5 Ausgewählte Themen

5.1 Dynamische Programmierung

5.1.1 Typische Vorgehensweise bei der dynamischen Programmierung

5 Schritte:

1. Definition des Optimierungskriteriums
2. Definition von Teilproblemen und der entsprechenden Hilfsgröße
3. Zerlegung in Teilprobleme
   $\Rightarrow$ führt zu Rekursionsgleichung
4. Auswertung der Rekursionsgleichung
   mittles Tabelle (iterativ oder mit Memoization)
5. Traceback: Rekonstruktion der optimalen Lösung

5.1.2 Rucksackproblem

Es gibt mehrere Varianten des Rucksackproblems.

Wir betrachten zunächst die Variante, die als 0/1-Rucksackproblem bekannt ist:

$$\begin{array}{rl} \text{Objekte:} & 1, \ldots, i, \ldots, n \quad \in \mathbb{N} \\ \text{Größe/Gewicht:} & g_1, \ldots, g_i, \ldots, g_n \quad \in \mathbb{N} \\ \text{Wert:} & v_1, \ldots, v_i, \ldots, v_n \quad \in \mathbb{N} \end{array}$$

Aufgabe: Wähle die Objekte so aus, daß ihr Gesamtwert möglichst groß ist und sie in einen Rucksack der Größe G passen.

Mathematische Formulierung:
Für jedes Objekt i ist eine binäre Entscheidung zu treffen:

$$\begin{array}{rcl} a_i & \in & \{0, 1\} \\ a_1^n & := & a_1, \ldots, a_i, \ldots a_n \end{array}$$

1. Definition des Optimierungskriteriums

$$\max_{a_1^n} \left\{ \sum_{j=1}^{n} a_j v_j : \sum_{j=1}^{n} a_j g_j \leq h \right\}$$

2. Definition von Teilproblemen und der entsprechenden Hilfsgröße

Betrachte Wert $w(i,h)$ eines partiell gefüllten Rucksacks der Größe h wobei nur die Objekte $1,\ldots,i$ verwendet werden:

Hilfsgröße:

$$w(i,h) \; := \; \max_{a_1^i} \left\{ \sum_{j=1}^{i} a_j v_j : \sum_{j=1}^{i} a_j g_j \leq h \right\}$$

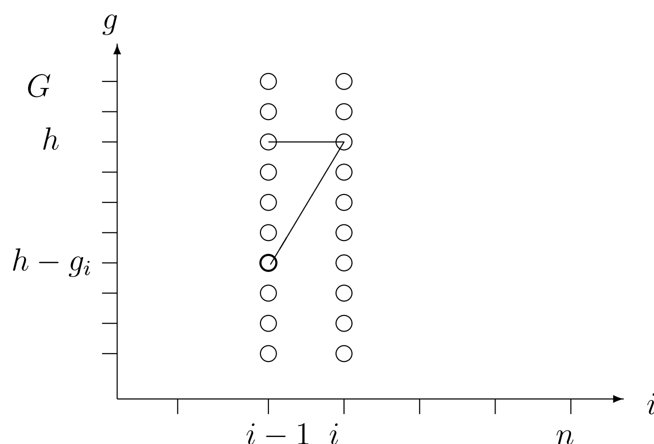
3. Zerlegung in Teilprobleme

Ansatz: $\{1,\ldots,i-1\} \rightarrow \{1,\ldots,i\}$

$$\begin{aligned} w(i,h) \; &:= \max_{a_1^i} \quad \{\text{“}a_i = 0\text{''}, \text{“}a_i = 1\text{''}\} \\ &= \max \quad \left\{ 0 + \max_{a_1^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j : \sum_{j=1}^{i-1} a_j g_j \leq h \right\}, \right. \\ &\qquad\qquad \left. v_i + \max_{a_1^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j : \sum_{j=1}^{i-1} a_j g_j \leq h - g_i \right\} \right\} \\ &= \max \quad \{w(i-1,h), v_i + w(i-1,h-g_i)\} \end{aligned}$$

offensichtliche Vorraussetzung für die Rekursionsgleichung:

$$\begin{aligned} w(i-1,h) \; &\Rightarrow \; i-1 \geq 0 \\ w(i-1,h-g_i) \; &\Rightarrow \; h-g_i \geq 0 \end{aligned}$$



4. Auswertung der Rekursionsgleichung

Randbedingungen:

$$\begin{array}{rll} i = 0: & w(i,h) = 0 & \forall h \\ i > 0, h < g_i: & w(i,h) = w(i-1,h) & \forall h \end{array}$$

Implementierung: 2 Schleifen

for $i = 1, \ldots, n$ do
  for $h = 0, \ldots, G$ do
    $w[i,h] = \max\{\ldots\}$
    $p[i,h] = \arg\max\{\ldots\}$ “predecessor”

5. Traceback: Rekonstruktion der optimalen Lösung

Rekonstruktion der optimalen Lösung mittels der Tabelle $p[i,h]$.

Variante des Rucksackproblemes

Wir betrachten die folgende Variante des Rucksackproblemes:

Von jedem Objekt sind beliebig viele Exemplare vorhanden

$$\begin{array}{rll} \text{Objekte:} & 1, \ldots, i, \ldots, n & \in \mathbb{N} \\ \text{Größe/Gewicht:} & g_1, \ldots, g_i, \ldots, g_n & \in \mathbb{N} \\ \text{Wert:} & v_1, \ldots, v_i, \ldots, v_n & \in \mathbb{N} \end{array}$$

Die Größe des Rucksackes sei G.

1. Definition des Optimierungskriteriums

Gesucht ist eine Indexfolge $i_1, \ldots, i_M$ mit $i_k \in \{1, \ldots, n\}$. Diese gibt die Reihenfolge an, in der die Exemplare in den Rucksack eingepackt werden. Das Optimierungskriterium ist dann:

$$\max_{M, i_1^M} \left\{ \sum_{k=1}^{M} v_{i_k} : \sum_{k=1}^{M} g_{i_k} \leq h. \right\}$$

2. Definition von Teilproblemen und der entsprechenden Hilfsgröße

Betrachte den Wert $w(h)$ eines partiell gefüllten Rucksack der Größe h und packe Gegenstände hinein:

$$w(h) \quad := \quad \max_{m, i_1^m} \left\{ \sum_{k=1}^{m} v_{i_k} : \sum_{k=1}^{m} g_{i_k} \leq h. \right\}$$

3. Zerlegung in Teilprobleme

$$\begin{aligned}
w(h) &:= \max_{m,i_1^m} \left\{ \sum_{k=1}^{m} v_{i_k} : \sum_{k=1}^{m} g_{i_k} \leq h. \right\} \\
&= \max_{m,i_1^m} \left\{ v_{i_m} + \underbrace{\max_{m-1,i_1^{m-1}} \left\{ \sum_{k=1}^{m-1} v_{i_k} : \sum_{k=1}^{m-1} g_{i_k} \leq h - g_{i_m} \right\}}_{w(h-g_{i_m})} \right\} \\
&= \max_{i_m} \{ v_{i_m} + w(h - g_{i_m}) \} \\
&= \max_{i} \{ v_i + w(h - g_i) \} \qquad \text{für } h - g_i \geq 0
\end{aligned}$$

4. Auswertung der Rekursionsgleichung

Randbedingungen:

$$w(h) = 0 \qquad \text{für } h = 0, \ldots, g_{min} - 1 \text{ mit } g_{min} = \min_{i=1,\ldots,n} \{g_i\}$$

Implementierung:

for $h = 1, \ldots, G$ do
  $w[h] = \max_i \{\ldots\}$
  $p[h] = \arg\max_i \{\ldots\}$ "predecessor"

5.1.3 Beispiele für dynamische Programmierung

- eindimensionale Tabelle
  - DAG
  - maximale Teilsumme (hier nicht behandelt)
- zweidimensionale Tabelle
  - Rucksackproblem
  - approximate string matching
    LCS, SCS, edit distance
  - endliche Automaten
- zweidimensionale Tabelle mit Indices von "gleichem" Typ
  - Klammerung für Matrizenprodukt
  - Suchbaum
  - CYK-Parser für kontextfreie Gramatiken (hier nicht behandelt)
  - Floyd-Algorithmus
- komplizierter
  - Traveling Salesman Problem
  - bei entsprechender Interpretation:
    * single-source-best-path (Dijkstra)
    * minimum-spanning tree (Prim/Kruskal)

(Anmerkung: Hier ist die Rekursionsgleichung von 2 Variablen abhängig. Es kann durchaus sein, dass sich das Feld für die dynamische Programmierung in einer Dimension darstellen lässt.)

5.2 Exact String Matching

(Suchen von Strings in Texten, Erkennen von Zeichenmustern)

- Zeichen:
  - Bit
  - Bitmuster
  - Byte: 8 Bits: (ASCII-)Zeichen:
    - Buchstaben
    - Ziffern
    - Sonderzeichen
    - (Steuerzeichen)
- Zeichenkette (String): lineare Folge von Zeichen
- Text-String: Folge von Text-Zeichen, (Bit oder ASCII)

Anwendungen: Editor, Text-Verarbeitung

Aufgabe:
Gegeben sei ein Text-String $a[1..N]$ und ein Suchmuster $p[1..M]$:
Finde ein Vorkommen (oder alle) von $p[1..M]$ in $a[1..N]$

5.2.1 Naiver Algorithmus (brute force)

Teste alle Positionen $i = 1...N$ des Text-Strings
Falls ein Mismatch auftritt, gehe eine Position weiter.

Algorithmus:

- Return-Wert: gefunden: Position $(0 < i \leq N)$
  negativ: $N + 1$
- maximal $M \cdot (N - M + 1) \simeq N \cdot M$ Zeichenvergleiche $(M \ll N)$
- Im ungünstigsten Fall bestehen $a[1..N]$ und $p[1..M]$ nur aus Nullen und einer abschließenden Eins. Dann müssen alle M Zeichen $p[1..M]$ in jeder Position überprüft werden.
- In der Praxis werden deutlich weniger Zahlenvergleiche als $(N \cdot M)$, (eher $M \cdot const$ Zahlenvergleiche) benötigt wegen vorzeitigen Abbruchs.

Beispiel: brute force
$a[..]$ = ,,A STRING SEARCHING EXAMPLE CONSISTING OF"
$p[..]$ = ,,STING"
(n): Positionen, wo die ersten n Zeichen passen:

```
A STRING SEARCHING EXAMPLE CONSISTING OF
  STING (2)
           STING (1)
                                    STING (1)
                                      STING (5)
```

```
function brutesearch : integer;
var i, j : integer;
begin
    i := 1; j := 1;
    repeat
        if a[i] = p[j]
            then begin i := i + 1; j := j + 1 end
            else begin i := i - j + 2; j := 1 end
    until (j > M) or (j > N)
    if j > M then return i - M ; else return i := i ;
end;
```

```
 1  0  0  1  1  1  0  1  0  0  1  0  1  0  0  0  1  0  1  0  0  1  1  1  0  0  0  1  1  1
 1  0 [1] 0  0  1  1  1
   [1] 0  1  0  0  1  1  1
      [1] 0  1  0  0  1  1  1
          1 [0] 1  0  0  1  1  1
             1 [0] 1  0  0  1  1  1
                1  0  1  0  0  1 [1] 1
                  [1] 0  1  0  0  1  1  1
                      1  0 [1] 0  0  1  1  1
                        [1] 0  1  0  0  1  1  1
                           [1] 0  1  0  0  1  1  1
                               1  0  1  0  0 [1] 1  1
                                 [1] 0  1  0  0  1  1  1
                                     1  0 [1] 0  0  1  1  1
                                       [1] 0  1  0  0  1  1  1
                                          [1] 0  1  0  0  1  1  1
                                             [1] 0  1  0  0  1  1  1
 1  0  0  1  1  1  0  1  0  0  1  0  1  0  0  0 [1][0][1][0][0][1][1][1] 0  0  0  1  1  1
```

Abbildung 5.1: Brute Force String Suche in einem binären String.

5.2.2 Knuth-Morris-Pratt-Algorithmus (KMP-Algorithmus)

Verbesserung des naiven Algorithmus:

- Wenn an Position j des Musters ein ,,Mismatch" aufgetreten ist, haben die letzten $(j-1)$ Zeichen im Text mit denen des Musters übereingestimmt.
- Eine doppelte Überprüfung von Zeichen im Text-String wird vermieden, indem die vorab bekannte ,,Struktur" des Musters ausgenutzt wird.

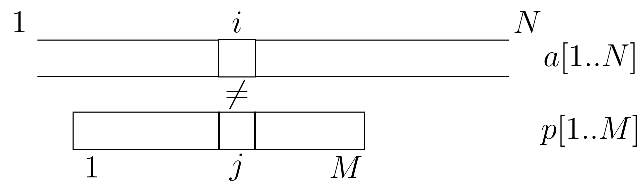
Veranschaulichung:

```
                 i-j+1
     1             |            i                        N
a[i] ------------ |x|x|x|x|x|x|x|x|x|x|x|x| ------------
p[j] ------------ |y|y|y|y|y|y|y|y|y|y|y|y| ------------
                   1            j       M
```

Definiere ein Array $next[1..M]$, so daß um $next[j]$ Positionen ,,zurückgegangen" wird, falls ein Mismatch $a[i] \neq p[j]$ auftritt. Wird die Suche bei (i, j) wegen $p[j] \neq a[i]$

abgebrochen, dann wird der nächste Vergleich gemacht für:

$$\begin{array}{rll} a[i+1] \text{ und } p[1] & \text{falls} & j=1 \\ a[i] \text{ und } p[next[j]] & \text{falls} & j>1 \end{array}$$



Darüber hinaus wird so ein Zurückgehen („Zurücksetzen”, „backing up”) im Text $a[1..n]$ vermieden.

KMP-Algorithmus

Wenn ein Mismatch auftritt, also $a[i] \neq p[j]$ für $j > 1$, dann wäre die neue Position im Textstring: $i := i - next[j] + 1$.
Da aber die ersten $(next[j] - 1)$ Textzeichen ab dieser Position zu den Zeichen des Musters passen, bleibt i unverändert und $j := next[j]$.

$j = 1$ oder $a[i] \neq p[1]$:

- kein overlap
- Wunsch: $i := i + 1$ und $j := 1$
- Trick: $next[1] := 0$ [Array vergrößern: $p[0..M]$ wegen „OR”]

Berechnung von $next[1..M]$:

$next[j]$:„Vergleiche die Zeichen $p[1..j]$ mit sich selbst”

- Schiebe eine Kopie der ersten $(j - 1)$ Zeichen über das Muster selbst von links nach rechts.
- Start: Das erste Zeichen der Kopie ist über dem zweitem Zeichen des Musters.
- Stop: Falls alle überlappenden Zeichen passen oder es keine passenden gibt:
  $next[j] := 1 +$ „Anzahl der passenden Zeichen”
- Definition: $next[1] := 0$

Erzeugung der Tabelle $next[1..M]$:

- elementar $O(M^2)$

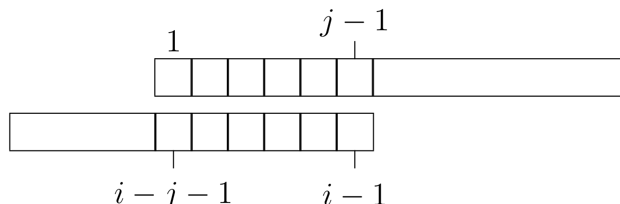
- KMP angewandt auf $p[1..M]$

```
j   next[j]

2   1        1 0 1 0 0 1 1 1
               1 0 1 0 0 1 1 1
3   1        1 0 1 0 0 1 1 1
                 1 0 1 0 0 1 1 1
4   2          1 0 1 0 0 1 1 1
               1 0 1 0 0 1 1 1
5   3        1 0 1 0 0 1 1 1
                 1 0 1 0 0 1 1 1
6   1        1 0 1 0 0 1 1 1
                       1 0 1 0 0 1 1 1
7   2        1 0 1 0 0 1 1 1
                       1 0 1 0 0 1 1 1
8   2        1 0 1 0 0 1 1 1
                         1 0 1 0 0 1 1 1
```

Neustart-Positionen für Knuth-Morris-Pratt Suche

$p[i] = p[j]$: i und j werden erhöht:



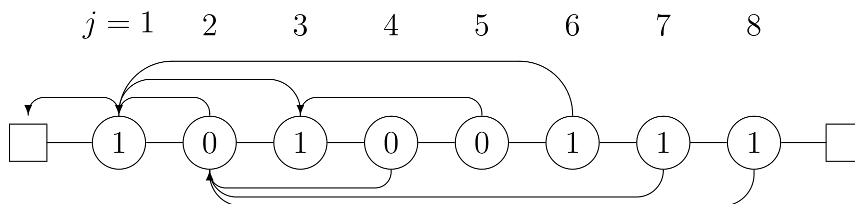
$(j-1)$ passende Zeichen $\Rightarrow$ $next[i] = j$

```
function kmpsearch : integer;
var i, j : integer;
begin
    i := 1; j := 1; initnext;
    repeat
        if (j = 0) or (a[i] = p[j])
        then begin
            i := i + 1; j := j + 1
        end
```

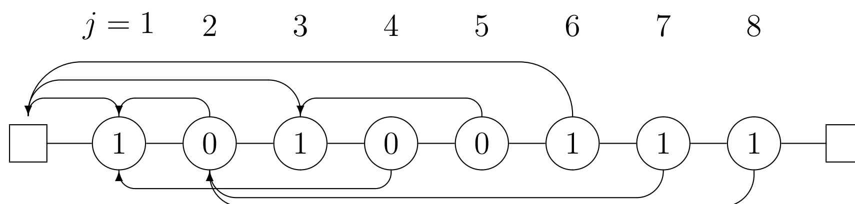
```
            else j := next[j]
      until (j > M) or (i > N)
      if j > M
      then kmpsearch := i - M
      else kmpsearch := i
end;
```

```
procedure initnext;
var i, j :integer;
begin
      i := 1; j := 0; next[1] := 0;
      repeat
            if (j = 0) or (p[i] = p[j])
            then begin i := i + 1; j := j + 1; [next[i] := j] end
            else j := next[j]
      until i > M
end;
```

Verbesserung: Ersetze [next[i] := j] durch:
if p[i] <> p[j] **then** next[i] := j **else** next[i] := next[j]



Verbesserte Version:



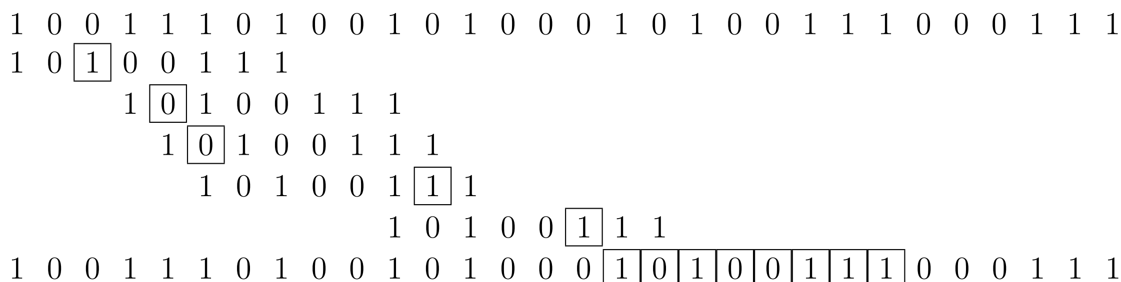
$1 \rightarrow 0$
$2 \rightarrow 1$
$3 \rightarrow 1 \rightarrow 0$
$4 \rightarrow 2 \rightarrow 1$
$5 \rightarrow 3$
$6 \rightarrow 1 \rightarrow 0$
$7 \rightarrow 2$
$8 \rightarrow 2$

Komplexität: KMP

maximal $N + M$ Zeichenvergleiche:
Für jedes $i = 1..N$:
entweder $j := j + 1$
oder $j := next[j]$

Praxis:

- In der Regel ist KMP kaum besser als der naive Algorithmus (ohne selbstwiederholende Teile im Muster kein Vorteil für KMP).
- Vorteil bei externen Speichern: Zurücksetzen im Text ist unnötig, da der Index i nur wachsen kann.



Knuth-Morris-Pratt String Suche in binärem Text

5.3 Approximate String Matching

(deutsch: Approximativer symbolischer Vergleich)

Motivation:

- Schreibfehler („phonetische" Schreibweise)
- Mustererkennung und Sprackerkennung
- Bioinformatik: DNA/RNA-Sequenzen, Genom-Sequenzen

5.3.1 Longest Common Subsequence

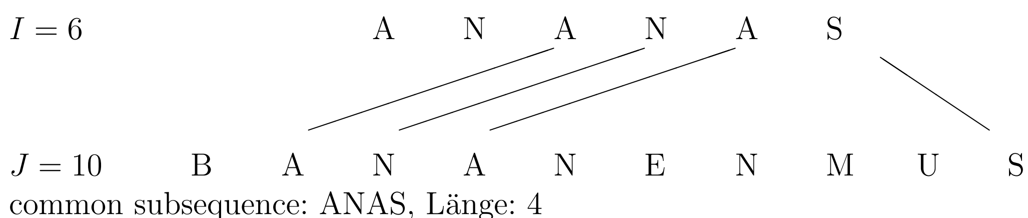
Gegeben sind zwei (Zeichen-)Folgen:

$$\begin{aligned} x_1^I &:= x_1, \ldots, x_i, \ldots, x_I \\ \text{und } y_1^J &:= y_1, \ldots, y_i, \ldots, y_J \end{aligned}$$

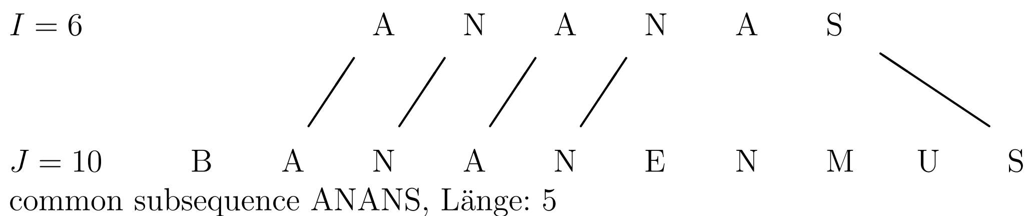
Aus jeder dieser beiden Folgen kann durch Streichen von Zeichen eine sog. Teilfolge erzeugt werden.

Aufgabe: Bestimme die längste gemeinsame Teilfolge (longest common subsequence) der beiden Folgen.

Beispiel: 'ANANAS' und 'BANANENMUS'



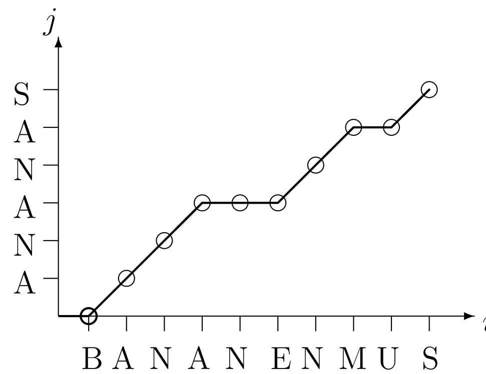
common subsequence: ANAS, Länge: 4



common subsequence ANANS, Länge: 5

nicht erlaubt:

Doppelbelegung eines Symbols und Überschneiden der Zuordnung



Pfad in der Ebene (i, j)

$$k \to (i_k, j_k) \quad k = 1, \ldots, K$$

Bedingungen an den Pfad:

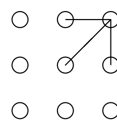
- Monotonie:

$$\begin{aligned} i_{k-1} &\leq i_k \\ j_{k-1} &\leq j_k \end{aligned}$$

- Auslassen ist verboten:

$$\begin{aligned} i_k &\in \{i_{k-1}, i_{k-1} + 1\} \\ j_k &\in \{j_{k-1}, j_{k-1} + 1\} \end{aligned}$$

zusammen:



Randbedingungen:

$$\begin{aligned} i_1 = 0, &\quad j_1 = 0 \\ i_K = I, &\quad j_K = J \end{aligned}$$

1. Definition des Optimierungskriteriums

$$\max_{K, i_1^K, j_1^K} \sum_{k=1}^{K} \delta(i_{k-1}+1, i_k) \cdot \delta(j_{k-1}+1, j_k) \cdot \delta(x_{i_k}, y_{j_k})$$

$$\text{Kronecker Delta: } \delta(a,b) = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases}$$

2. Definition von Teilproblemen und der entsprechenden Hilfsgröße

LCS für Teilfolgen x_1^i und y_1^j und des entsprechenden Optimierungskriteriums:

$$D(i,j) \quad := \quad \max_{n, i_1^n, j_1^n : i_n = i, j_n = j} \left\{ \sum_{k=1}^{n} \delta(i_{k-1}+1, i_k) \cdot \delta(j_{k-1}+1, j_k) \cdot \delta(x_{i_k}, y_{j_k}) \right\}$$

3. Zerlegung in Teilprobleme

Mit den Pfadbedingungen erhält man die Rekursionsgleichung:

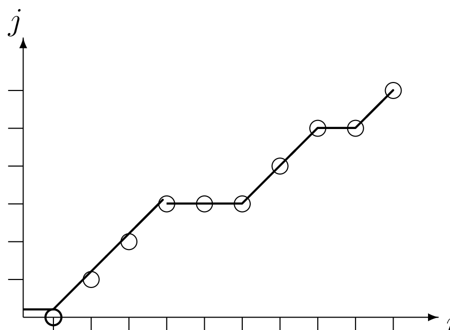
$$\begin{aligned} D(i,j) \quad &:= \quad \max \left\{ \text{“ ”} \right\} \\ &= \quad \max\{D(i-1,j), D(i-1,j-1) + \delta(x_i, y_j), D(i, j-1)\} \end{aligned}$$

4. Auswertung der Rekursionsgleichung

for $i = 0, \ldots, I$ do D[i,0]:=0
for $j = 0, \ldots, J$ do D[0,j]:=0
for $i = 1, \ldots, I$ do
  for $j = 1, \ldots, J$ do
    $D[i,j] := \max\{\ldots\}$
    $P[i,j] := \arg\max\{\ldots\}$

Komplexität: $I \cdot J$

5. Traceback: Rekonstruktion der optimalen Lösung



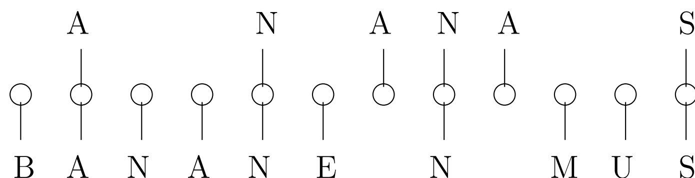
Für die LCS muß gelten:

$$\begin{aligned} \delta(x_i, y_j) \quad &= \quad 1 \\ \text{und } p(i,j) \quad &= \quad \text{“diagonal”} \end{aligned}$$

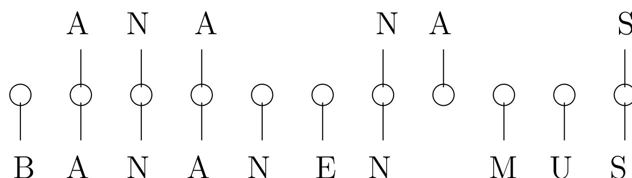
5.3.2 Shortest Common Supersequence

Aufgabe:
Bestimme die kürzeste gemeinsame Oberfolge (shortest common supersequence = SCS), d.h. die kürzeste Folge, die beiden Folgen x_1^I und y_1^J als Teilfolgen enthält.

Beispiele:



CS=BANANEANAMUS
$|CS|$=12



CS=BANANENAMUS
$|CS|$=11

Es gilt: $\max\{I, J\} \leq |SCS| \leq I + J$

Optimierungskriterium:

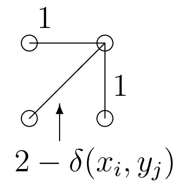
Pfad in der Ebene:

$$k \rightarrow (i_k, j_k), \qquad k = 1, \ldots K$$

$$\min_{K, j_1^K, i_1^K} \left\{ \sum_{k=1}^{K} [1 + \delta(i_{k-1} + 1, i_k) \cdot \delta(j_{k-1} + 1, j_k) \cdot (1 - \delta(x_{i_k}, y_{j_k}))] \right\}$$

Pfadbedingungen: wie bei LCS

Veranschaulichung:



Rekursionsgleichung:

$$\begin{aligned} D(i,j) &= \min\{ && D(i-1,j)+1, D(i,j-1)+1, \\ & && D(i-1,j-1)+2-\delta(x_i,y_j)\} \end{aligned}$$

Es gilt:

$$|SCS| = I + J - |LCS|$$

5.3.3 Edit Distance

1966: Levenshtein: Levenshtein distance
1974: Wagner: edit distance

Editieroperationen:

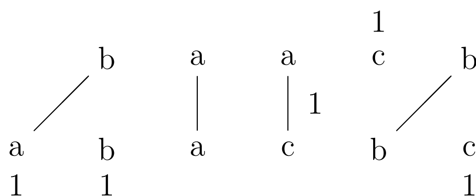
- Deletions
- Insertions
- Substitutions

Bedingungen (wie bei LCS):

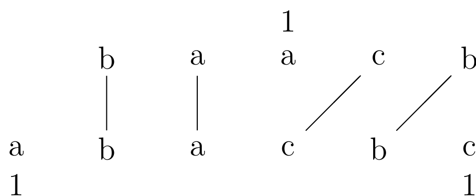
- Überschneidungen sind verboten.
- Die wechselseitige Monotonie der beiden Symbolfolgen wird beibehalten.

Beispiel: A = baacb; B = abacbc

Zuordnung:



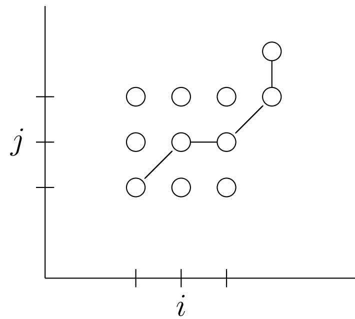
andere und bessere Zuordnung:



Aufgabe:
Bestimme die Zuordnung mit minimalen Kosten. Der Einfachheit halber werden Einheitskosten für Deletions, Insertions und Substitutions festgelegt, so daß gilt:

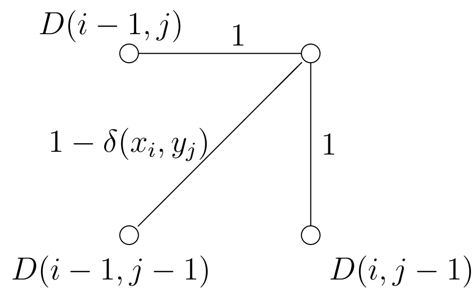
$$\text{Kosten} = \text{Anzahl der Editieroperationen}$$

Eine Zuordnung ist dabei ein „Pfad" zwischen $x_1...x_i...x_I$ und $y_1...y_j...y_J$ in der Gitterebene $\{i, j : i = 0, \ldots, I, j = 0, \ldots, J\}$.

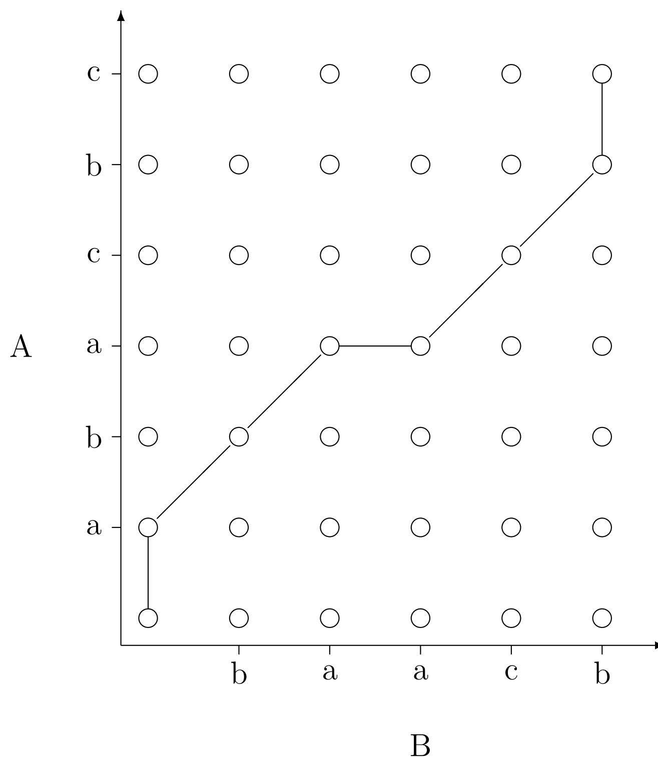


Definiere Hilfsgröße:

$D[i, j] :=$ Kosten der besten Zuordnung für die Teilstrings $x_1 \ldots x_i$ und $y_1 \ldots y_j$



Beispiel: A = abacbc; B = baacb



Rekursionsgleichung der dynamischen Programmierung:

$$D[i,j] = \min\{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+1-\delta(x_i,y_j)\}$$
$$\text{für } 0 < i \leq I,\ 0 < j \leq J$$
$$\text{mit } \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}$$

Initialisierung:

$$\begin{aligned} D[0,0] &= 0 \\ D[0,j] &= j \text{ , für } 1 \leq j \leq J \\ D[i,0] &= i \text{ , für } 1 \leq i \leq I \end{aligned}$$

Auswertung in zwei Schleifen (wie bei LCS):

for $i = 0, \ldots, I$ do D[i,0]:=i
for $j = 0, \ldots, J$ do D[0,j]:=j
for $i = 1, \ldots, I$ do

for $j = 1, \ldots, J$ do
    $D[i, j] := \min\{\ldots\}$
    $P[i, j] := \arg\min\{\ldots\}$

⇒ Komplexität:

- Zeit: $O(I \cdot J)$
- Platz: $O(I \cdot J)$
- ohne explizites Berechnen der Zuordnung: Platz: $O(\min(I, J))$

Vergleiche: Gesamtzahl der möglichen Zuordnungen ca. 2^I bis 3^I
grobe Abschätzung, aber in jedem Fall exponentiell

modifizierte Aufgabe: approximatives Pattern Matching und Suchen

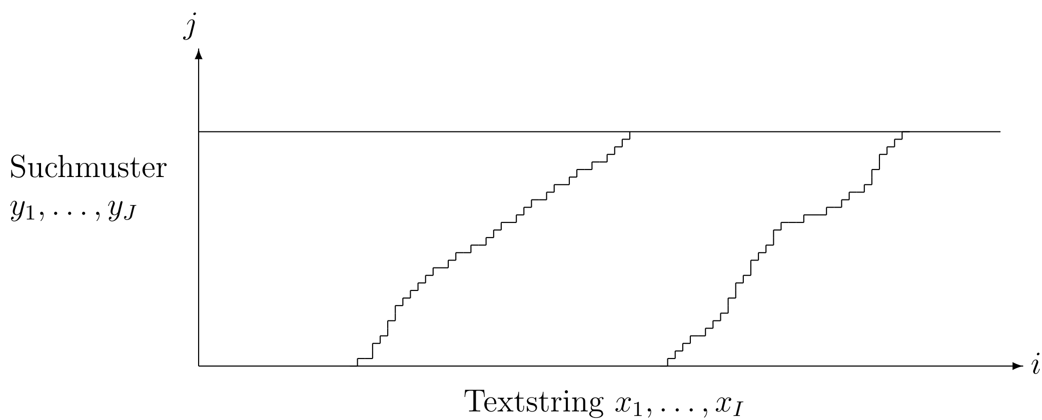


Abbildung 5.2: Approximatives Stringsuche und Pattern Matching

Das Suchmuster kommt im Textstring “näherungsweise” vor:

- Häufigkeit des Vorkommens ist unbekannt
- Positionen sind unbekannt

5.3.4 Verfeinerungen der Edit Distance

a) Gewichte Die Editieroperationen haben Gewichte:

$$f_{del}, f_{sub}, f_{ins} \text{ z.B.} \in \{\frac{1}{2}, 1, 2\}$$

Rekursionsgleichung:

$$\begin{aligned} D(i,j) = \quad \min\{ \quad & D(i-1,j) + f_{del} \cdot 1, D(i,j-1) + f_{ins} \cdot 1, \\ & D(i-1,j-1) + f_{sub} \cdot (1 - \delta(x_i, y_j))\} \end{aligned}$$

Spezialfall:

$$f_{del} = f_{ins} = 1$$
$$f_{sub} = 2$$

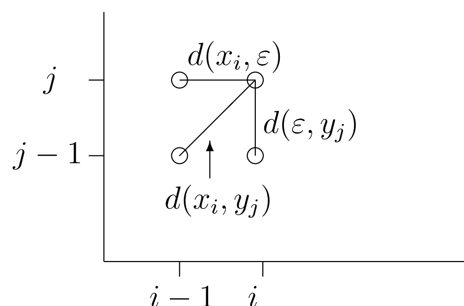
$$|edit| = I + J - 2 \cdot |LCS|$$

b) Lokale Bewertungen Die Editieroperationen haben *symbol-abhängige* Kosten (ε = leeres Symbol):

deletion:	$d(x_i, \varepsilon)$
substitution:	$d(x_i, y_j)$ mit $d(x_i, y_j) = 0 \Leftrightarrow x_i = y_j$
insertion:	$d(\varepsilon, y_j)$

Rekursionsgleichung:

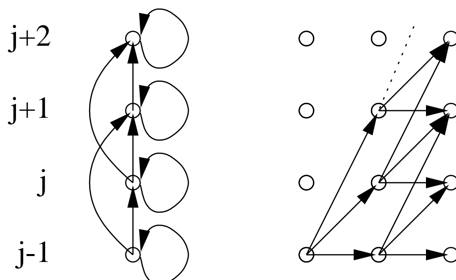
$$\begin{aligned} D(i,j) \quad = \min\{ \quad & D(i-1,j) + d(x_i, \varepsilon), D(i,j-1) + d(\varepsilon, y_j), \\ & D(i-1,j-1) + d(x_i, y_j)\} \end{aligned}$$



c) Endliche Automaten

Änderungen gegenüber Edit Distance:

1. Symmetrie wird aufgegeben.
2. Modell: In das Modell werden die drei Fehlerarten integriert (Deletions, Insertions, Substitutions).
3. Ein Pfad durch das Modell entspricht einem betrachteten (aktuellen) String.



Definition von Kosten: in Abhängigkeit den "Zuständen" j:

1. Insertions und Deletions in Abhängigkeit vom Ausgangszustande j' und dem erreichten Zustand j:
$$t(j', j)$$
2. Substitutions in Abhängigkeit vom Zustand j:
$$d(x_i, j) \qquad x_i : i\text{-tes Symbol im String}$$

Dynamische Programmierung: für String $x_1, \ldots, x_i, \ldots, x_I$

$$D(i, j) = d(x_i, y_j) + \min\{D(i-1, j') + t(j', j) : j'\}$$

Kosten: $d(\ldots)$ und $t(\ldots)$ können auf negativen Logarithmus von Wahrscheinlichkeiten zurückgeführt werden. (*Terminologie in der Bioinformatik: Scores*)

Erweiterung: "regelmäßige" Struktur des Automaten wird aufgegeben, beliebige Strukturen werden zugelassen.

5.4 Traveling Salesman Problem

5.4.1 Problemstellung

deutsch: Problem des Handlungsreisenden, optimale Rundreise

Aufgabe: Gegeben seien n Städte mit Kostenmatrix $d_{ij} \geq 0, i = 1, \ldots, n, j = 1, \ldots, n$. Es soll diejenige Rundreise bestimmt werden, die jede Stadt genau einmal besucht und die Summe der Kosten minimiert.

Formal: gewichteter, bewerteter Graph mit Kostenfunktion

$$\begin{aligned} d_{ij} &\geq 0: \text{ Kosten für den Weg von Knoten } i \text{ zu Knoten } j \\ V &= \{1, \ldots, n\}: \text{ Knotenmenge} \end{aligned}$$

Beachte:

1. Die Kostenmatrix muß *nicht* symmetrisch sein, d.h. $d_{ij} \neq d_{ji} \quad j \neq i$ ist möglich.
2. Dreiecksungleichung: $d_{ij} \leq d_{ik} + d_{kj}$ wird nicht vorrausgesetzt.
3. Übliche Konvention: $d_{ij} = \infty$, falls es *keine* Kante von i nach j gibt. Insbesondere: $d_{ii} = \infty \quad \forall i \in V$. Dies kann unter Umständen bedeuten, daß für die gegebene Kostenmatrix gar keine Rundreise existiert.

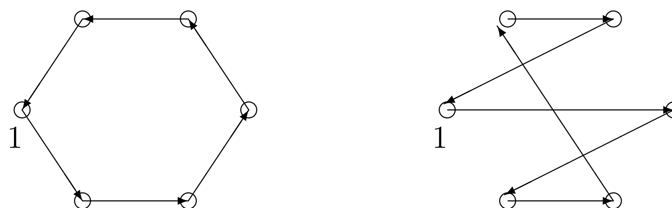
Jede Rundreise läßt sich als Permutation π darstellen:

$$\begin{aligned} \pi: \quad & V \to V \\ & i \to \pi(i) \end{aligned}$$

Damit lautet das Optimierungsproblem:

$$\min_{\pi} \left\{ d_{\pi(n),\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} \right\}$$

Beispiele für verschiedene Rundreisen im gleichen Graphen.:



5.4.2 Exhaustive Search

anderer Name: brute force search, erschöpfende Suche

Auflisten alle Permutationen und Berechnen der Summe. Es genügt, nur Permutationen mit $\pi(1) = 1$ zu betrachten (d.h. Knoten 1 ist der Startknoten). Damit ergibt sich

$$(n-1)! \text{ Permutationen mit } n \text{ Additionen pro Permutation}$$

Komplexität:

$$n! \text{ Additionen und } (n-1)! \text{ Vergleiche}$$

Eine geringfügige Verbesserung der Komplexität kann man erreichen, wenn man die Permutationen als Baum darstellt: An jedem Knoten des Baumes kann man einen *noch nicht* besuchten Graphknoten auswählen. Dieser Baum repräsentiert die möglichen Lösungen und wird als Lösungsbaum bezeichnet.

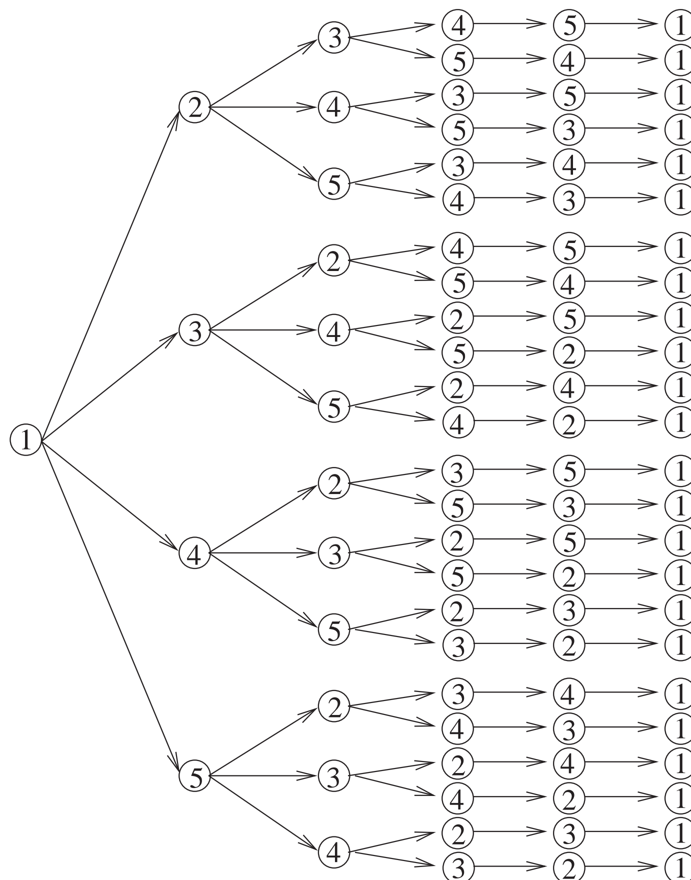


Abbildung 5.3: Lösungsbaum für das TSP mit 5 Knoten.

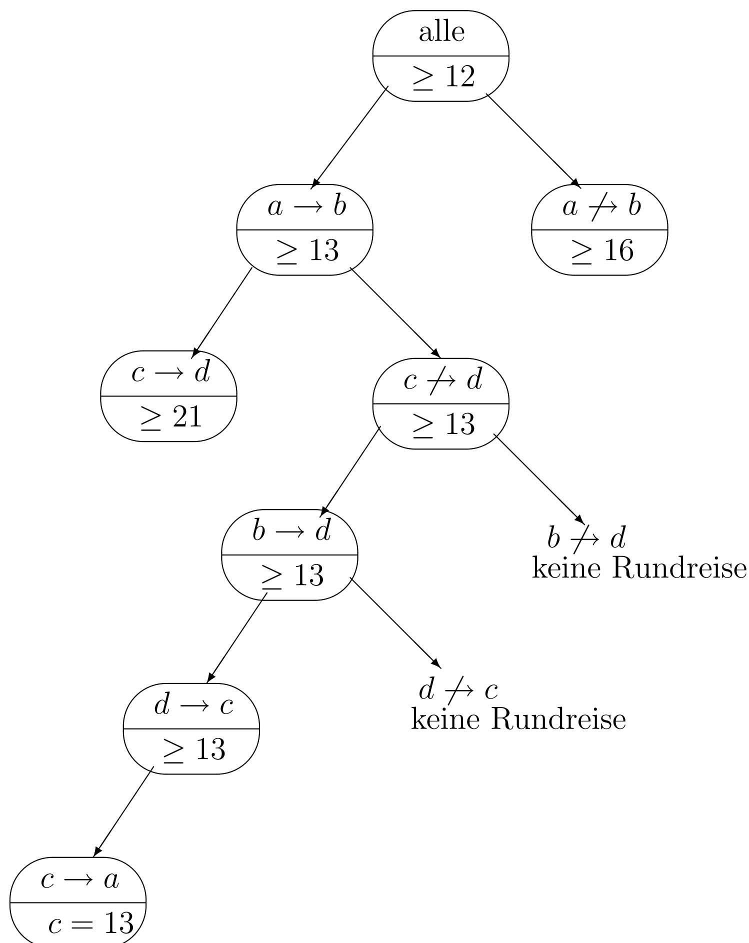
5.4.3 Branch & Bound

Idee: Im Lösungsbaum versuchen wir, für jeden Knoten (d.h. partielle Rundreise) eine untere Schranke (=bound) für die Kosten des entsprechenden Pfades zu berechnen. Wir wählen den Knoten mit der kleinsten unteren Schranke, verzweigen (=branch) und berechnen eine neue untere Schranke (=bound). Usw.

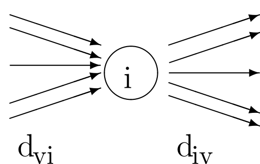
Beispiel: Kostenmatrix

		nach			
		a	b	c	d
von	a	∞	2	8	6
	b	2	∞	6	4
	c	4	5	∞	5
	d	8	7	3	∞

Für diese Kostenmatrix werden wir folgenden "Branch-and-Bound"-Lösungsbaum berechnen:



Für jeden Knoten (Stadt) berechnen wir eine *untere Schranke* für ihren Beitrag zu den Gesamtkosten einer Rundreise. Dann betrachten wir die einlaufenden und auslaufenden Kanten:



Von jeder Stadt muß eine Kante *ausgehen*, so daß wir eine neue Kostenmatrix $\tilde{d}_{ij}$ definieren:

$$\tilde{d}_{ij} := d_{ij} - \min_v d_{iv}$$

In jede Stadt muß eine Kante *einlaufen*, so daß wir eine neue Kostenmatrix $\tilde{\tilde{d}}_{ij}$ definieren:

$$\tilde{\tilde{d}}_{ij} := \tilde{d}_{ij} - \min_v \tilde{d}_{vi}$$

Implementierung:

- Subtrahiere von jeder Zeile das Minimum → neue Kostenmatrix
- Subtrahiere von jeder Spalte dieser neuen Kostenmatrix das Minimum.

$$\begin{bmatrix} \infty & 2 & 8 & 6 \\ 2 & \infty & 6 & 4 \\ 4 & 5 & \infty & 5 \\ 8 & 7 & 3 & \infty \end{bmatrix} \underset{\substack{\text{minimum} \\ 2+2+4+3=11}}{\overset{\text{Zeilen-}}{\longrightarrow}} \begin{bmatrix} \infty & 0 & 6 & 4 \\ 0 & \infty & 4 & 2 \\ 0 & 1 & \infty & 1 \\ 5 & 4 & 0 & \infty \end{bmatrix} \underset{\substack{\text{minimum} \\ 0+0+0+1=1}}{\overset{\text{Spalten-}}{\longrightarrow}} \begin{bmatrix} \infty & 0 & 6 & 3 \\ 0 & \infty & 4 & 1 \\ 0 & 1 & \infty & 0 \\ 5 & 4 & 0 & \infty \end{bmatrix}$$

$\boxed{\text{Bound}=12}$

Wir suchen einen 0−Eintrag, z.B. $a \to b$. Für die Runreisen mit $a \to b$ streichen wir die erste Zeile ("von a nach ...") und zweite Spalte ("von ... nach b") und setzen den (b, a)−Eintrag auf ∞. Damit ergibt sich die neue Kostenmatrix:

	a	c	d
b	∞	4	1
c	0	∞	0
d	5	0	∞

$\Rightarrow$

	a	c	d
b	∞	3	0
c	0	∞	0
d	5	0	∞

$0 + 1 = 1$

$\boxed{\text{Bound}=13}$

Für die Rundreisen mit $a \not\to b$ setzen wir den (a, b)-Eintrag auf ∞ und erhalten:

$$\begin{bmatrix} \infty & \infty & 6 & 3 \\ 0 & \infty & 4 & 1 \\ 0 & 1 & \infty & 0 \\ 5 & 4 & 0 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & 3 & 0 \\ 0 & \infty & 4 & 1 \\ 0 & 0 & \infty & 1 \\ 5 & 3 & 0 & \infty \end{bmatrix}$$

$3 + 1 + 0 + 0 = 4$

$\boxed{\text{Bound}=16}$

usw.

5.4.4 Dynamische Programmierung

Der "Lösungsbaum" aus Abbildung 5.3 läßt sich in einen "Lösungsgraphen" (siehe Abbildung 5.4 umwandeln, indem jeweils partielle Rundreisen, die dieselben Städte besucht haben, zusammengelegt werden. Jeder Knoten im "Lösungsgraph" repräsentiert eine partielle Rundreise und definiert:

- Menge der bereits besuchten Städte
- die aktuell erreichte Stadt.

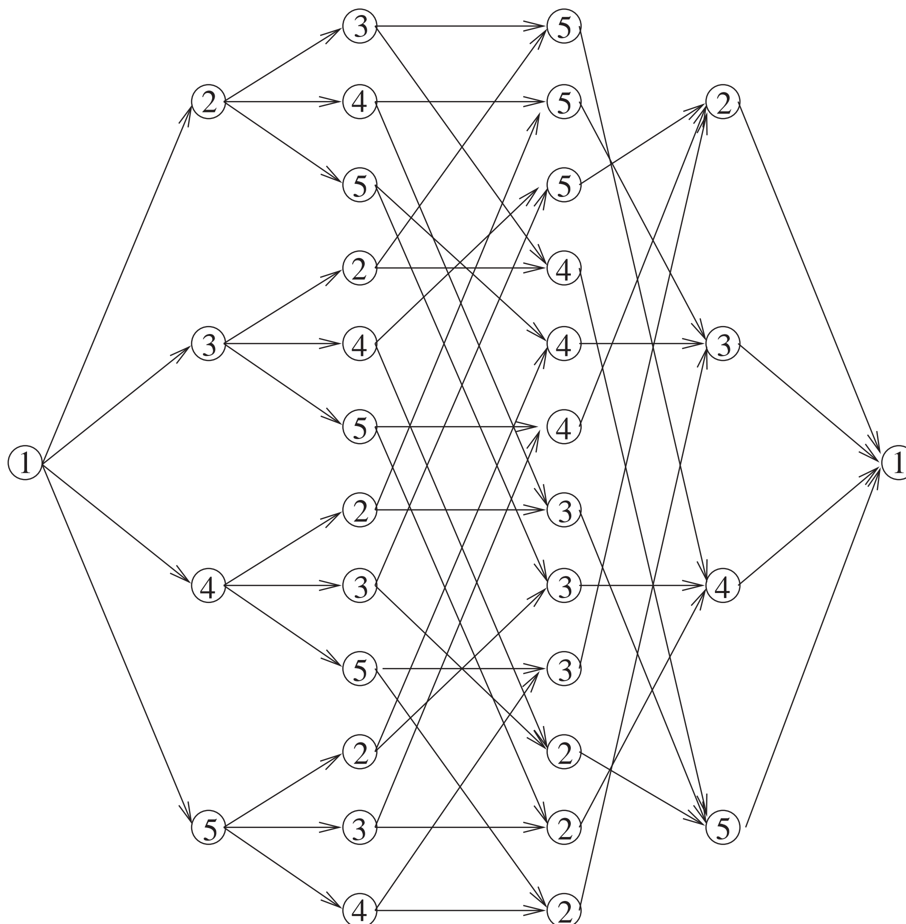


Abbildung 5.4: Lösungsgraph für einen Graphen mit 5 Knoten.

Ansatz: Die Teillösungen werden über die Mengen der bereits besuchten Knoten (Städte) definiert: $k \in S \subset V \setminus \{1\}$:

$D(S, k) :=$ Kosten des optimalen Pfades, der ausgehend von Knoten 1 alle Knoten der Menge S durchläuft und dabei in Knoten $k \in S$ endet.

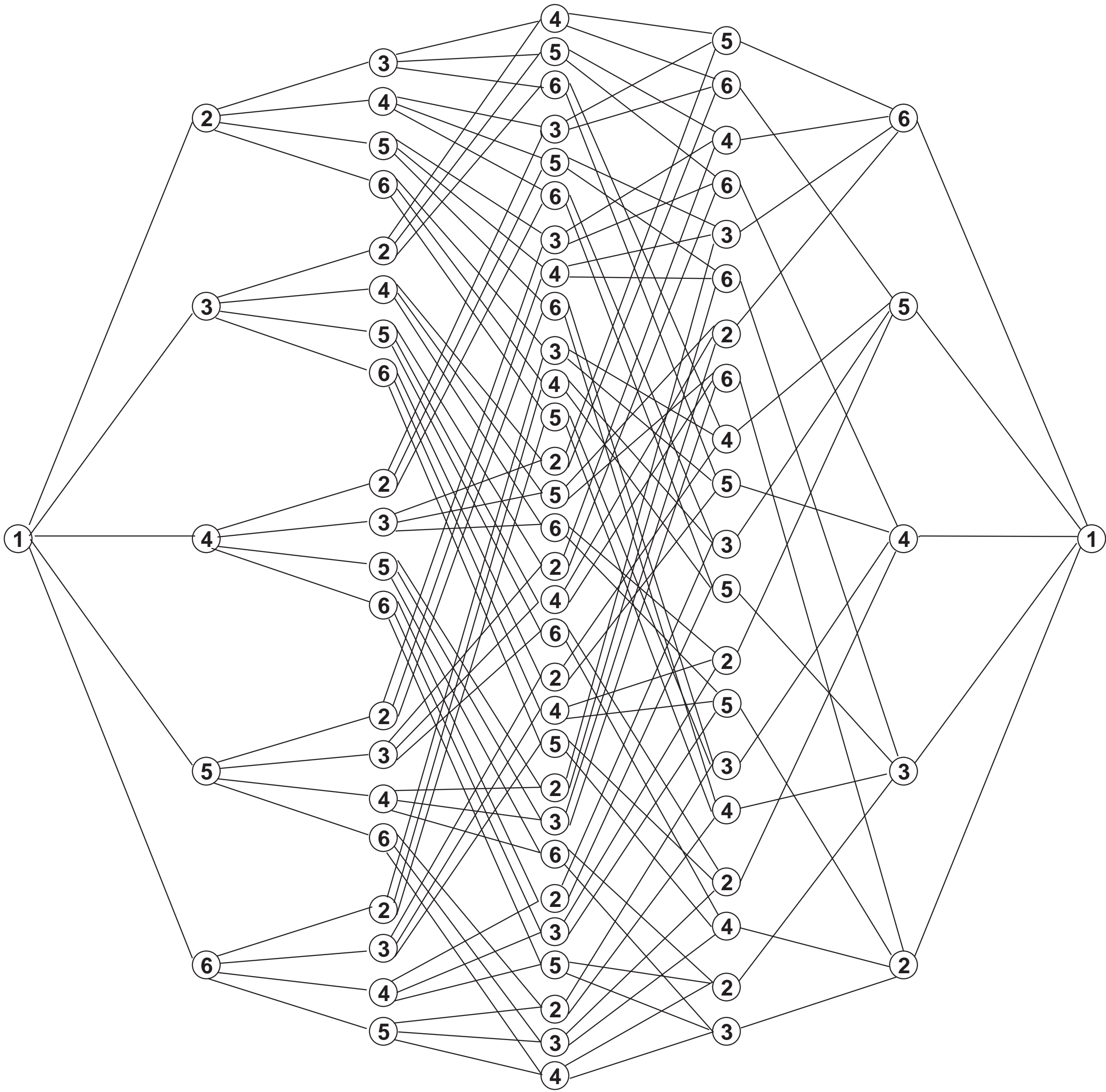


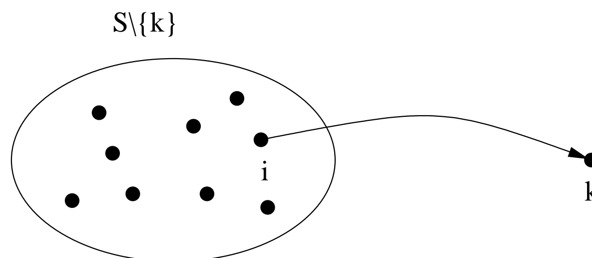
Abbildung 5.5: Lösungsgraph für einen Graphen mit 6 Knoten.

Aus dieser Definition ergibt sich die Rekursionsgleichung:

$$D(S,k) = \min_{i \in S \setminus \{k\}} \{D(S \setminus \{k\}, i) + d_{ik}\}$$

Das Auswerten der Rekursionsgleichung erfolgt, indem man die Teilmengen, genauer ihre Kardinalität, immer größer werden läßt.

Veranschaulichung:



Knotenmenge V mit $n = |V|$
Kostenmatrix d_{ij}
Startknoten $v_0 \in V$, z.B. $v_0 = 1$

(1) **initialization:** $D(\{i\}, i) := d_{v_0,i} \quad \forall i \in V \setminus \{v_0\}$

(2) **for each cardinality** $c = 2, \dots, n-1$ **do**
(3) **for each subset** $S \subset V \setminus \{v_0\}$ **with** $|S| = c$ **do**
(4) **for each vertex** $k \in S$ **do**
(5) $D(S,k) = \min_{i \in S \setminus \{k\}} \{D(S \setminus \{k\}, i) + d_{ik}\}$

(6) **optimized cost:** $D^* = \min_{i \in V \setminus \{v_0\}} \{D(V \setminus \{v_0\}, i) + d_{iv_0}\}$

Komplexität: Anzahl $A(n)$ der Additionen und Vergleiche:

Rekursionsgleichung: $A(n)$ Anzahl der möglichen Teilmengen mit Kardinalität $c = 2, \dots, n-1$ bei $n-1$ Elementen ist gegeben durch den Binomialkoeffizienten:

$$\binom{n-1}{c}$$

Damit folgt:

$$\begin{aligned}
A(n) &= \underbrace{n-1}_{\text{Zeile (6)}} + \underbrace{\sum_{c=2}^{n-1}}_{\text{Zeile (2)}} \underbrace{\binom{n-1}{c}}_{\text{Zeile (3)}} \cdot \underbrace{c}_{\text{Zeile (4)}} \cdot \underbrace{(c-1)}_{\text{Zeile (5)}} \\
&= n-1+\sum_{c=2}^{n-1}(n-1)(n-2)\binom{n-3}{c-2} \\
&= n-1+(n-1)(n-2)\sum_{c=2}^{n-1}\binom{n-3}{c-2} \\
&= \ldots \\
&= n-1+(n-1)\cdot(n-2)\cdot 2^{n-3} \\
&\cong n^2 \cdot 2^{n-3} \text{ für } n \gg 1
\end{aligned}$$

Diese Komplexität ist immer noch exponentiell, bedeutet aber im Vergleich zu $n!$ schon eine erhebliche Verbesserung:

n	$n!$	$A(n)$
5	120	52
10	$3.6 \cdot 10^6$	11529
15	$1.3 \cdot 10^{12}$	$8.6 \cdot 10^5$
20	$2.4 \cdot 10^{18}$	$5.0 \cdot 10^7$

$A(n)$ ist exakt angegeben, *nicht* $n^2 \cdot 2^{n-3}$

5.4.5 A^*

$$\begin{aligned}
A^* \text{ search} \cong\; & \text{priority first search (Dijkstra)} \\
+\; & \text{branch \& bound (estimate of remaining cost)} \\
+\; & \text{dynamic programming}
\end{aligned}$$