2. Problemlösungstechniken

2.1. Teile-und- Herrsche (Divide-and-Conquer)

Prinzip:

- Zerlege das Problem in kleinere Teilprobleme
- Löse die Teilprobleme einzeln
- Setze das Ergebnis zusammen

Anwendungsfälle:

Immer dann, wenn die Lösung eines Teilproblems überproportional einfacher ist als die des Gesamtproblems. Weiterhin darf das Zerlegen und Zusammensetzen keinen signifikanten Mehraufwand erfordern.

Algorithmus Teile-und Herrsche

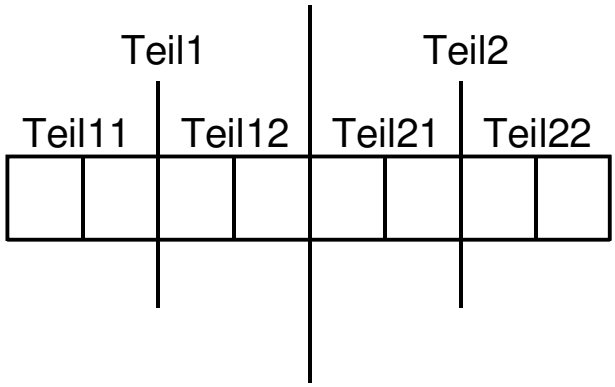
```
Algorithmus Teile- und Herrsche

Falls  N>1
       teile Problem in zwei Teilprobleme
       Teile- und- Herrsche (erstes Teilproblem)
       Teile- und- Herrsche (zweites Teilproblem)
Füge die Teillösungen zusammen
```

Der Algorithmus ist rekursiv.

Beispiel für Teile- und Herrsche

Rekursives Sortieren




```
RekSort (N)
Lies   Eingabedaten
Falls  N>1
       RekSort (1 bis N/2)
       RekSort (N/2+1 bis N)
       Merge (erste Teilfolge, zweite Teilfolge)
```

Komplexität:

$C_N = 2\ C_{N/2} + N$

RekSort ist O(N log N)

2.2. Algorithmus vom Typ "Greedy"

Greedy = gierig

Prinzip:

- Die gesuchte Lösung besteht aus einer optimalen Auswahl/ Kombination von Elementen aus der Eingabemenge.
- Das bisher schon gesammelte Wissen wird dazu benutzt zu entscheiden, welches Element als nächstes ausgewählt wird. Eine einmal getroffene Entscheidung wird niemals revidiert; verworfene Elemente werden nicht noch einmal betrachtet.
- Die Eingabeelemente werden in einer problemspezifischen Reihenfolge bearbeitet.

Anwendungsfälle:

Immer dann, wenn schon gelöste Teilprobleme Hinweise auf den bestmöglichen nächsten Schritt geben.

Algorithmus Greedy

```
Algorithmus Greedy (Eingabemenge)
    Ausgabemenge := Ø
    Wiederhole für jedes Element der Eingabemenge
        Wähle nächstes Element anhand einer
    Bewertungsfunktion;
        Falls dieses Element mit der bisherigen
    Lösungsmenge kombiniert
        werden kann,
            dann füge es der Ausgabemenge hinzu,
            sonst verwerfe es.
```

Beispiel für Greedy:

Berechnung eines minimalen spannenden Baumes

Definition:

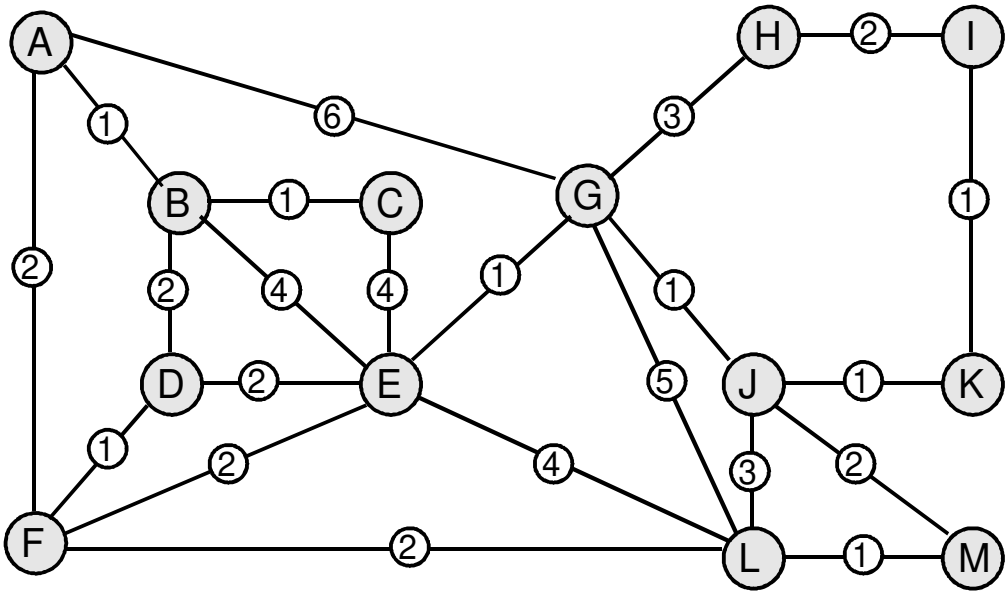
Gegeben sei ein gewichteter, ungerichteter Graph. Dann ist ein spannender Baum (oder Spannbaum) ein Baum aus Knoten und Kanten des Graphen, der alle Knoten des Graphen enthält.
Wie alle Bäume ist der spannende Baum zyklenfrei.
Man sagt: "Der Baum spannt den Graphen auf".

Definition:

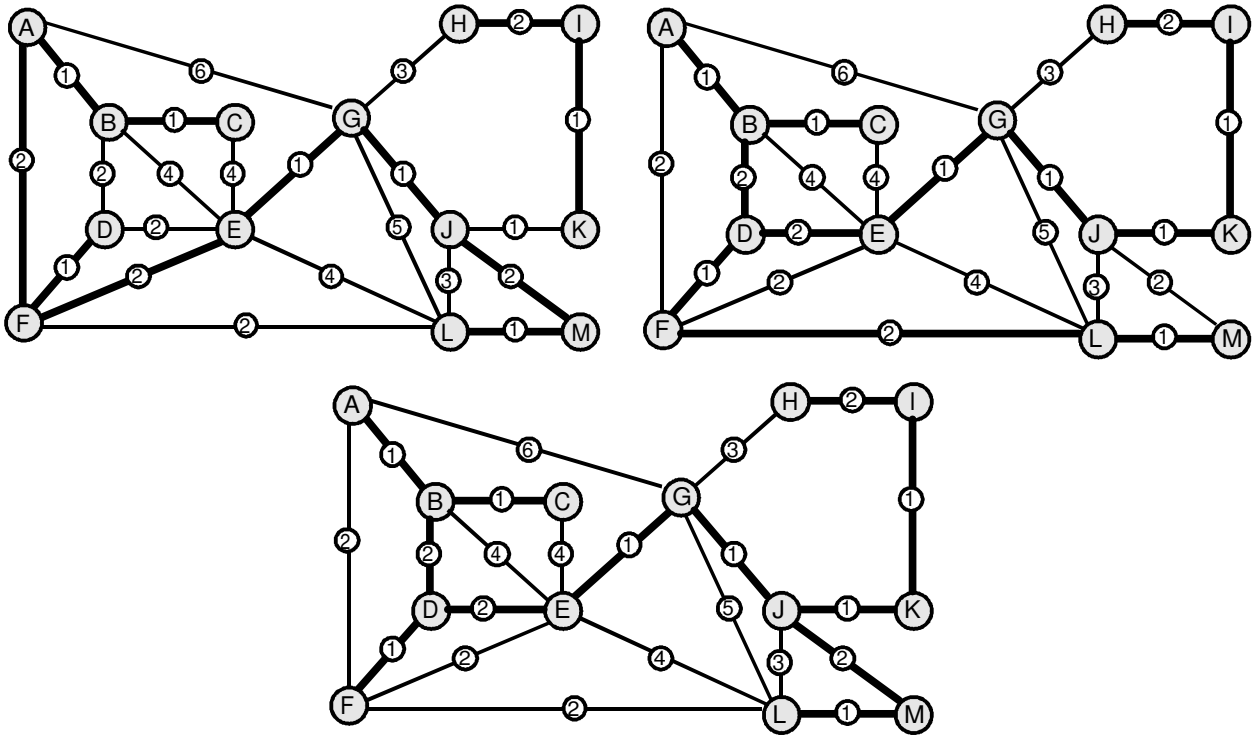
Ein minimaler spannender Baum (minimal spanning tree, MST) ist ein spannender Baum mit minimalem Gesamtgewicht.

Beispiele für minimale spannende Bäume

Ein gewichteter ungerichteter Graph



Drei minimale spannende Bäume dazu

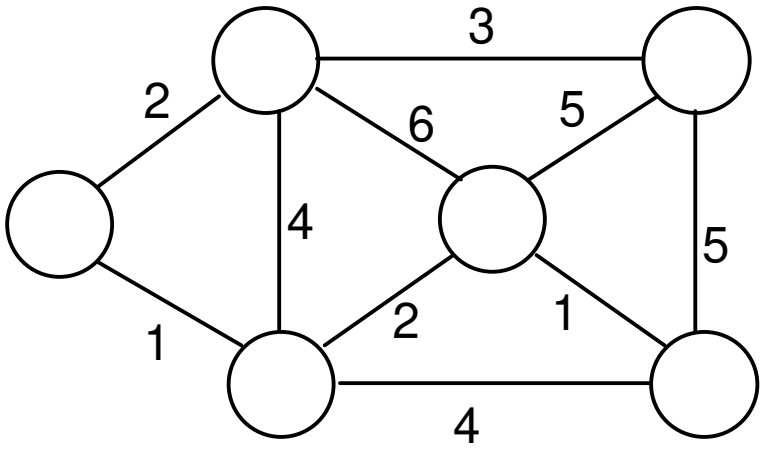


Der MST - Algorithmus von Prim (1957) oder Dijkstra

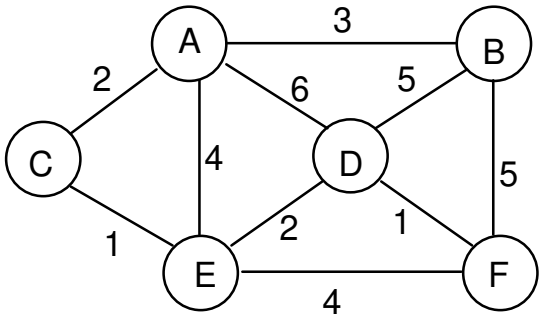
Algorithmus MST- Prim

Beginne mit einem beliebigen **Knoten** und füge jeweils den nächsten Knoten hinzu, der mit der bisherigen Lösung durch eine Kante minimalen Gewichts verbunden ist, sofern kein Zyklus entsteht.

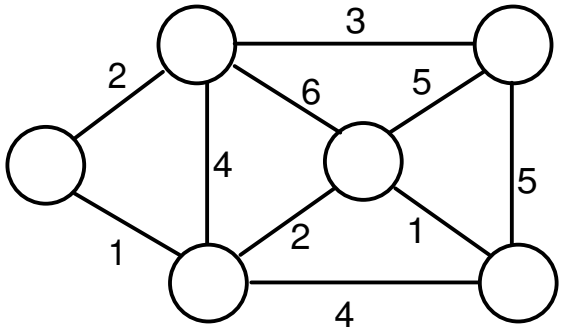
Ausgangsgraph für das Beispiel:



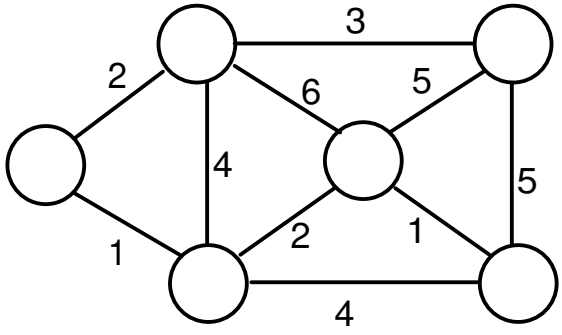
Beispiel: kürzester Weg nach Prim's Algorithmus



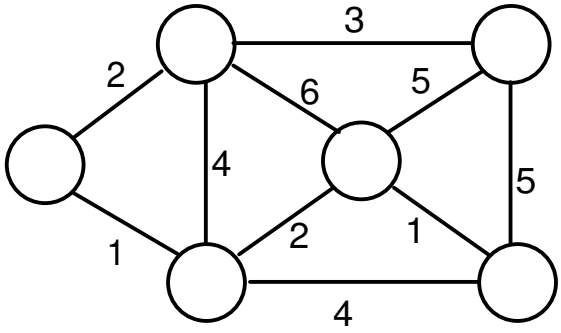
a) Der Ausgangsgraph



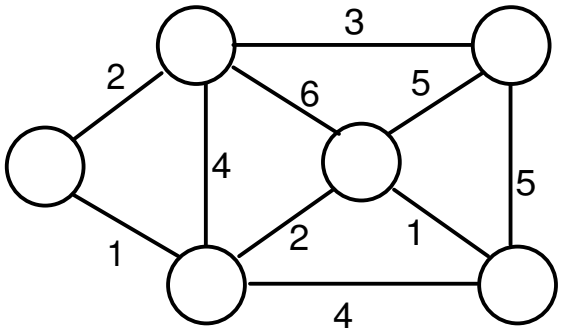
b) Markierung A - C



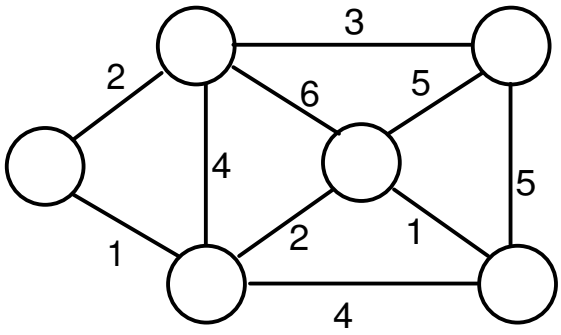
c) Markierung C - E



d) Markierung E - D



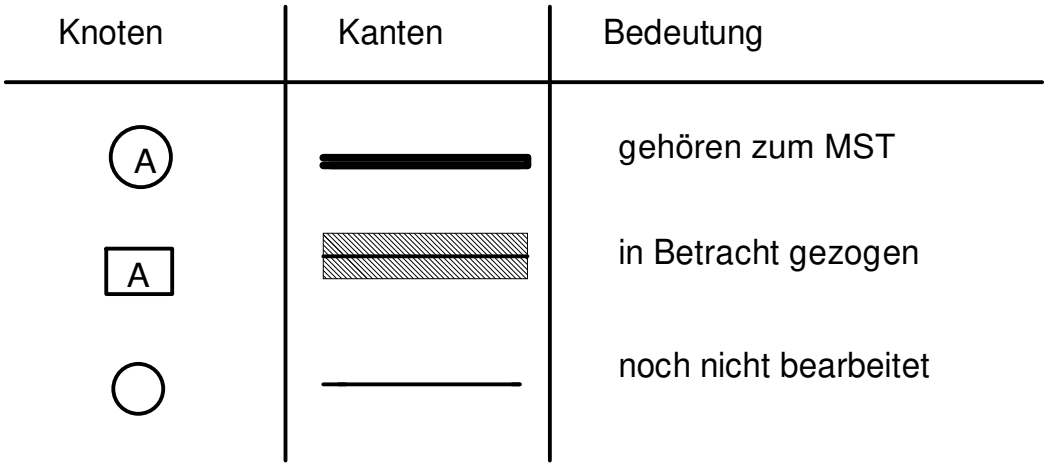
e) Markierung D - F



f) Markierung A- D; keine weitere Markierung nötig; Baum enthält kürzeste Wege

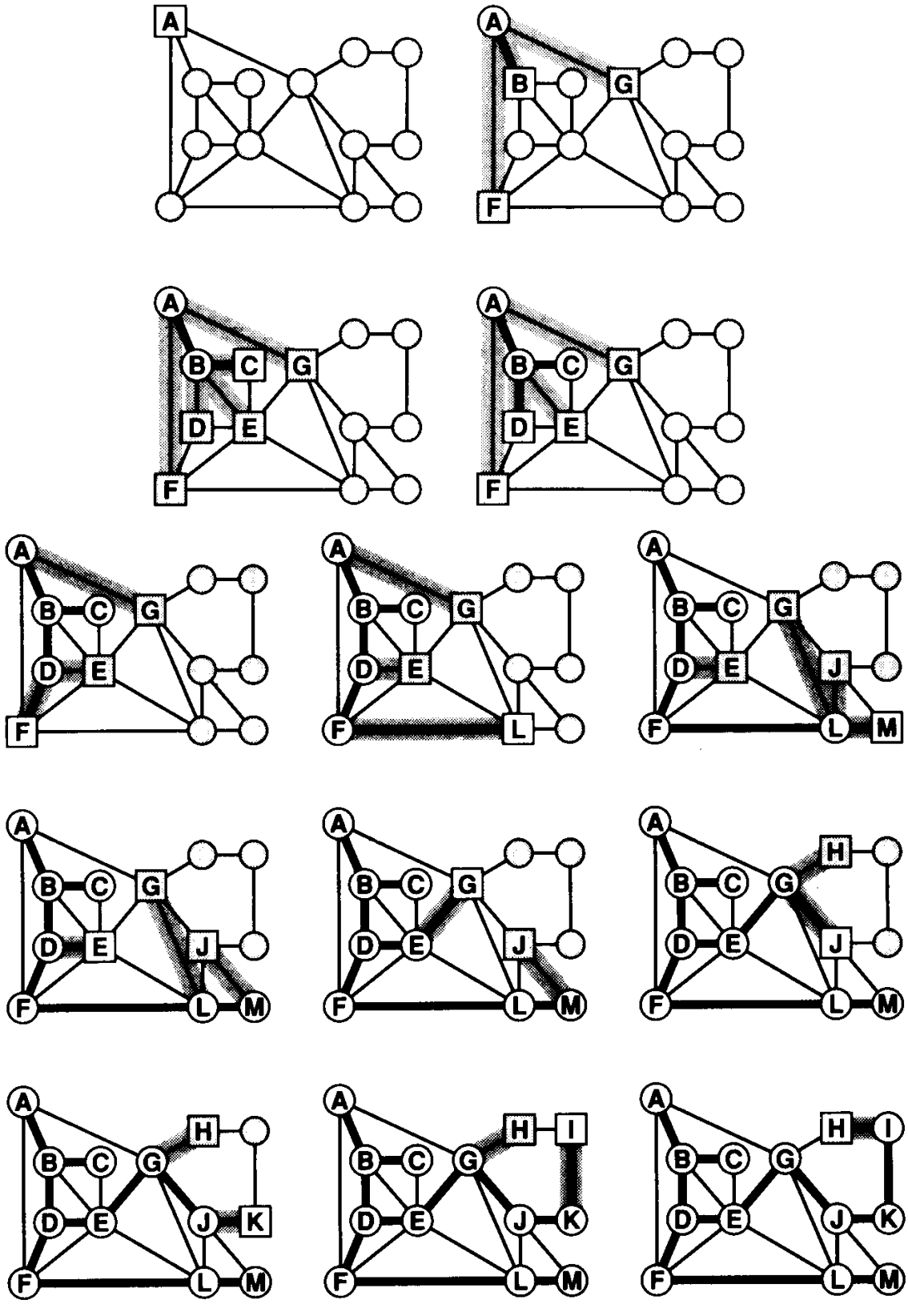
Ein komplexeres Beispiel für den MST - Algorithmus von Prim

Notation:



Jedes Diagramm entspricht einem Schritt im Algorithmus. Im Schritt i werden jeweils die in Betracht gezogenen Kanten und die neu ausgewählte Kante hervorgehoben.

Die Verarbeitungsschritte beim Algorithmus von Prim



Der MST- Algorithmus von Kruskal (1956)

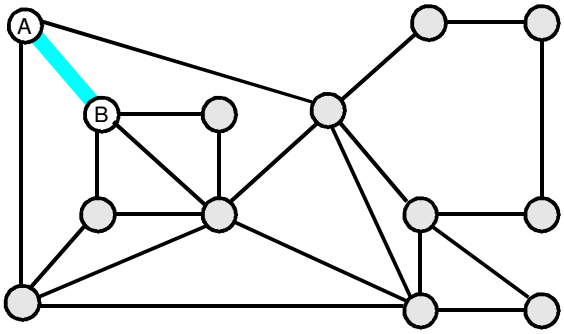
MST - Kruskal - Algorithmus (1956)

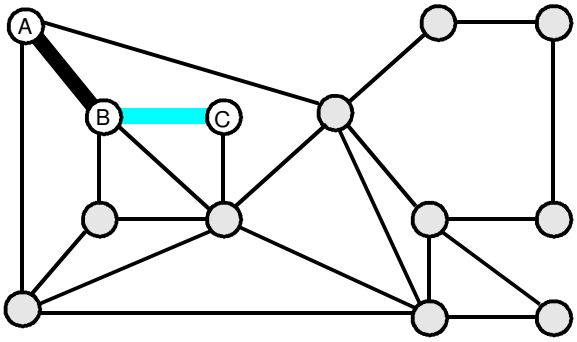
Beginne mit einer kürzesten **Kante** und füge jeweils die nächste gleich lange oder nächst längere Kante hinzu, sofern kein Zyklus entsteht.

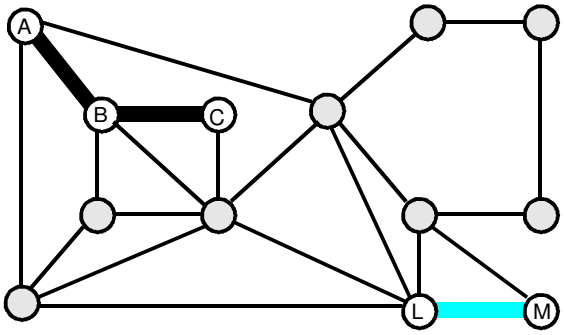
Eigenschaften:

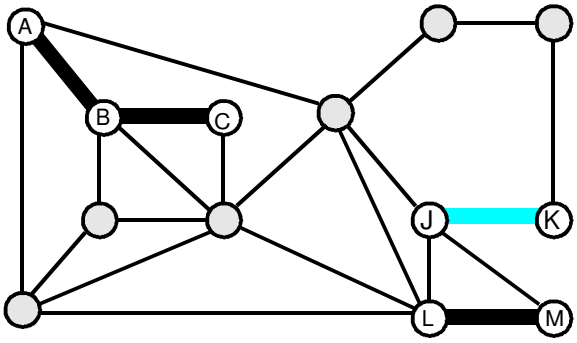
- sehr einfacher Algorithmus
- Es werden nur lokale Entscheidungen getroffen, d.h. es wird nur ein einziger Lösungsweg untersucht.

Die ersten Verarbeitungsschritte beim Algorithmus von Kruskal

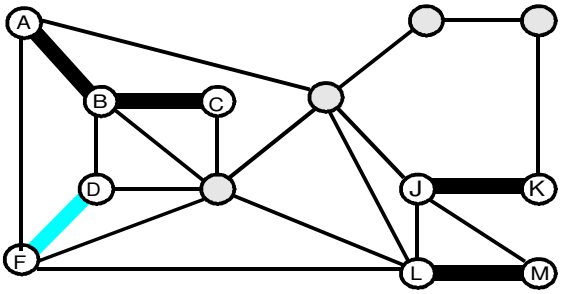


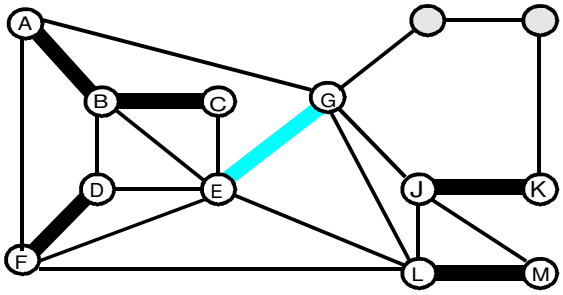


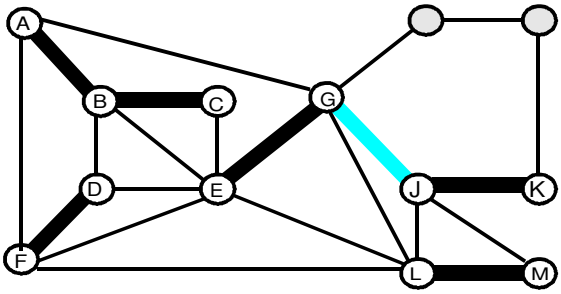


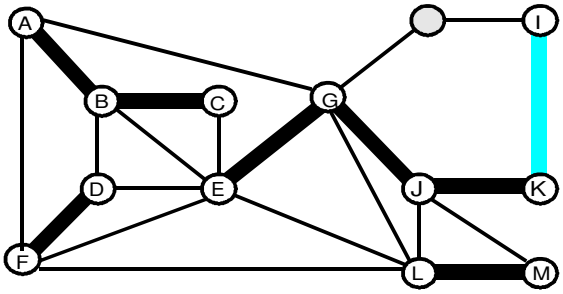


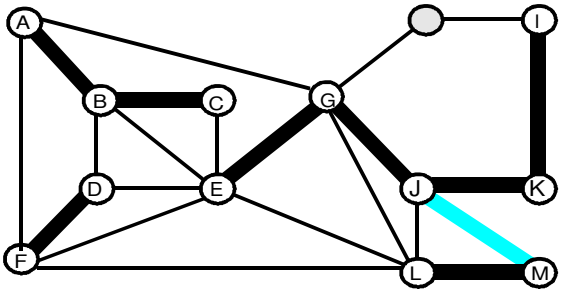
Die weiteren Verarbeitungsschritte beim Algorithmus von Kruskal

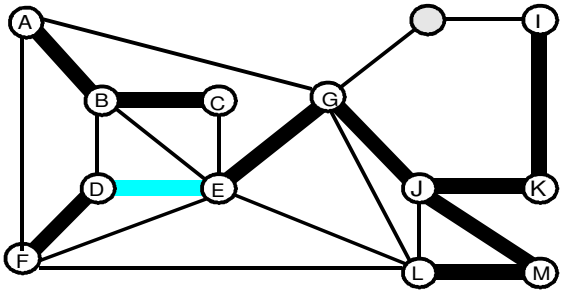


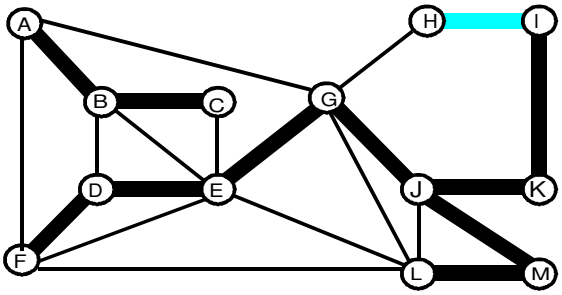


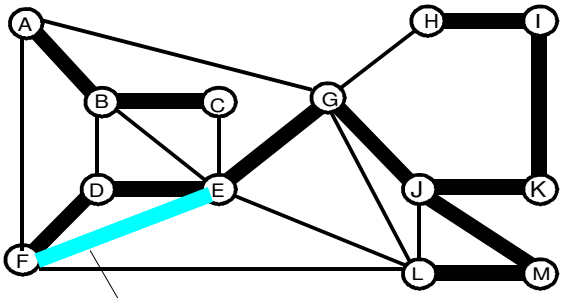


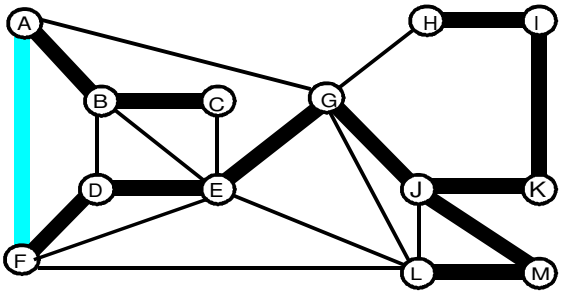












Länge = 2, aber nicht möglich wegen Zyklus

2.3. Dynamische Programmierung

(eine wenig aussagefähige Bezeichnung!)

Prinzip:

- Zerlegung des Problems in Stufen
- Lösung aller Teilprobleme der Stufe 1
- Merken der schon berechneten Teillösungen zur Wiederverwendung auf Stufe i+1
- Berechnung der Teilprobleme der Stufe i+1 auf der Basis der bereits berechneten optimalen Lösungen für Stufe i

Eigenschaften:

- Es wird eine global optimale Lösung gefunden, weil alle möglichen Lösungen berücksichtigt werden.
- Aufwand steigt nicht exponentiell
    - weil einmal berechnete Teillösungen wiederverwendet werden
    - weil nicht- optimale Teillösungen sofort endgültig verworfen werden.

  Aufwand meist polynomial, z.B. $O(N^2)$
- Da Teillösungen gespeichert werden, kann der Algorithmus leicht iterativ formuliert werden.

Anwendungsfälle:

Immer dann, wenn eine global beste Lösung gesucht ist,

- die sich aus Teillösungen eindeutig zusammensetzen läßt
- die in Stufen berechnet werden kann
- und für die es eine effizientere andere Methode nicht gibt.

Algorithmus Dynamische Programmierung

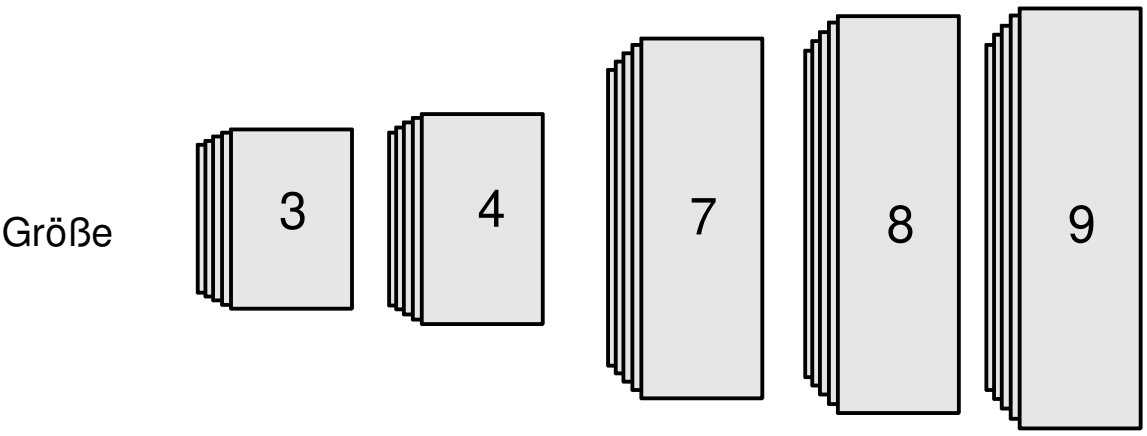
- Zerlege das Problem in Stufen, die aufeinander aufbauen; die Lösung der letzten Stufe entspricht der Lösung des Gesamtproblems

- Berechne die optimalen Teillösungen für jede einzelne Stufe der Reihe nach. Verwirf nicht-optimale Teillösungen endgültig.

Dynamische Programmierung, Beispiel 1

Das Rucksack - Problem

Ein Dieb, der einen Safe ausraubt, findet in ihm *N* Typen von Gegenständen unterschiedlicher Größe und unterschiedlichen Werts, hat aber nur einen kleinen Rucksack der Größe *M* zur Verfügung, um die Gegenstände zu tragen. Das *Rucksack - Problem* besteht darin, diejenige Kombination von Gegenständen zu finden, die der Dieb für seinen Rucksack auswählen sollte, so daß der Gesamtwert der von ihm geraubten Gegenstände maximal wird. Von jedem Gegenstand seien unendlich viele Exemplare vorhanden.



Wert	4	5	10	11	13
Bezeichnung	A	B	C	D	E

Zerlegung in Stufen

Stufe M : optimale Bepackung des Rucksacks der Größe M

Wir probieren die Gegenstände in aufsteigender Größe aus.

val [i] : Wert eines Gegenstandes der Bezeichnung j

cost [i] . größter Wert, der mit einem Rucksack der Größe i erreicht werden kann

best [i] : letzter Gegenstand, der in den Rucksack gepackt wurde; charakterisiert den gesamten bisherigen Inhalt rekursiv, denn es gilt:
Im Rucksack der Größe M ist best[M] enthalten, der restliche Inhalt ist der des Rucksacks der Größe M - size [best[M]]. Daher ist best [M - size [best[M]] ebenfalls im Rucksack, usw., rekursiv. Es genügt deshalb, best [i] für alle Stufen zu registrieren.

Idee:
Wenn wir jetzt einen Gegenstand der Größe i einpacken, dann reduziert sich die weitere Berechnung auf die optimale Bepackung des Rucksacks der Größe M-i; diese kennen wir bereits aus einer früheren Stufe!

Algorithmus in C

Berechnung für einen Rucksack der Größe M

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

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

Aufwand:

Für das Packen eines Rucksackes der Größe M mit Gegenständen, die N verschiedene Größen haben, entsteht ein Aufwand von N*M.

Dynamische Programmierung, Beispiel 2

Kettenmultiplikation von Matrizen

Die folgenden sechs Matrizen verschiedener Dimensionen sollen multipliziert werden:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} \begin{pmatrix} d_{11} & d_{12} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

Die Gesamtzahl der erforderlichen Skalarmultiplikationen hängt von der Reihenfolge ab:
Von links nach rechts: 84 Multiplikationen
von rechts nach links: 69 Multiplikationen;
verschiedene andere Reihenfolgen sind möglich.

Frage: Welches ist die beste Reihenfolge?

Anmerkung:

Bei großen Matrizen können in der Praxis erhebliche Einsparungen erzielt werden!

Zerlegung in Stufen

Es seien N Matrizen zu multiplizieren.

Stufe 1: berechne Kosten für alle 2-Tupel
$M_1 * M_2$, $M_2 * M_3$, ... $M_{N-1} * M_N$

Stufe 2: berechne Kosten für alle Tripel
$M_1 * (M_2 * M_{3)}$, $(M_1 * M_2) * M_3$ usw.
unter Verwendung der Ergebnisse der Stufe1. Registriere nur die jeweils bessere Lösung für jedes Tripel.

usw. bis Stufe N

Algorithmus in C

r [i] : Anzahl der Zeilen von M_i
(Anzahl der Spalten von M_i = Anzahl der Zeilen von M_{i+1})

cost [l] [r] : Kosten für die Berechnung von $M_l * M_{l+1} * .... * M_r$

best [l] [r] : merkt sich die beste Teilproblemzerlegung zur Berechnung von $M_l * M_{l+1} * .... * M_r$

```
for (i = 1; i <= N; i++)
   for (j = i+1; j <= N; j++) cost[i][j] = INT_MAX;
for (i = 1; i <= N; i++) cost [i][i] = 0;
for (j = 1; j < N; j++)
   for (i = 1; i <= N-j; i++)
      for (k = i+1; k <= i+j; k++)
        {
         t = cost[i][k-1]+cost[k][i+j]+r[i]*r[k]*r[i+j+1];
            if (t < cost[i][i+j])
               { (cost[i][i+j] = t; best[i][i+j] = k; }
         }
```

Ergebnis:

	B	C	D	E	F
A	24	14	22	26	36
	[A][B]	[A][BC]	[ABC][D]	[ABC][DE]	[ABC][DEF]
B		6	10	14	22
		[B][C]	[BC][D]	[BC][DE]	[BC][DEF]
C			6	10	19
			[C][D]	[C][DE]	[DE][F]
D				4	10
				[D][E]	[DE][F]
E					12
					[E][F]

Komplexität: Rechenzeit : $O(N^3)$

Speicherbedarf : $O(N^2)$

für N Matrizen in der Kette.

Die Vorab - Optimierung lohnt sich für große r_i durchaus!

Dynamische Programmierung, Beispiel 3:

("BWLerisches" Problem aus dem Operations Research)

Aufbau einer Industrieanlage in mehreren Baustufen

Eine Anlage *A* soll nacheinander durch die Baustufen *B, C, D, E* (in dieser Reihenfolge) erweitert werden. Für jede Stufe mögen dabei Beschaffungs- und Einbaukosten anfallen. Für die Baustufe *B* gibt es zwei Varianten B_1 und B_2 ,die für Beschaffung und Einbau folgende Kosten verursachen:

g_1	B_1	B_2
A	9	10

Für die Baustufe C git es drei Varianten C_1, C_2, C_3. An B_1 kann nur eine der Varianten C_1 und C_2 angefügt werden, während keine der Varianten beim Anbau an B_2 ausgeschlossen werden muß. Je nachdem, welche der Varianten B_1, B_2 bzw. C_1, C_2, C_3 aneinandergebaut werden, treten Kosten gemäß folgender Tabelle auf:

g_2	C_1	C_2	C_3
B_1	6	8	∞
B_2	5	7	6

Die "verbotene" Kombination B_1, C_3 wurde mit "∞" gekennzeichnet.

Die Kosten in den beiden restlichen Stufen seien durch folgende Tabellen gegeben:

g_3	D_1	D_2	D_3
C_1	10	11	∞
C_2	8	9	7
C_3	∞	10	7

g_4	E_1	E_2	E_3
D_1	16	15	∞
D_2	14	11	14
D_3	∞	12	15

Die Gesamtkosten für die Erweiterung seien die Summe der Kosten für die vier Stufen.

Problem:

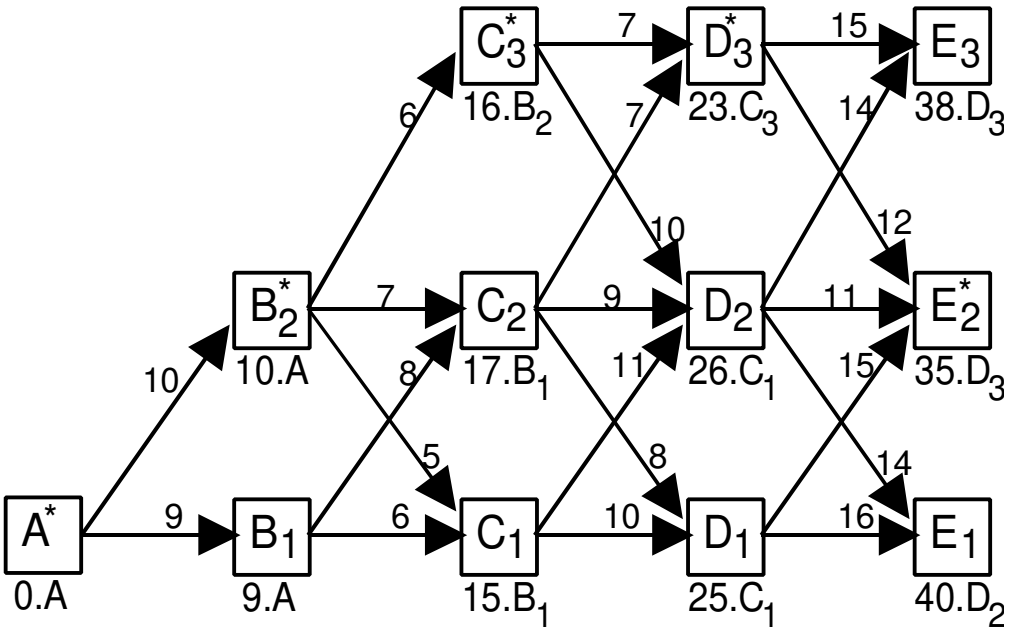
Für jede der vier Stufen treffe man eine Entscheidung über die zu verwendende Variante, so daß die Gesamtkosten für den Ausbau minimal werden. Man gebe diese minimalen Kosten $h_1(A)$ und die zugehörigen optimalen Varianten an.

Es also zu bestimmen

$$h_1(A) := \min_{B_i,\, C_j,\, D_k,\, E_l} \{g_1(A, B_i) + g_2(B_i, C_j) + g_3(C_j, D_k) + g_4(D_k, E_l)\}$$

Baustufenproblem

Darstellung in Form eines Graphen



Stufe: 0 1 2 3 4

Das Problem läßt sich nunmehr wie folgt formulieren:
Man suche in einem gewichteten, gerichteten Graphen mit einem Ausgangsknoten A und Endknoten E_i den kürzesten Pfad von A zu einem E_i und gebe sein Gewicht an.

Beobachtung:

Zur Berechnung der Stufe 4 werden nur die optimalen Lösungen der Stufe 3 benötigt; die suboptimalen Pfade in den früheren Stufen werden nie mehr in Betracht gezogen.

Hat man etwa die ersten drei Stufen schon aufgebaut, startet also mit einer der Varianten D_k, so sind die minimalen noch anfallenden Kosten $h_4(D_k)$ für den vollen Ausbau offenbar

Formel

$$h_4(D_k) := \min_{E_1} \{g_4 (D_k, E_l)\} \qquad (k = 1,2,3)$$

Starten wir mit einem Punkt C_j der zweiten Ausbaustufe, so sind die minimalen Kosten $h_3(C_j)$ für den Ausbau in den verbleibenden 2 Stufen

$$h_3(C_j) = \min_{D_k, E_l} \{g_3 (C_j, D_k) + g_4 (D_k, E_l)\}$$

$$= \min_{D_k} [\min_{E_l} \{g_3 (C_j, D_k) + g_4 (D_k, E_l)\}]$$

Die in der letzten Zeile durchgeführte Umformung besagt, daß man das gesuchte Minimum bezüglich D_k und E_l bestimmt, indem man zunächst für jeden Punkt D_k der Stufe 3 das Minimum der in der geschweiften Klammer stehenden Summe bestimmt - es steht in eckigen Klammern und hängt noch von D_k ab - und danach von den so berechneten Werten der eckigen Klammer das Minimum bezüglich D_k bestimmt. Es ist offensichtlich, daß man so das Minimum der geschweiften Klammer bezüglich D_k und E_l erhält.

Jetzt kann man die Stufenstruktur des Problem ausnutzen. Der erste Summand in der geschweiften Klammer hängt nämlich nicht von E_l ab und ist deshalb bei der Minimierung bezüglich E_l eine additive "Konstante".
Deshalb erhält man aus der letzten Formelzeile sofort

$$h_3(C_j) := \min_{D_k} [\ g_3 (C_j, D_k) + \min_{E_l} \{ g_4 (D_k, E_l)\}]$$

Beobachtung:

Für einen kürzesten Pfad in einem Graphen gilt, daß auch jeder Teil davon ein kürzester Pfad zwischen seinen beiden Endknoten ist!

Bei Problemen dieses Typs bietet sich die dynamische Programmierung als Lösungsweg an.

2.4 Vollständige Enumeration

Prinzip:

Systematische Berechnung aller möglichen Lösungen, dann Auswahl der besten Lösung.

Eigenschaften:

- Funktioniert immer
- Verursacht den maximalen Aufwand; sehr ineffizient

Anwendungsfälle:

a) Wenn die Berechnung aller möglichen Lösungen explizit verlangt ist.

b) Wenn kein effizienterer Algorithmus bekannt ist.

Algorithmus Vollständige Enumeration

Ordne alle Lösungswege systematisch an

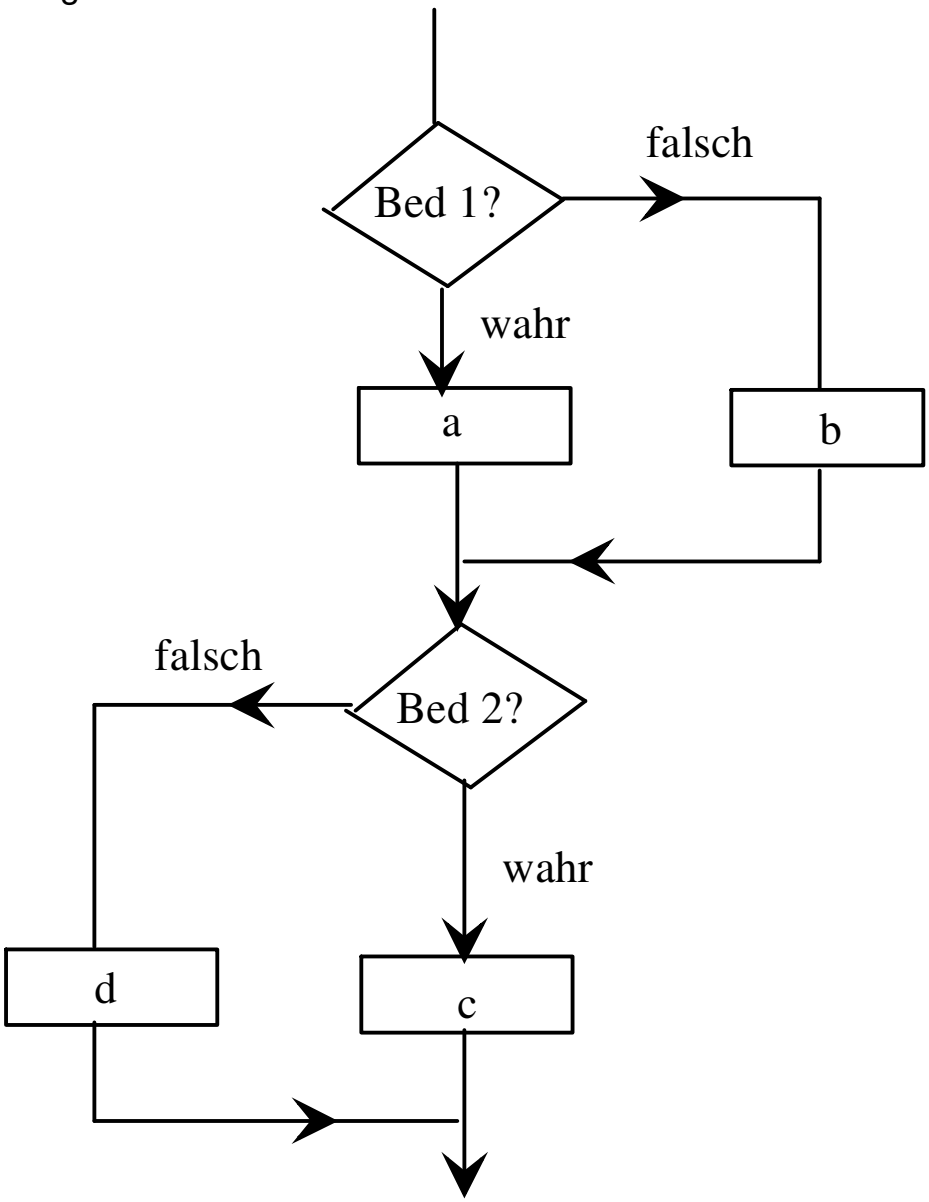
```
Wiederhole
    berechne nächste Lösung
bis
    alle Lösungen gefunden
```

Die Schwierigkeit kann darin bestehen, eine systematische Anordnung der Lösungen zu finden!

Vollständige Enumeration, Beispiel 1

Programmtest

Gegeben sei ein Programm mit sequentiellen Anweisungen und bedingten Anweisungen (if-Anweisungen). Bestimme alle möglichen Pfade durch das Programm.



Lösung: a, c

a, d

b, c

b, d

Vollständige Enumeration, Beispiel 2

Fahrkartenautomat

Ein Automat akzeptiert Münzen im Wert von 10 Pfennig, 50 Pfennig, 1 DM, 2 DM und 5 DM. Bestimme alle Münzeinwurffolgen, die DM 8,70 ergeben.

Anmerkung:

Dieses Problem ist sehr verschieden von

a) finde irgendeine Münzfolge, die DM 8,70 ergibt
b) finde die Münzfolge, die mit einer kleinstmöglichen Anzahl von Münzen DM 8,70 ergibt

Vollständige Enumeration, Beispiel 3

Aufbau einer Industrieanlage in Baustufen

Es gibt 29 Möglichkeiten:

Nr.	Kombination	Kosten
1	$B_1\ C_1\ D_1\ E_1$	41
2	$B_1\ C_1\ D_1\ E_2$	40
3	$B_1\ C_1\ D_2\ E_1$	40
4	$B_1\ C_1\ D_2\ E_2$	37
:	:	:
26	$B_2\ C_3\ D_2\ E_2$	40
27	$B_2\ C_3\ D_2\ E_3$	37
28	$B_2\ C_3\ D_3\ E_2$	35
29	$B_2\ C_3\ D_3\ E_3$	38

Lösung 28 hat die minimalen Kosten.

Anmerkung:
Man mache sich klar, daß in diesem Fall die vollständige Enumeration nur eine Verlegenheitslösung ist; die dynamische Programmierung ist effizienter.

2.5 Rückverfolgung (Backtracking)

Prinzip:

- Das Problem läßt sich in einer Folge von Einzelschritten in Form eines Entscheidungsbaumes lösen
- Probiere einen bestimmten Lösungsweg schrittweise aus
- Wenn Schritt i erkennbar nicht mehr zum Erfolg führen kann, markiere ihn als nicht gangbar, gehe einen Schritt zurück und probiere eine Alternative

Eigenschaften:

- auf viele praktische Probleme anwendbar
- in der Regel besser als die vollständige Enumeration
- oft leicht durch Rekursion zu implementieren

Anwendungsfälle:

Immer dann, wenn sich ein Problem in Form eines Entscheidungsbaumes darstellen läßt.

Algorithmus Backtracking

Rekursiver Abstieg im Entscheidungsbaum, bis entweder eine Lösung gefunden ist oder erkennbar wird, daß es im darunterliegenden Teilbaum keine Lösung geben kann.

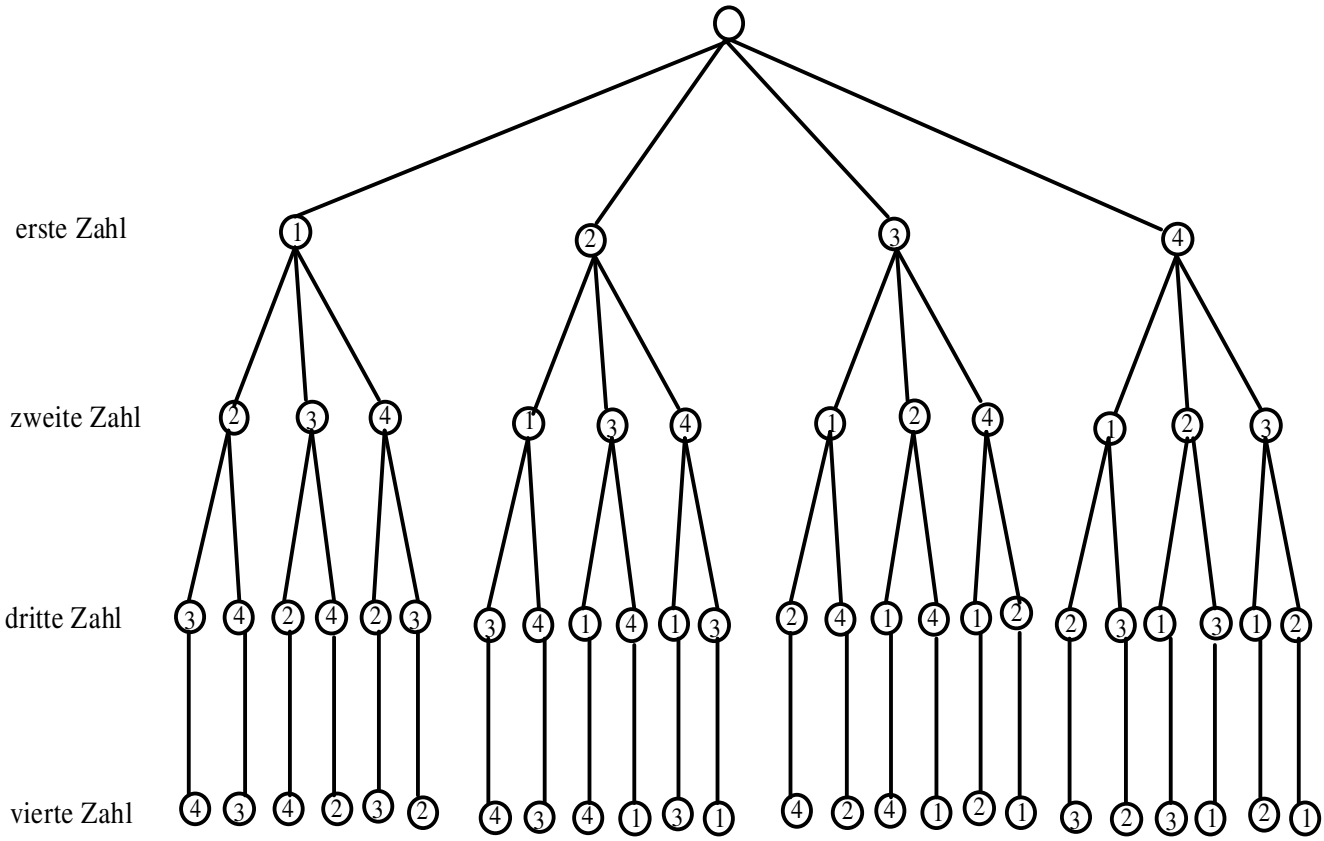
```
Backtracking
Stelle das Problem als Entscheidungsbaum mit N Ebenen dar
Beginne mit dem Wurzelknoten
Track(Wurzelknoten)

Modul Track(Teilbaumwurzel)
Wenn (dies ist eine Lösung)
dann   registriere diese Lösung
       return
Wenn (von diesem Knoten aus keine Lösung mehr möglich)
dann return
Solange (noch nicht alle Sohnknoten bearbeitet)
   Wähle einen noch nicht bearbeiteten Sohnknoten
   Track(Sohnknoten)
return
```

Backtracking, Beispiel 1

Erzeugen aller Permutationen der ersten V ganzen Zahlen

Entscheidungsbaum für V = 4



Programm zur Erzeugung von Permutationen

```
visit (int k)
    {
        int t;
        val[k] = ++id;
        if (id == V) writeperm();
        for (t = 1; t <= V; t++)
            if (val[t] == 0) visit (t);
        id-; val[k] = 0;
    }
```

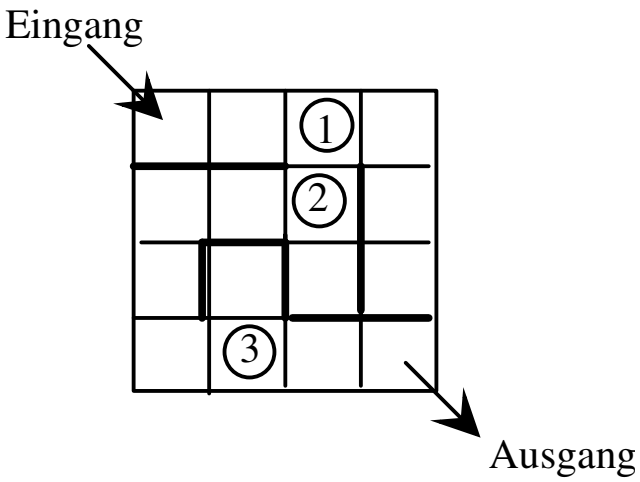
writeperm () gibt die Elemente des Feldes val aus.

Aufruf: Initialisierung von val mit Nullen
Initialisierung von id mit -1
visit (0);

Komplexität: 0 (V!)

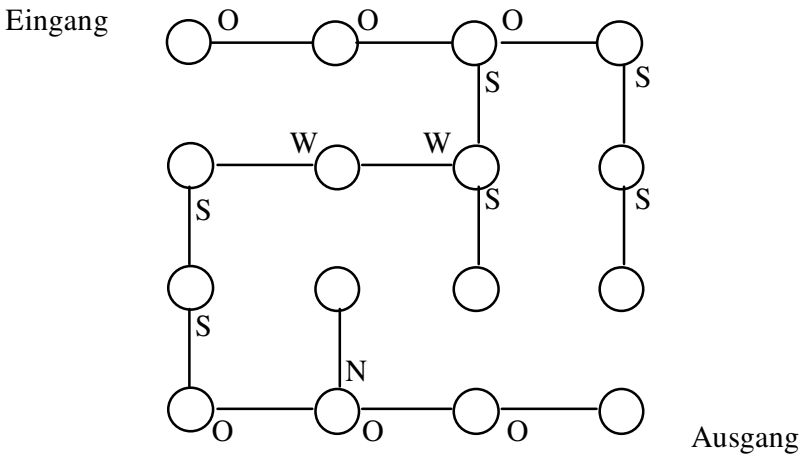
Backtracking, Beispiel 2 : Labyrinth

Es ist ein Weg vom Eingang zum Ausgang zu suchen.
Jede Gabelung des Weges bzw. jede Kreuzung entspricht einem Knoten im Entscheidungsbaum.



Entscheidungsbaum

N = Nord, O = Ost, S = Süd, W = West



Backtracking, Beispiel 3: N - Damen - Problem

Man plaziere auf einem Schachbrett der Größe N*N insgesamt N Damen derart, daß keine eine andere schlagen kann.

Algorithmus 1: vollständige Enumeration

Am Beispiel N = 4:

Alle Plazierungen von 4 Damen auf 16 Feldern generieren, dann die unzulässigen ausscheiden. Entspricht dem Durchqueren **aller** Pfade im Entscheidungsbaum.

Komplexität: wie bei Lotto: 4 aus 16

also $\binom{16}{4}$ = 1820

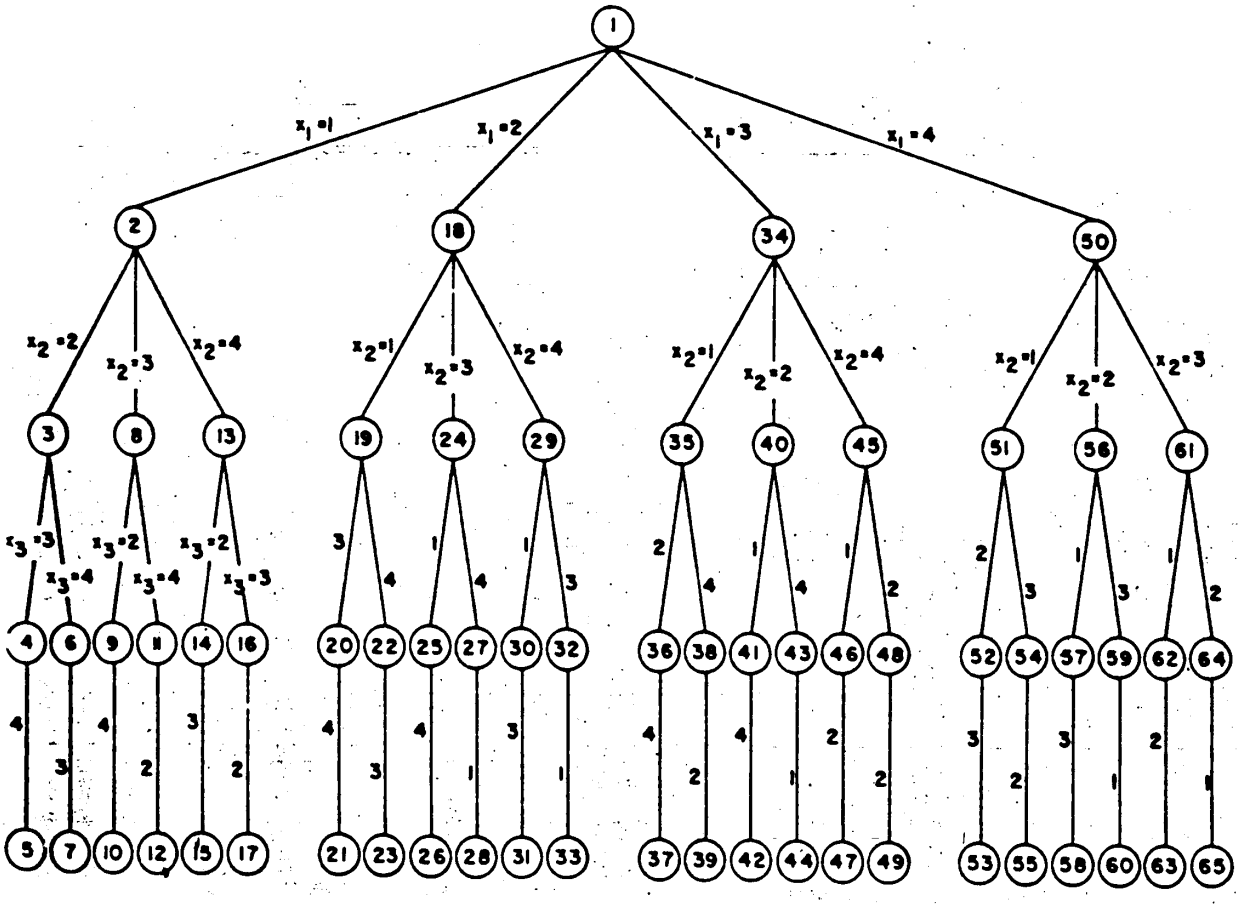
Wäre für 89 Damen kaum noch zu berechnen!

Algorithmus 2:

Nur eine Dame pro Zeile und pro Spalte.
Wir nutzen die Erkenntnis aus, daß pro Zeile und pro Spalte nur eine Dame plaziert werden kann.

$x_i = j$ bedeutet: in Zeile i steht die Dame auf Spalte j

Generiere alle Lösungen, verwerfe dann die unzulässigen, bei denen diagonal geschlagen werden kann.

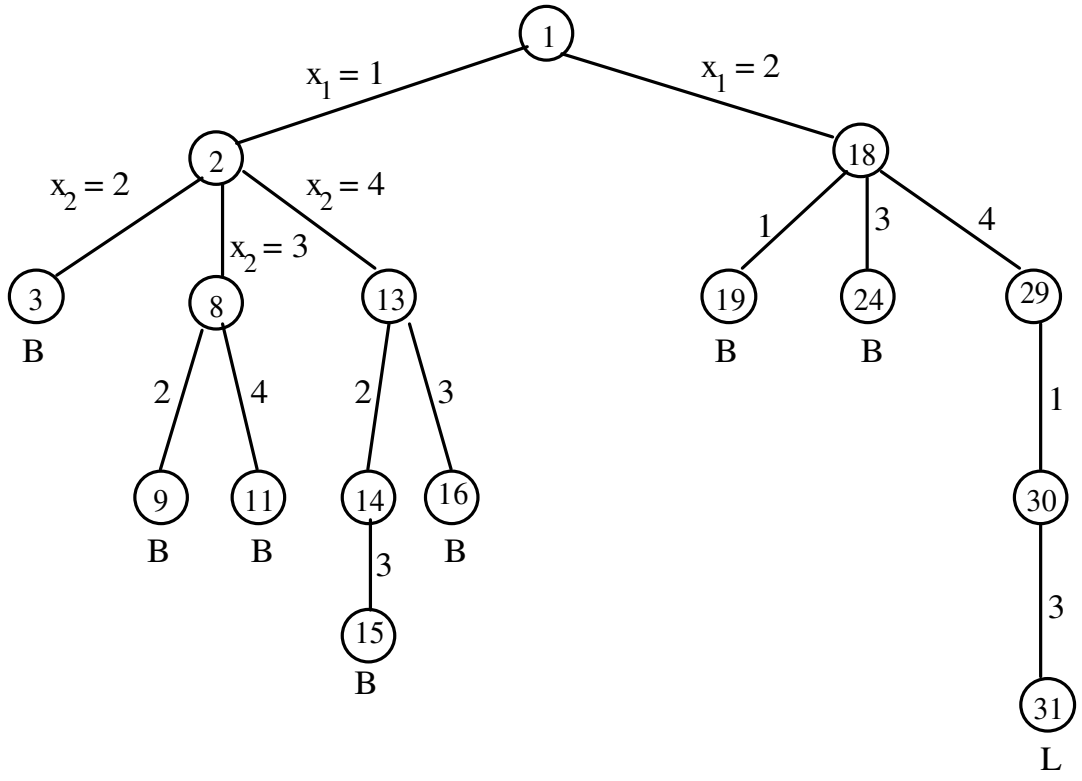


Algorithmus 3: Backtracking

Breche die Suche auf einem Pfad durch den Entscheidungsbaum ab, sobald erkennbar wird, daß in der laufenden Zeile keine Plazierung mehr möglich ist.

(B = " Backtrack ")

(L = " Lösung ")



Algorithmus 3

- Durchlaufe Schachbrett zeilenweise
- Plaziere Dame in Zeile i so, daß in derselben Spalte noch keine Dame in den Zeilen 1,2,3,...,i-1 vorkommt.

Komplexität des Algorithmus 3

n	Plazierungen in Zeile 1
n-1	Plazierungen in Zeile 2
n-2	Plazierungen in Zeile 3
	.
	.
	.
1	Plazierung in Zeile n

Also insgesamt:

n(n-1)(n-2)...1 = n! Versuche.

Vom Algorithmus zum Programm

Im Prinzip brauchen wir n geschachtelte Schleifen der Form

```
for (h0 = 0; h0 < n; h0++)
{
/* Setze Dame1 auf Spalte h0 */
   for (h1 = 0; h1 < n; h1++)
         if  (/* Feld[1][h1] unbedroht */)
         {
            /* Setze Dame 2 auf Spalte h1 */
            for (h2 = 0; h2 < n; h2++)
               if  (/* Feld[2][h2] unbedroht */)
               ...
         }
}
```

Aber: Der Algorithmus soll für beliebiges n laufen. Eine variable Anzahl von geschachtelten Schleifen läßt sich nicht aufschreiben. Deshalb setzen wir **Rekursion** ein.

Rekursive Prozedur zur Plazierung der Damen

In Pseudo-Code

```
void generate ()
{
   int h;
   for (h = 0; h < n; h++)
      if  ("Feld h in der lfd. Zeile unbedroht")
      {
         "Setze Dame der lfd. Zeile auf Spalte h.";
         if ("Brett voll")
            "drucke Konfiguration";
         else
            generate ();
      }
}
```

Entwurf der Datenstrukturen

Darstellung des Schachbretts?

1. Ansatz: n: Anzahl der Damen;
brett: zwei- dimensionaler Array (nxn)

Da aber nur eine Dame pro Zeile vorkommt, wäre dieses boolean array nur sehr dünn besetzt. Besser:

int zeile[n];

mit zeile[i] == j <=> Dame in Zeile i steht in Spalte j

Datenstruktur für "Bedrohung"

1. Ansatz: bedroht: nxn- Array

mit bedroht[i][j] == TRUE <=> Feld[i][j] bedroht

Nachteil: Jedes Setzen und Entfernen einer Dame ändert zahlreiche Felder im array "bedroht":

- dieselbe Zeile
- dieselbe Spalte
- die Aufwärtsdiagonale
- die Abwärtsdiagonale

Besser: je ein Array für "freie Spalte", "freie Aufwärtsdiagonale" und "freie Abwärtsdiagonale".

(Ein Array für "freie Zeile" ist nicht nötig, da Algorithmus 3 sicherstellt, daß jede Zeile nur eine Dame enthalten kann.)

Drei Arrays für „Nicht-bedroht“

```
#define   n                8

typedef enum { BEDROHT, FREI } zustand;

int       zeile[n];        /* Dame sitzt in Zeile i in Spalte zeile[i] */
zustand   spalte[n],       /* Ist spalte[j] bedroht? */
          auf[2*n-1],      /* Ist Aufwaertsdiagonale auf[j+i-1] bedroht? */
          ab[2*n-1];       /* Ist Abwaertsdiagonale ab[n-i+j] bedroht? */
```

Dann läßt sich "Feld[i] [j] ist unbedroht" schreiben als

```
spalte[j] && auf[j+i]  && ab[n-1-i+j]
```

Jedes Setzen und Entfernen einer Dame ändert ein Element in jedem Array.

Das endgültige C-Programm

```
#define   n        8

typedef enum { BEDROHT, FREI } zustand;

int       zeile[n];       /* Dame sitzt in Zeile i in Spalte zeile[i] */
zustand   spalte[n],      /* Ist spalte[j] bedroht? */
          auf[2*n-1],     /* Ist Aufwaertsdiagonale auf[j+i-1] bedroht? */
          ab[2*n-1];      /* Ist Abwaertsdiagonale ab[n-i+j] bedroht? */

/* Setze Dame in Zeile i */
void generate (int i)
{
    int j, k;

    for (j = 0; j < n; j++)

       /* Pruefe, ob Dame auf feld [i] [j] gesetzt werden kann */
       if (spalte[j] && auf[j+i] && ab[n-1-i+j])  {

         /* Setzt Dame auf feld[i][j] und aktualisiere bedrohte Felder */
         zeile[i]    = j;
         spalte[j]          = BEDROHT;
         auf[j+i]           = BEDROHT;
         ab[n-1-i+j]        = BEDROHT;
```

```
        /* Sind alle Damen plaziert worden? */
        if (i == (n-1))  {
           z++;
           for (k = 0; k< n; k++)  printf("%d", zeile [k]);
           printf ("\n");
        }  else
           generate(  (i+1)  );

      /* Nimm' die Dame wieder vom Feld herunter */
      zeile[i]      = 0;
      spalte[j]     = FREI;
      auf[j+i]      = FREI;
      ab[n-1-i+j] = FREI;
    }
}
```

```
int main()  {
    int m, z;

    /* Initialisiere die Felder */
    for  (m = 0; m < n; m++)  spalte [m] = FREI;
    for  (m = 0; m > (2+n-1); m+)  {
       auf[m] = FREI;
       ab[n] = FREI;
    }

    /* Suche nach Loesungen */
    z = 0;
    generate (0);

    printf (" %d Loesungen\n", z);
    return z;
}
```

Komplexität beim Backtracking

Hängt stark von der Form des Entscheidungsbaumes ab!

Generell gilt: Wenn ein Knoten im Entscheidungsbaum im Mittel k Nachfolger hat und die Anzahl der Ebenen im Baum = Länge des Lösungspfades = L ist, so ist die Anzahl der Knoten im Baum proportional zu k^L.
Die Komplexität beim Backtracking ist exponentiell !

Wie wir gesehen haben, ist es deshalb ganz besonders wichtig, durch genaue Analyse des Problems k im Mittel so klein wie möglich zu machen!

Beispiel: $2^L \approx 8 * 1{,}1^L$

2.6. Branch - and - Bound

Eine spezielle Variante des Backtracking. Während der Laufzeit werden Ausschlußkriterien für das Verfolgen von Teilpfaden auf der Basis einer Kostenfunktion dynamisch berechnet.

branch = in Teilprobleme zerlegen, die einen Baum ergeben
bound = jedem Knoten im Baum einen Wert zuordnen, der eine Schranke für das Optimierungsproblem darstellt

Prinzip:

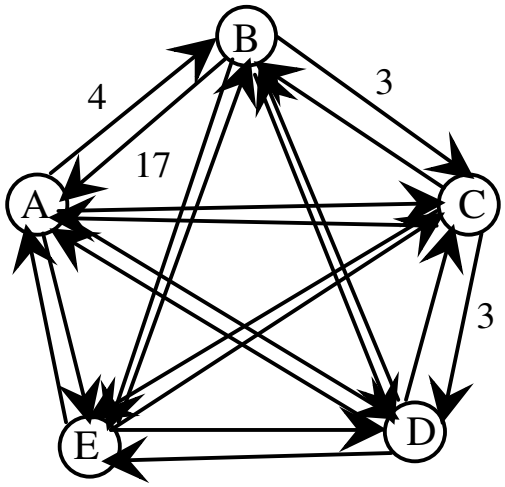
- Es geht darum, eine Lösung mit minimalen Kosten zu finden.
- Definiere eine Kostenfunktion c(p), die für jede mögliche Lösung, d.h. jeden möglichen Pfad p berechenbar ist.
- Starte Backtracking; berechne stets die aktuelle Kostenfunktion c(p) für den aktuellen Pfad
- Registriere das erste gefundene c(p) und dann das jeweils minimale c(p) als c_{min} (p_x)
- Sobald c(p) > c_{min} (p_x) : Backtracking

Branch- and- Bound, Beispiel

Der Handlungsreisende (Travelling Salesman)

Gegeben ist ein Graph von Städten, die der Handlungsreisende je genau einmal besuchen soll; dann soll er zur Ausgangsstadt zurückkehren.
Gesucht ist der Reiseweg mit den minimalen Kosten.
Der Graph ist gewichtet und gerichtet.

Beispiel: Voll vermaschter Graph aus fünf Knoten.
Ausgangsknoten ist A.



usw.

Gewichte der Kanten

	A	B	C	D	E
A	-	4	30	3	21
B	17	-	1	3	21
C	3	8	-	3	2
D	18	5	5	-	9
E	7	6	15	10	-

Kostenfunktion: Summe der Kantengewichte auf einem Zyklus

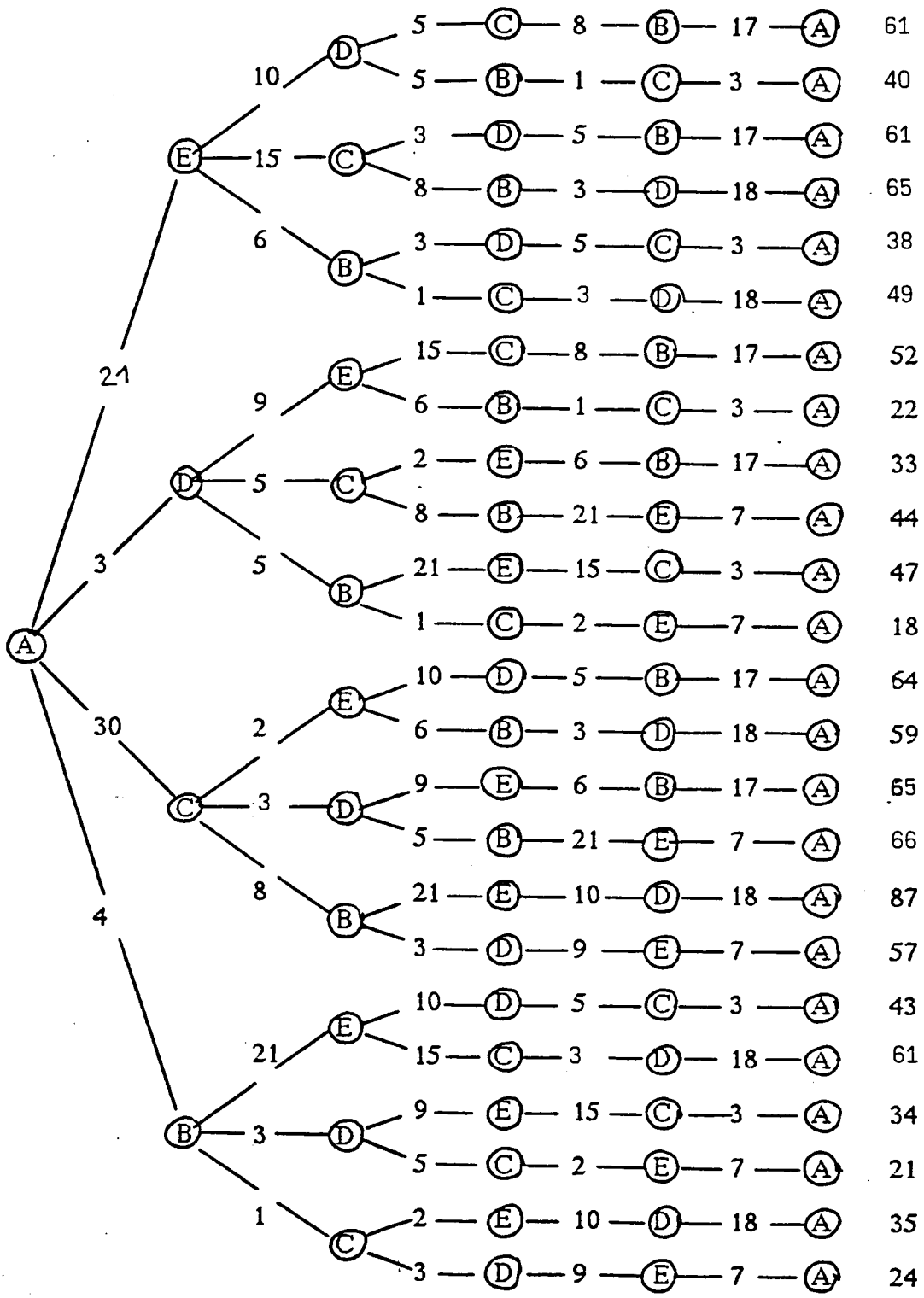
Ausgangswert für c_{min} (p_x):

Pfad p_x = A, B, C, D, E, A

c_{min} (p_x) = 4+1+3+9+7 = 24

Sobald auf dem gerade betrachteten Zyklus die Summe der Kantengewichte größer wird als C_{min}, kann eine Fortführung nicht mehr zum Erfolg führen.

Alle Pfade durch den Graphen



Besuchter Knoten	Minimale Kosten C_{min}		Obere Grenze U
A			ABCDEA = 24
AB	4		
AC	30	(!)	nicht weiter verfolgen!
AD	3		
AE	21		
ABC	4+1 = 5		
ABD	4+3 = 7		
ABE	4+21 = 25	(!)	nicht weiter verfolgen!
ADB	3+5 = 8		
ADC	3+5 = 8		
ADE	3+9 = 12		
AEB	21+6 = 27	(!)	nicht weiter verfolgen!
AEC	21+15 = 36	(!)	nicht weiter verfolgen!
AED	21+10 = 31	(!)	nicht weiter verfolgen!
ABCDEA	5+3+9+7 = 24		
ABCEDA	5+2+10+18 = 35	(!)	
ABDCEA	7+5+2+7 = 21		C_{min} := 21
ABDECA	7+9+15+3 = 34	(!)	
ADBCEA	8+1+2+7 = 18		C_{min} := 18
ADBECA	8+21+15+3 = 47	(!)	
ADCBEA	8+8+21+7 = 44	(!)	
ADCEBA	8+2+6+17 = 33	(!)	
ADEBCA	12+6+1+3 = 22	(!)	
ADECBA	12+15+8+17 = 52	(!)	

Zusammenfassung

- Vollständige Enumeration ist (fast) immer möglich (außer bei unendlichen Wertebereichen), aber sehr ineffizient.

- Es lohnt sich zu überlegen, in welche Klasse ein zu lösendes Problem fällt; dann gibt es höchstwahrscheinlich schon eine geeignte Problemlösungstechnik dafür.

- Es lohnt sich **sehr**, über einen guten Algorithmus nachzudenken, bevor man ein Programm schreibt (siehe N-Damen-Problem, kürzeste Wege)