterschiedlichen Knoten, die an i-ter Stelle ($2 \leq i \leq n$) besucht werden können.
- Startknoten fest vorgegeben → es können zu Beginn noch $n-1$ Knoten, dann noch $n-2$ Knoten, …, am Ende nur noch ein Knoten besucht werden.
- Die Anzahl der möglichen Lösungen ist also $(n-1)!$.
- Bearbeitungsaufwand ist proportional zu der Anzahl der Wege im Graphen und beträgt $O(n^{n-1})$. (ohne Beweis)

Erschöpfende Suche III

Entscheidendes Problem:

- Insbesondere für ‚große' Probleme (großes n) mit vielen möglichen Lösungen ist die erschöpfende Suche extrem aufwendig.
- 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**.

Lokal-optimierende Berechnung I

Greedy-Algorithmen:

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

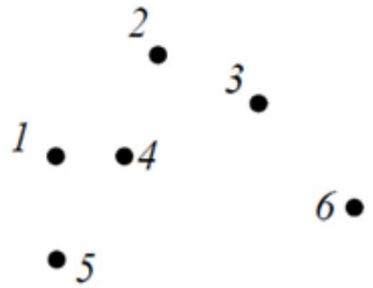
- **Beispiele:**
  - Dijkstra's Algorithmus für kürzeste Pfade in Graphen
  - Kruskal's Algorithmus für minimale Spannbäume
- **Prinzip:**
  - Jeder Schritt in Richtung der Problemlösung wird auf Basis eines lokalen Optimalitätskriteriums (**Heuristik**) ausgeführt.
  - Es werden nicht alle Lösungskandidaten, sondern nur eine einzige Lösung konstruiert.
  - Anwendbar, wenn aus einer Folge von optimalen Einzelschritten eine (nahezu) optimale Lösung des Gesamtproblems resultiert.

Lokal-optimierende Berechnung II

In unserm Beispielproblem (Travelling-Salesman-Problem)**:**

Mögliche Heuristik:

- 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.



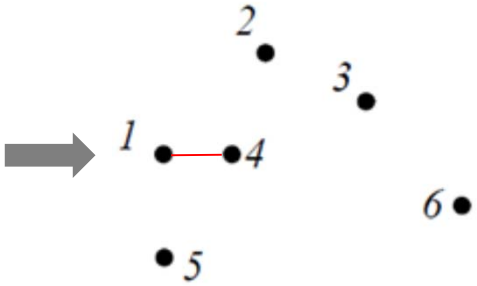
Hinzufügen von Kante (1,4)

Kostenmatrix:
(Euklid. Dist.)

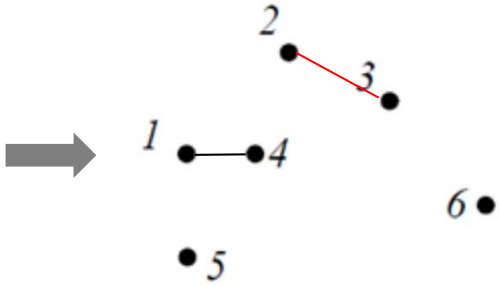
	1	2	3	4	5	6
1	--	1.4	2.1	0.6	1.0	3.0
2		--	1.1	1.1	2.2	2.5
3			--	1.5	2.5	1.4
4				--	1.2	2.5
5					--	3.0
6						--

Lokal-optimierende Berechnung III

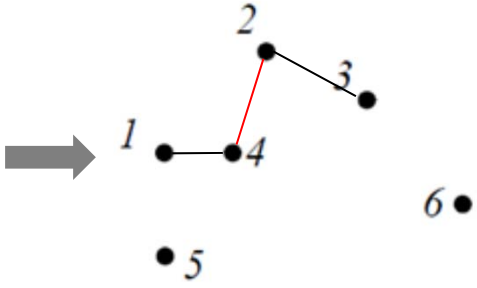
In unserm Beispielproblem (Travelling-Salesman-Problem):



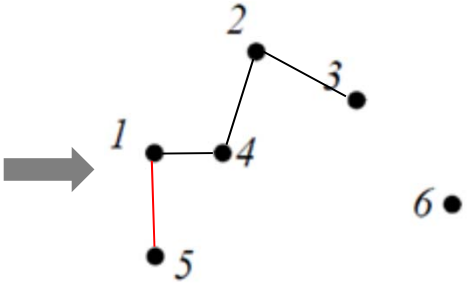
+ Kante (1,4)



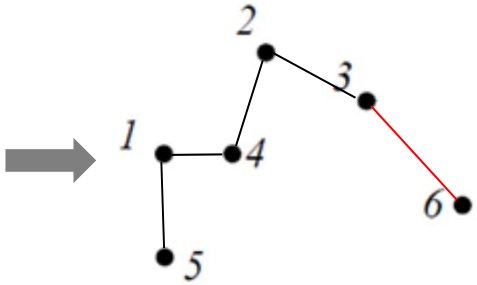
+ Kante (2,3)



+ Kante (4,2)



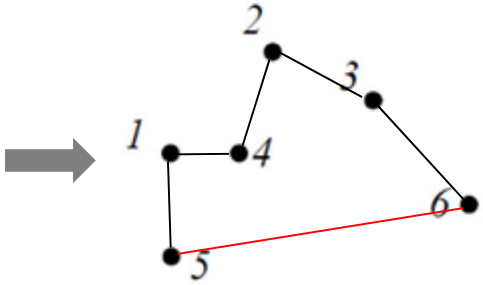
+ Kante (1,5)



+ Kante (3,6)

→ Spannbaum (noch keine Lösung!)

→ alle Kanten bis auf (6,5) verletzen die Bedingung "Grad der Knoten <3"



+ Kante (6,5)

→ „Nahezu" optimale Lösung !!!

Lokal-optimierende Berechnung IV

Analyse der Laufzeit

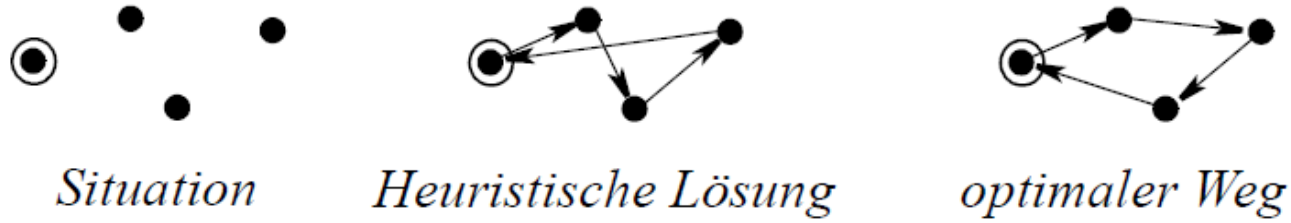
- Dieser konstruktive Ansatz verringert die Laufzeitkomplexität also von $O(n^n)$ auf $O(n^2 \cdot \log n^2)$ und damit auf eine praktikable Größenordnung.

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).

  Beispiel:

  – ein kürzester Hamilton'scher Weg verläuft nicht notwendig über die kürzesten Verbindungskanten der Knoten.



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

Backtracking und ‚branch-and-bound'-Verfahren I

- 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.

Backtracking und ‚branch-and-bound'-Verfahren II

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
  - die auf dem Weg zu einer Lösung durchlaufenen ‚Situationen' (Teillösungen) entsprechen den inneren Knoten des Baumes.

Backtracking und ‚branch-and-bound'-Verfahren III

Lösungsprinzip

- Statt eines kompletten Durchlaufs des Lösungsbaumes (**erschöpfende Suche**) versucht man, Zweige (Teilbäume) zum frühstmöglichen 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.
- die Verfahren stellen sicher, dass stets eine (global) **optimale Lösung des Problems** gefunden wird.

Backtracking und ‚branch-and-bound'-Verfahren IV

Bemerkung:

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.

Backtracking I

Grundlage:
(eingeschränkter) **Tiefendurchlauf** durch den Lösungsbaum.

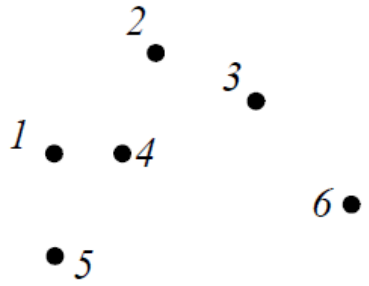
Prinzip:

- 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.

Backtracking VI

Beispiel

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



Koordinatenwerte:

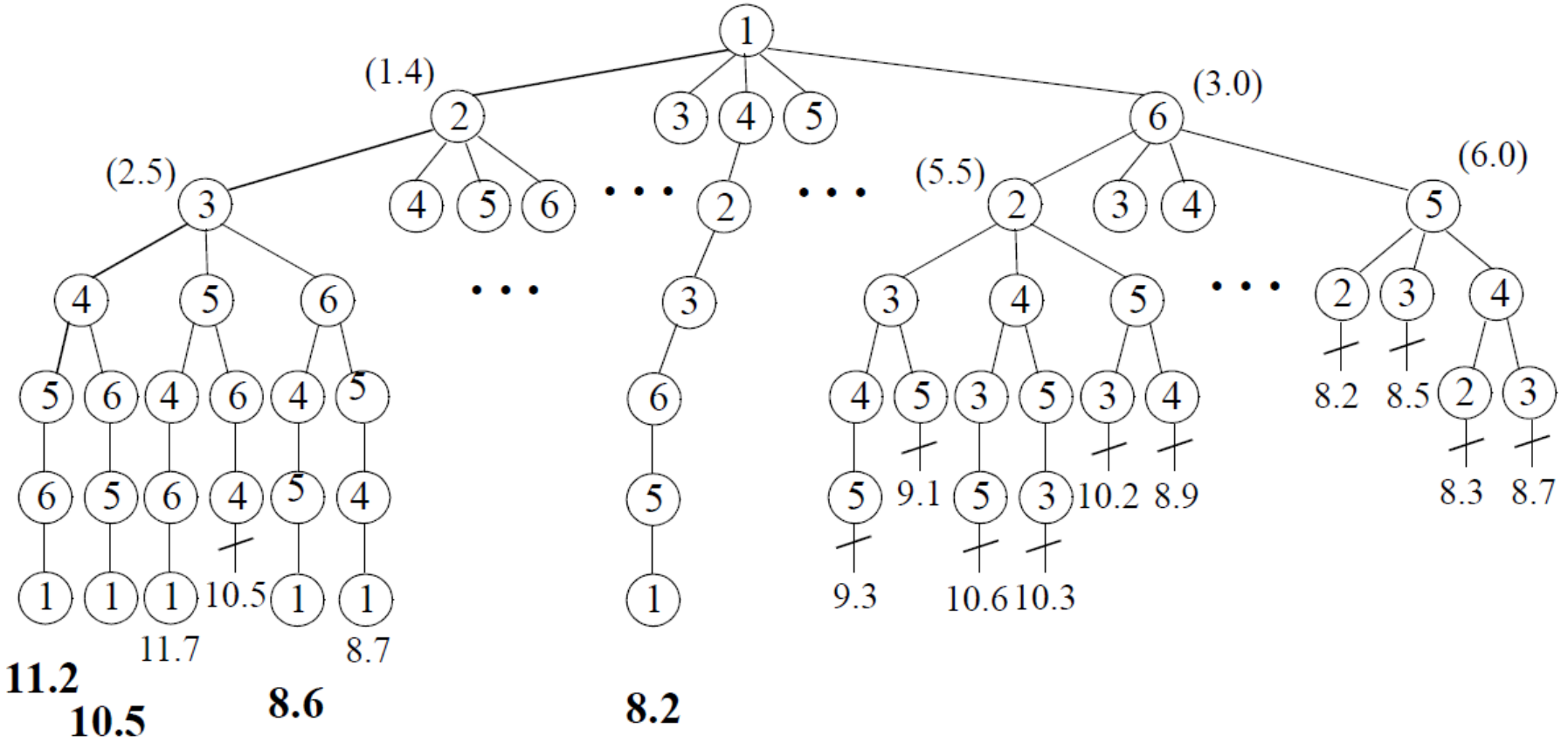
1	(1, 2.5)
2	(2, 3.5)
3	(3, 3)
4	(1.6, 2.5)
5	(1, 1.5)
6	(4, 2)

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

Backtracking VII



Min. Kosten = 8.2 (min_path_length)

Backtracking VIII

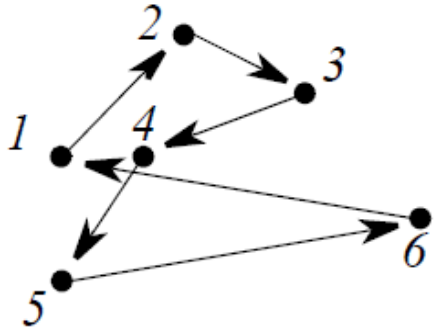
Der Backtracking-Prozess gliedert sich in zwei wesentliche Teile:

(1) finden einer ersten möglichen Lösung über einen Tiefendurchlauf

(2) Backtracking-Prozess zum Finden der optimalen Lösung
→ Zurücksteigen in der Rekursionshierarchie;
Test alternativer Verzweigungen.

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

Länge des Pfades: **11.2**



Backtracking IX

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).

Branch-and-bound I

Grundlage:
(eingeschränkter) **Breitendurchlauf** durch den Lösungsbaum.

Prinzip:

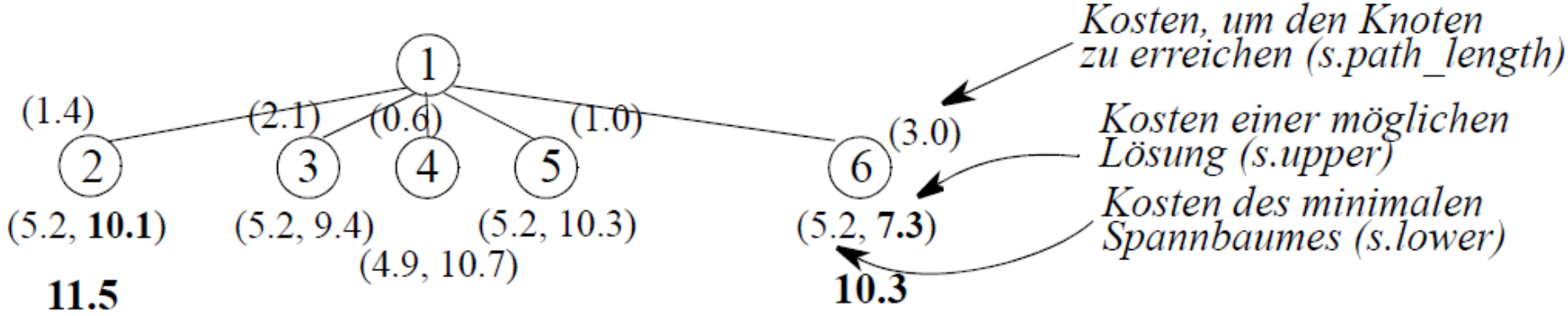
- Die Bearbeitung startet wiederum mit der Wurzel des Lösungsbaumes.
- Beim Durchlauf werden jedoch zunächst *alle* von einem Knoten ausgehenden Kanten des Lösungsbaumes betrachtet.
- Die Bearbeitung der Söhne erfolgt erst, wenn der Knoten selbst komplett bearbeitet ist.
- Abschneiden von Pfaden aufgrund zum Teil aufwendiger Untersuchungen und Abschätzungen, z.B. über Abschätzung von untere (lower-) und obere (upper-) Schranken (bounds) der Kosten.

Branch-and-bound II

- Ein Teilbaum braucht nicht betrachtet zu werden, falls dessen untere Schranke (lower bound) größer ist als eine bereits für einen anderen Teilbaum bekannte obere Schranke (upper bound); in diesem Fall kann der Teilbaum keinen kürzesten Weg enthalten und somit ignoriert (pruning).
- Als unterer (`s.lower`) bzw. oberer (`s.upper`) Schranken 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)

Branch-and-bound X

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



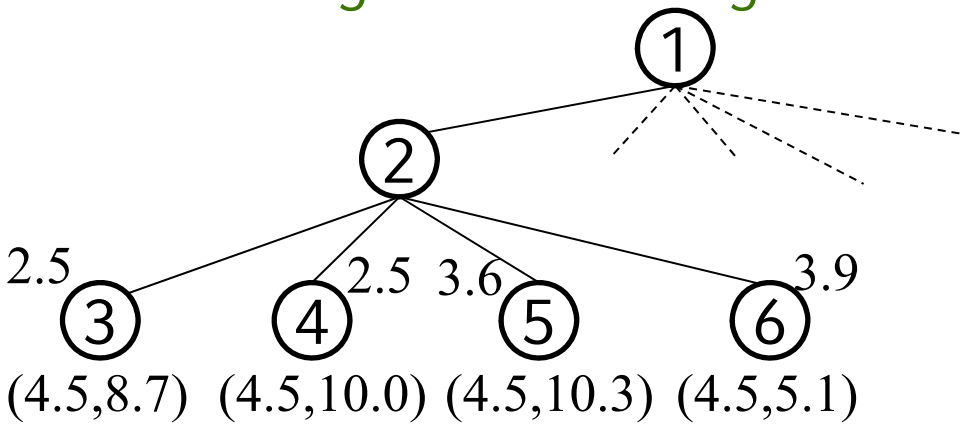
min_upper = 10.3:

  - Globale Variable *min_upper* zum Merken der besten oberen Schranke mit der Teilbäume abgeschnitten werden.
  - Abschneiden falls s.lower + s.path_length > min_upper

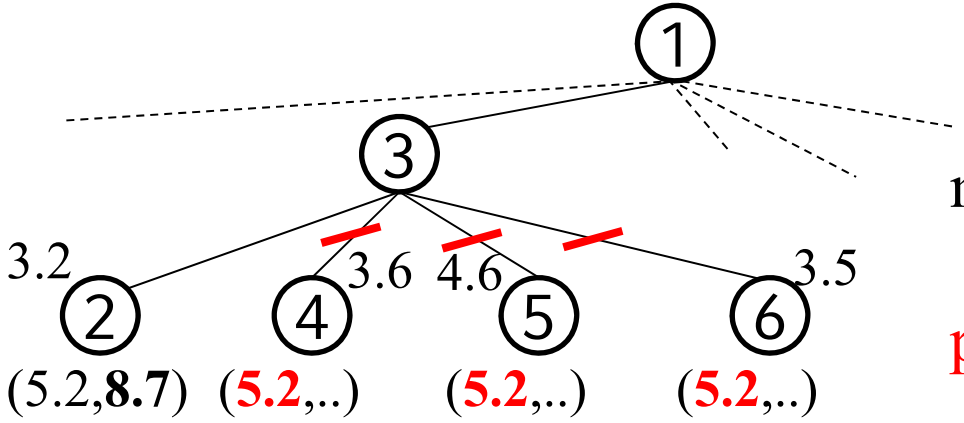
→ auf oberster Ebene des Baumes müssen alle 5 Verzweigungen betrachtet werden.

Branch-and-bound XI

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



min_upper = 9.0:



min_upper = **8.7**:

pruning von Teibäumen 4,5 u. 6

Branch-and-bound XIV

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.

Branch-and-bound XV

Aber:

- Weder branch-and-bound noch Backtracking können den exponentiellen Bearbeitungsaufwand grundsätzlich verringern (Worst Case: $O(n^n)$), unabhängig von den Kriterien, die zur Beschränkung der Baumsuche herangezogen werden.

Andererseits:

- Die in akzeptabler Zeit lösbare Problemgröße kann mit heuristischen Ansätzen in der Regel (**Average Case**) um eine Größenordnung und mehr gesteigert werden.

Divide-and-Conquer-Verfahren I

Divide-and-Conquer (lat.: divide et impera):

Das wohl am häufigsten und allgemeinsten angewendete Verfahrensprinzip zur Konstruktion effizienter Problemlösungen und Algorithmen für umfangreiche Probleme

Prinzip:

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

- Sind die Teilprobleme klein genug: Direkte Lösung
- ansonsten: Bearbeitung ihrerseits nach dem gleichen Prinzip (→ rekusive Anwendung des Verfahrens)

Abschließend: Lösungen der Teilprobleme → Zusammensetzung zu einer Lösung des Gesamtproblems.

Divide-and-Conquer-Verfahren II

Allgemeine Kontrollstruktur von Divide-and-Conquer-Algorithmen:

```
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;
```

Divide-and-Conquer-Verfahren III

Bereits bekannte Beispiele:

- einige ´gute´ Sortierverfahren (**QuickSort** und **MergeSort**).
- 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*)

Divide-and-Conquer-Verfahren IV

Laufzeitanalyse

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

- Divide- und Merge-Schritt nicht länger als $O(n)$ Zeit benötigt
- Die Bearbeitung ‚kleiner' Probleme in konstanter Zeit ($O(1)$) erfolgt
- Eine balancierte Unterteilung des Ausgangsproblems garantiert ist.

Divide-and-Conquer-Verfahren V

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 des Beispielproblems dieses Kapitels (Travelling-Salesman-Problem).

Dynamische Programmierung I

- 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 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.
- Dieser Ansatz wird häufig im Bereich des Operations Research angewandt.

Dynamische Programmierung II

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.

Dynamische Programmierung III

Optimalitätsprinzip:

- 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 $k < n$ betrachtet werden, sondern nur ‚lohnende', weil in obigem Sinne optimale Teillösungen berechnet werden.

Dynamische Programmierung IV

Bereits bekannte Beispiele für dieses Lösungskonzept:

Algorithmen von Floyd und Warshall

- **Algorithmus von Floyd** zur Lösung des ‚all pairs shortest path'-Problems
- **Algorithmus von Warshall** zur Berechnung der transitiven Hülle in gewichteten Graphen

Die aktuellen Teilprobleme werden dabei unter Rückgriff auf bereits berechneter Teilprobleme gelöst.

Dynamische Programmierung V

Probleme bei 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.

Dynamische Programmierung VI

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.

Dynamische Programmierung VII

→ die dynamische Programmierung berechnet Informationen zur Lösung von Teilproblemen

→ setzt diese ‚in geeigneter Weise' nach dem Optimalitätsprinzip zu den zur Lösung größerer Probleme benötigten Informationen zusammen

→ setzt diese schließlich zur Lösung des Gesamtproblems zusammen.

Allgemein:

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