Fachhochschule
Kaiserslautern
University of Applied Sciences

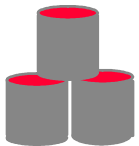
Informatik und
Mikrosystemtechnik
Zweibrücken

Grundlagen der Informatik
– Algorithmenmuster & Graphen –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Algorithmenmuster

- Ziel der Verwendung von Algorithmenmustern: Entwicklung algorithmischer Muster für bestimmte Problemklasse, die dann jeweils an eine konkrete Aufgabe angepasst werden können.
  Grundprinzip kann auf verschiedene Weise konkretisiert werden:
  - ⇨ Lösung wird an einem einfachen Vertreter der Problemklasse vorgeführt und dokumentiert. Späterer Entwickler kann die Problemlösungsstrategie verstehen und auf sein Programm übertragen.
  - ⇨ Bibliothek von Mustern (Design Pattern) wird erstellt und kann zur Generation eines abstrakten Programmrahmens genutzt werden. Die freien Stellen des Programmrahmens werden dann problemspezifisch ausgefüllt.
  - ⇨ Moderne Programmiersprachen (Objekte, Komponenten, Beans) benutzen parametrisierte Algorithmen, um Algorithmenmuster als lauffähige Programme zur Verfügung zu stellen und diese dann an konkretes Problem anzupassen.

Algorithmenmuster

- Im folgenden werden wir verschiedene Algorithmenmuster kennen lernen:
  - ⇨ Greedy-Algorithmen
  - ⇨ Divide-and-Conquer
  - ⇨ Backtracking
  - ⇨ Dynamische Programmierung

Greedy Algorithmen

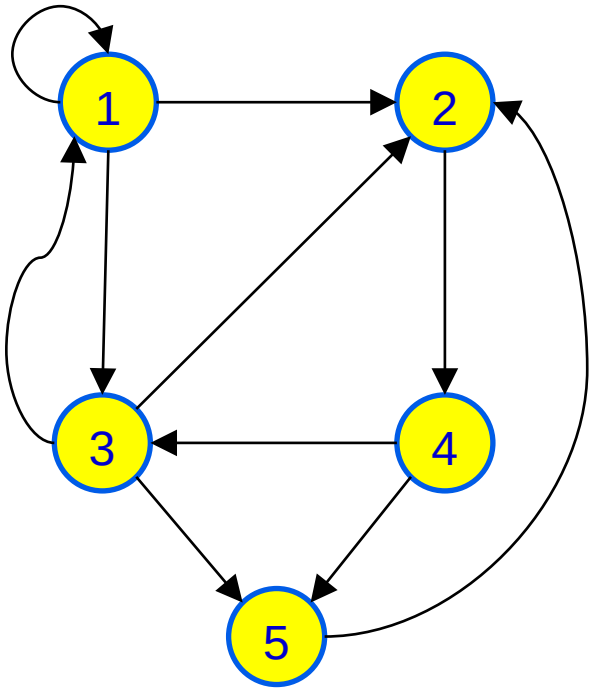
- Die „gierigen“ Algorithmen zeichnen sich durch folgende Merkmale aus:
  - ⇨ Die Lösung wird schrittweise berechnet
  - ⇨ In jedem Schritt wird immer ein lokales Optimum angestrebt
- Typische Anwendung: Optimierungsprobleme
- Für bestimmte Probleme können so global optimale Lösungen gefunden werden
  - ⇨ Z.B. minimaler aufspannender Baum
- Lokale Optima sind für viele NP-vollständige Probleme erreichbar, wie:
  - ⇨ Rucksackproblem
  - ⇨ Problem des Handlungsreisenden
  - ⇨ …

Beispiel: Graphentheorie

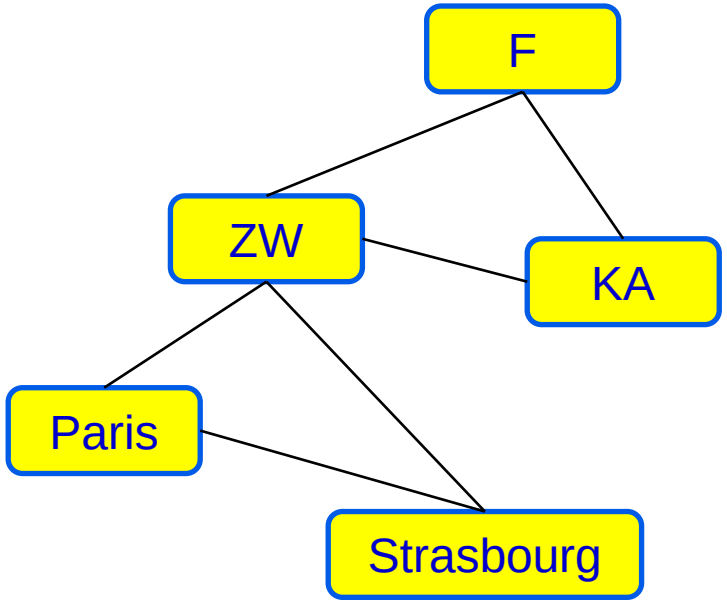
- Wichtige „gierige“ Algorithmen gibt es unter anderem bei der Bearbeitung von Graphen
- Graphen helfen bei der Lösung vieler praktischer Probleme!
  - ⇨ Wie findet man die kürzeste Verbindung zwischen 2 Orten?
  - ⇨ Wie transportiert man eine Ladung am billigsten von A nach B?
  - ⇨ Wann kann ein Projekt frühestens fertig sein?
  - ⇨ Wie muss man eine Reiseroute planen, um jeden Ort auf einer Liste genau einmal zu besuchen?
  - ⇨ …
- Im folgenden daher zunächst eine kurze Einführung in den Bereich der Graphentheorie

Was ist ein Graph?

- Beispiele:
  - ⇨ ein gerichteter Graph

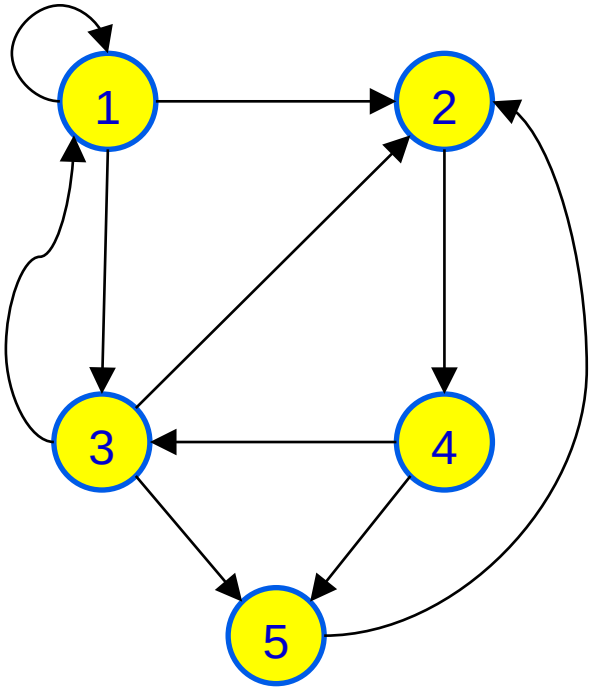


ein ungerichteter Graph



Definition: Gerichteter Graph G

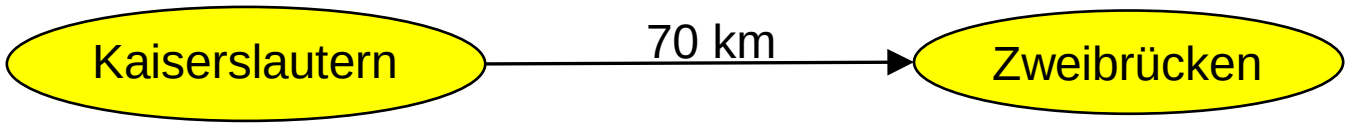
- 1. Eine Menge **N** von *Knoten* und
  2. eine 2-stellige Relation **A** auf N
- A wird die Menge der *Kanten* von G genannt
  - ⇨ Kanten sind Knotenpaare
  - ⇨ gerichtete Kanten werden *Bogen* genannt
- Beispiel:
  - ⇨ G = (N, A)
    N = { 1, 2, 3, 4, 5 }
    A = { (1,1) (1,2) (1,3) (2,4) (3,1) (3,2)
    (3,5) (4,3) (4,5) (5,2) }



- Beachten:
  - ⇨ Jeder ungerichtete Graph kann gleichwertig als gerichteter Graph interpretiert werden, bei dem zu jeder Kante (a,b) auch eine Kante (b,a) existiert.

Weitere Definitionen

- Als *Schlinge* wird ein Bogen von einem Knoten auf sich selbst bezeichnet
- Für Bogen (u, v)
  - ⇨ u ist *Vorgänger* von v
  - ⇨ v ist *Nachfolger* von u
- Knoten und Kanten können mit *Markierungen* versehen werden.
  - ⇨ Beispiel:



Aufspannende Bäume

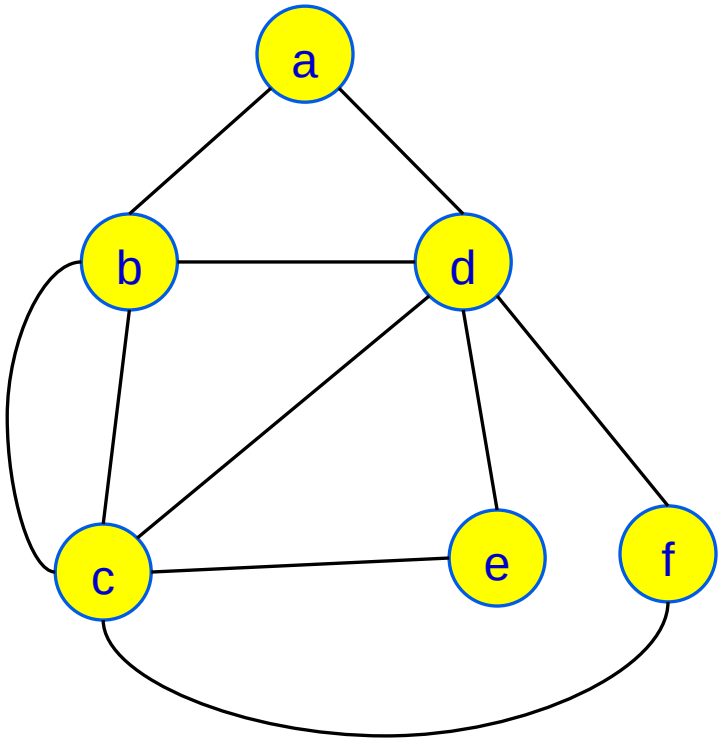
- Definition: Ein *Spannbaum* oder auch *Aufspannender Baum* eines ungerichteten Graphen G (N, A) ist ein Graph (N, A'), für den gilt:
  - ⇨ A' ⊂ A
  - ⇨ zwischen jeweils zwei Knoten aus N existiert ein Pfad in A'
  - ⇨ es gibt keine Zyklen in (N, A')

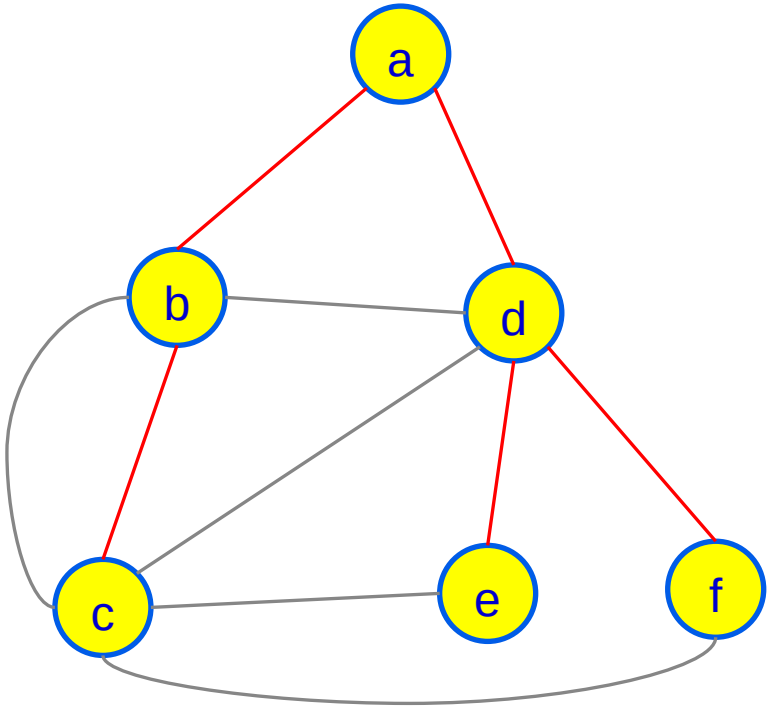
Aufspannende Bäume - Bemerkungen

- Aufspannende Bäume existieren nur in zusammenhängenden Graphen.
- Für jeden Graph mit genau einer zusammenhängenden Komponente existiert immer ein aufspannender Baum.
- Jeder Knoten lässt sich zur Wurzel eines solchen Baumes machen.
- Praktische Bedeutung in vielen Bereichen:
  - ⇨ Erstellung zusammenhängender Netzwerke
  - ⇨ Wegeermittlung
    - bei Berücksichtigung von Kosten → minimaler Spannbaum
  - ⇨ …

Aufspannender Baum - Beispiel

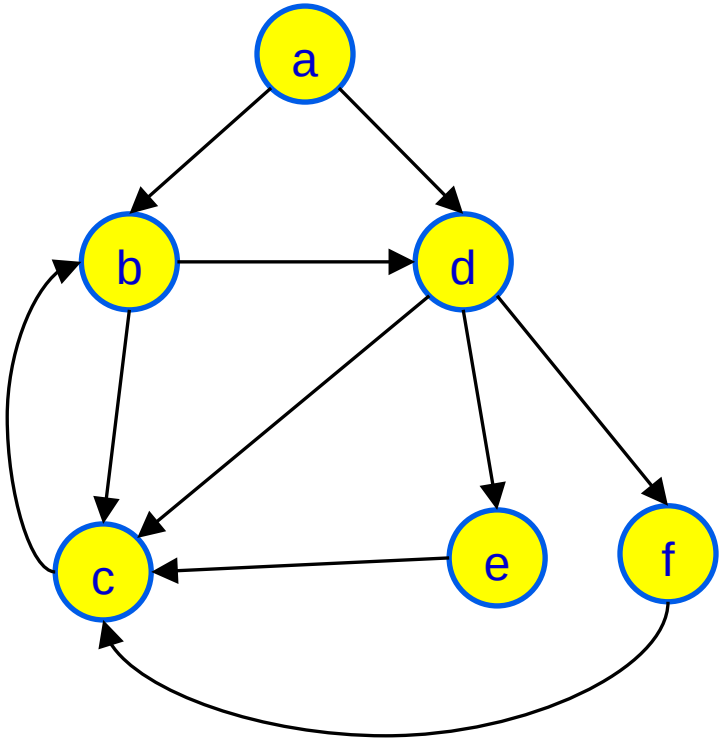
- Ein möglicher aufspannender Baum:

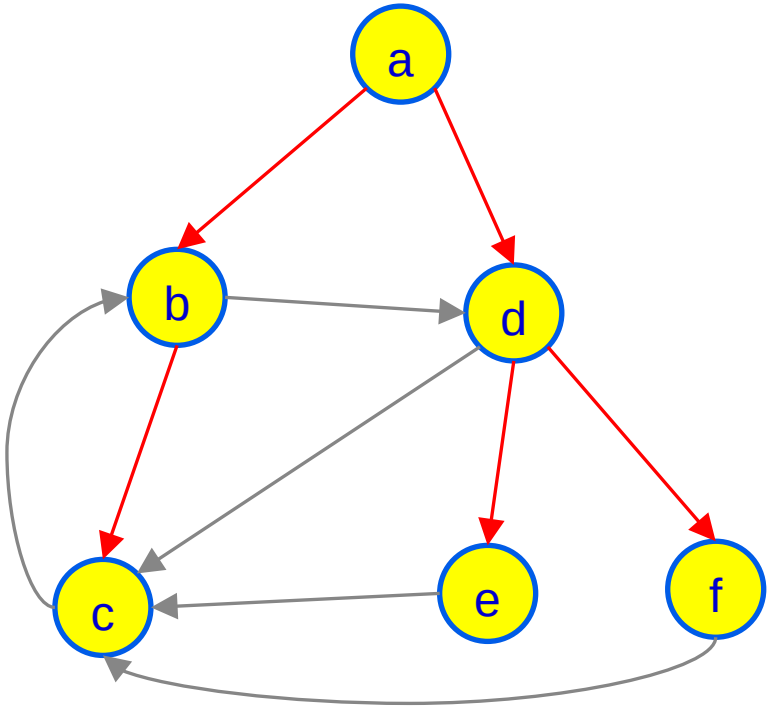




Anmerkung zu gerichteten Graphen

- Das Problem gibt es in ähnlicher Form auch bei gerichteten Graphen
  - ⇨ hier gilt es ausgehend von einer Wurzel einen Spannbaum zu finden:





Minimal aufspannender Baum

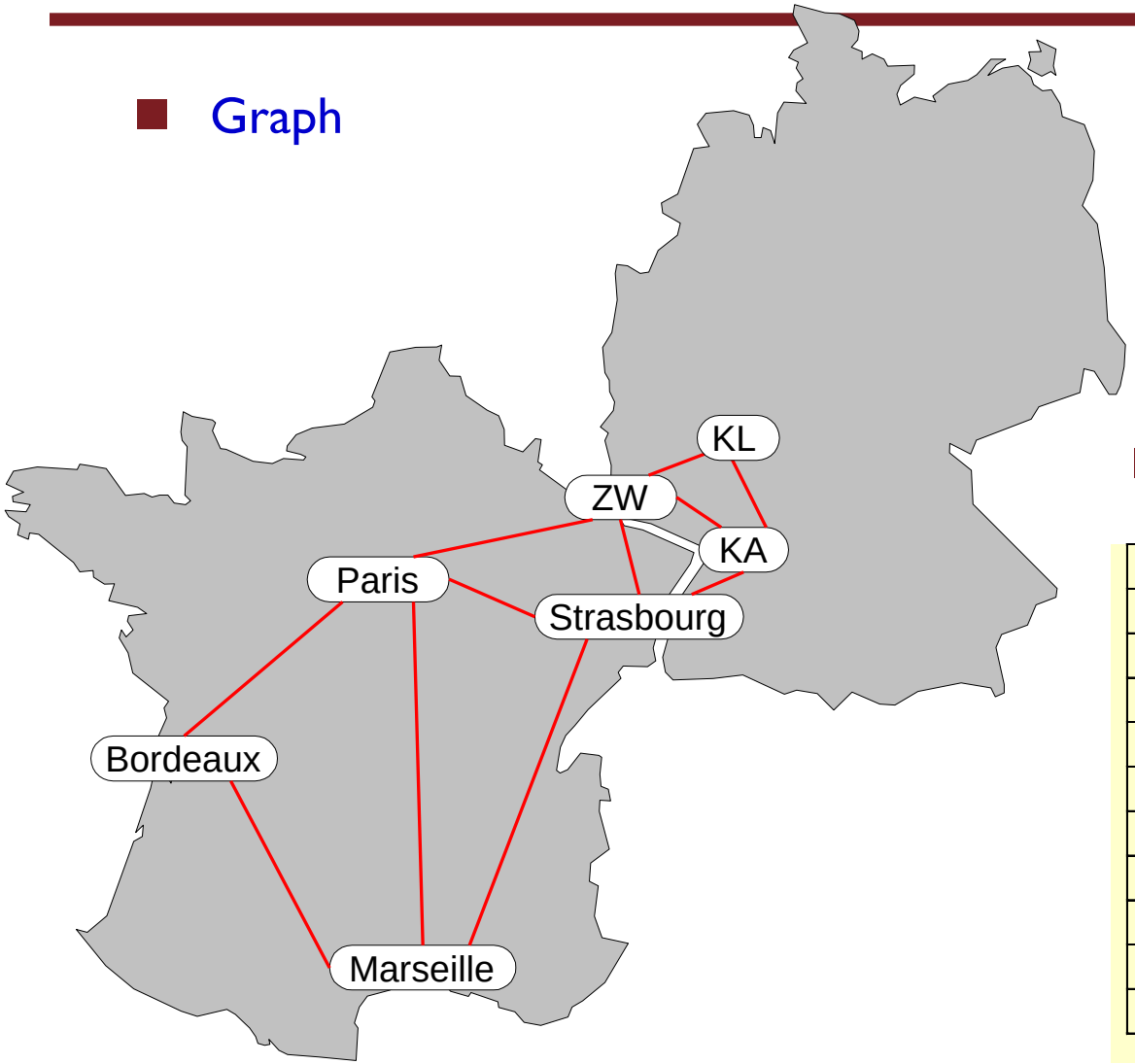
- Definition1:
  Ein *Distanzgraph* ist ein Graph (N, A) mit Markierungen an den Kanten, die die „Länge“ (oder „Kosten“) der Kante repräsentieren.
  - ⇨ Kanten A = { (u, v, m) | u ∈ **N**, v ∈ **N**, m ∈ ℝ }
- Definition2:
  Ein *minimal aufspannender Baum* (N, A) ist der aufspannende Baum in einem zusammenhängenden Distanzgraph, bei dem die Summe der Kantenmarkierungen minimal ist.

Minimal spannender Baum
- Anwendungen -

- Typische Anwendung:
  Minimierung von Verbindungskosten.
  - ⇨ Verlegen von Telefonleitungen
  - ⇨ Planung von hausinternen Versorgungsleitungen
  - ⇨ Logistik: Ermittlung von Reiserouten
  - ⇨ ...

Beispiel: Karte

- Graph

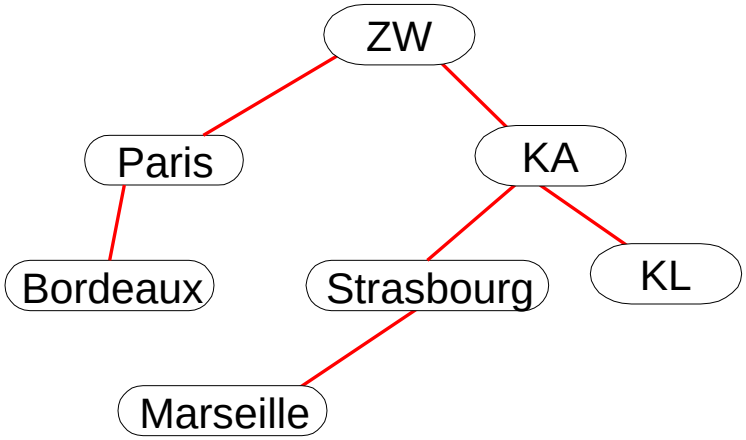


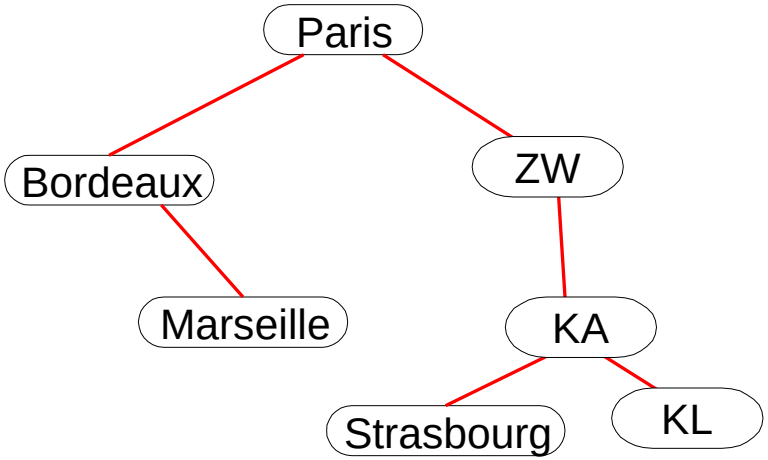
- Entfernungen / Kosten

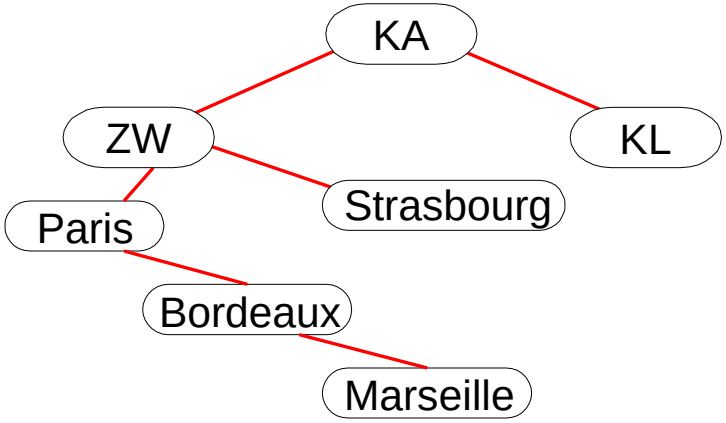
ZW	KL	50
KA	Strasbourg	70
KL	KA	90
ZW	KA	110
ZW	Strasbourg	115
ZW	Paris	425
Paris	Strasbourg	455
Paris	Bordeaux	560
Bordeaux	Marseille	715
Marseille	Strasbourg	720
Marseille	Paris	780

Beispiele

- Aufspannende Bäume







Idee zum Lösungsansatz

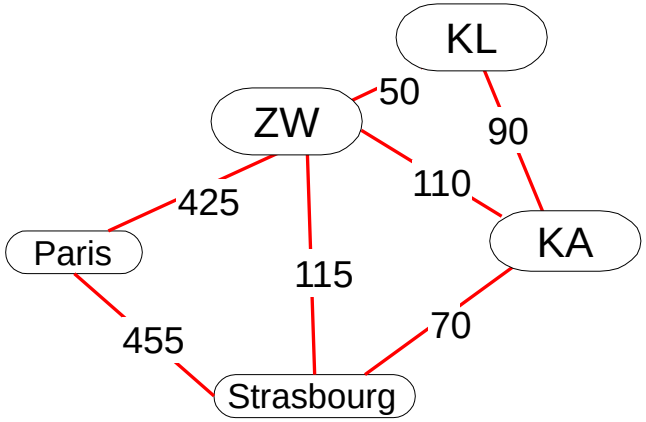
- Für den Graph G (N, A)
  mit bewerteten Kanten A = { (u,v,m) | u ∈ N, v ∈ N, m ∈ ℝ } gilt,
  dass ein minimal aufspannender Baum die Kante (x,y,w) enthält, wenn:
  - ⇨ Sei N' ⊆ N und die Kante (x,y,w) ∈ A und
    x ∈ N' und y ∈ N-N'
  - ⇨ Für w gelte:
    w ≤ m ∀ (u,v,m) ∈ A | u ∈ N' und v ∈ N-N'
- D.h. eine Kante mit minimalen Kosten, die aus einer vorhandenen Knotenmenge hinausführt, gehört zu einem minimal spannenden Baum!

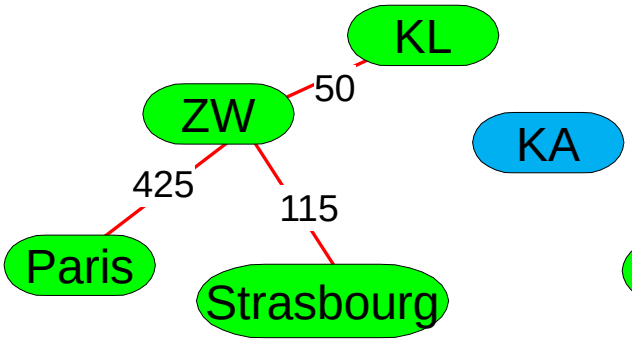
Beweis des Satzes

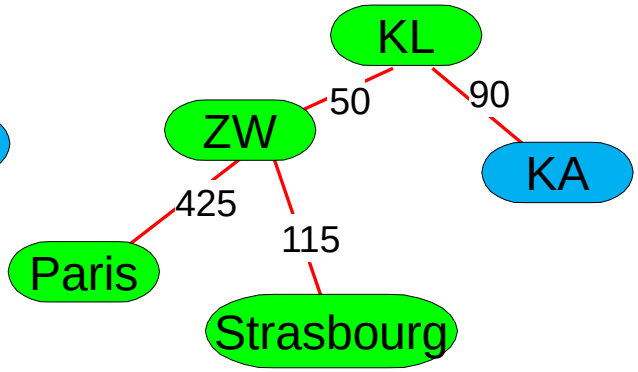
- Annahme: (x,y,w) liege in keinem minimal aufspannenden Baum.
  - ⇨ Sei B ein minimal aufspannender Baum
  - ⇨ Fügt man in diesen (x,y,w) ein, so erhält man einen geschlossenen Weg W
  - ⇨ W enthält Knoten aus N und N'
  - ⇨ Es muss also noch eine Kante (x2,y2,w2) in W geben mit x2 $\in$ N' und y2 $\in$ N-N'.
  - ⇨ Entfernt man (x2,y2,w2) aus B, so ist B wieder ein aufspannender Baum.
  - ⇨ Nach der Definition von (x,y,w) ist jedoch w2 < w !

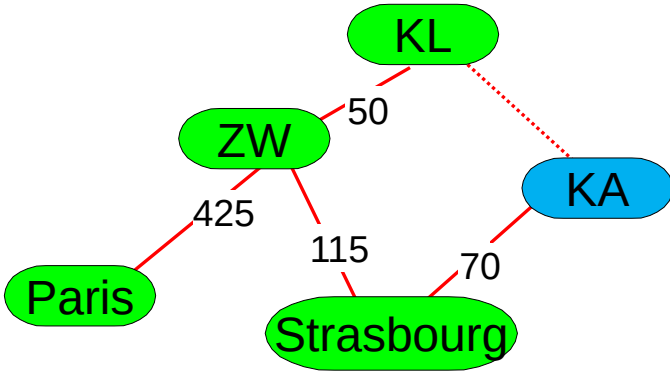
Illustration

- N' = { F, ZW, Paris, Strasbourg }





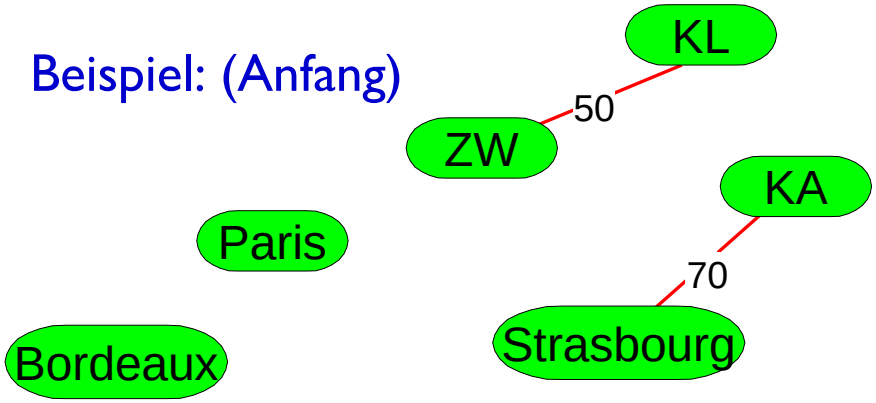




Kruskals Algorithmus (1956)

- Vorgehen:
  - ⇨ Sortiere alle Kanten in aufsteigender Reihenfolge ihrer Markierungen
  - ⇨ Betrachte alle Kanten der Reihe nach
  - ⇨ Wenn sich die Enden der zu betrachtenden Kante in unterschiedlichen Komponenten befinden, dann wählen wir die Kante,
  - ⇨ Falls das Ende in der gleichen Komponente befindet, dann wird sie nicht benutzt.

- Beispiel: (Anfang)



ZW	KL	50
KA	Strasbourg	70
KL	KA	90
ZW	KA	110
ZW	Strasbourg	115
ZW	Paris	425
Paris	Strasbourg	455
Paris	Bordeaux	560
Bordeaux	Marseille	715
Marseille	Strasbourg	720
Marseille	Paris	780

- Warum wird dies ein minimal aufspannender Baum?

“Gierige” Strategien

- Kruskals Algorithmus gehört zur Klasse der “gierigen” (greedy) Algorithmen
  - ⇨ Jede Entscheidung wird lokal gefällt.
  - ⇨ Die Entscheidung führt zu dem (momentan) bestmöglichen Ergebnis.

- Anmerkung: Nicht immer führen lokal optimale Entscheidungen zu einem globalen Optimum!
  - ⇨ Gier macht sich nur manchmal bezahlt
  - ⇨ Hier schon!

Laufzeit von Kruskals Algorithmus

m = Anzahl der Kanten,
n = Anzahl der Knoten

- Bei Verwendung von Adjazenzlisten
  - ⇨ Alle Kanten werden in O (m) gefunden
- Zu Beginn müssen die Kanten gemäß den Markierungen sortiert werden
  - ⇨ Aufwand für Sortierung von m Elementen:
    O (m log m)
- Anschließend untersuchen der Kanten
  - ⇨ Für Ausführen der find_root und merge Aufrufe:
    O (m log n)

Laufzeit von Kruskals Algorithmus (2)

- Aufwand für Sortierung + Aufbau des Baumes
  - ⇨ O (m log m) + O (m log n) = O (m (log n + log m))
- Zusätzlich gilt jedoch, dass es max. n(n-1)/2 Knotenpaare gibt
  - ⇨ $m <= n^2$
  - ⇨ log m <= 2 log n
  - ⇨ m (log n + log m) <= 3m log n
- Daraus ergibt sich als obere Schranke für Kruskals Algorithmus:
  - ⇨ O (m log n)

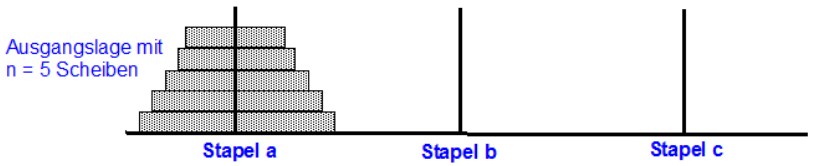
m = Anzahl der Kanten,
n = Anzahl der Knoten

Verbesserungen

- Beobachtung: Es werden maximal n-1 Kanten in den Baum eingefügt!
- Prims Algorithmus (1957)
  - ⇨ Sehr einfach
  - ⇨ basiert nicht auf Vereinigung von Teilgraphen
  - ⇨ erweitert einen Graph sukzessive
  - ⇨ Problem: Auswahl der nächsten Kante mit der kleinsten Bewertung.
  - ⇨ Anordnung der Kanten in Prioritätenwarteschlange
  - ⇨ Aufwand: O (m + n log n)
- Bis heute kein Algorithmus mit O (m) bekannt.

Divide-and-Conquer

- Grundidee: Gesamtproblem in mehrere Teilproblem aufspalten, diese separat lösen und anschließend zu Gesamtlösung zusammenfügen.
  - ⇨ Kann auch wieder auf Teilprobleme angewendet werden, anschließend wieder auf deren Teilprobleme etc., bis die Problemgrößen so klein sind, dass eine direkte Lösung in einfacher Weise angegeben werden kann
- Ist jedes Teilproblem wieder von derselben Art wie das ursprüngliche Problem, erhält man somit unmittelbare rekursive Algorithmen
  - ⇨ Verfahren ist also klassischer Einsatz des Rekursionsprinzips in der algorithmischen Lösung
  - ⇨ Beispiel bereits kennengelernt:
    Lösung des „Türme von Hanoi“ Problems



Divide-and-Conquer: Allgemeines Schema

```
if ( /* Problem P klein */ ) {
   // Explizite Lösung von P angeben;
} else {
   // Teile in Teilprobleme P_1 bis P_n auf;
   for (int i = 1; i <= n; i++ )  {
      // Löse Problem P_i
   }  // for

   // Kombiniere Teil-Lösungen zu Gesamt-Lösung

} // else
```

Divide-and-Conquer: Beispiel Binäre Suche

- Voraussetzung; Wir haben ein Array der Größe n in dem die Einträge sortiert sind
- Aufgabe; Prüfen, ob ein Wert vorhanden ist, bzw. an welcher Position er ggf. gespeichert ist
- Verfahren:
  - ⇨ Wähle den mittleren Eintrag und prüfe ob er dem gesuchten Wert entspricht. Falls ja, sind wir fertig → Gefunden!
  - ⇨ Ist n == 1, sind wir fertig → Nicht gefunden!
  - ⇨ Falls der gesuchter Wert kleiner ist,
    dann das Verfahren mit der linken Hälfte wiederholen
  - ⇨ Falls der gesuchte Wert größer ist,
    dann das Verfahren mit der rechten Hälfte wiederholen

Divide-and-Conquer: Beispiel Turnierplan I

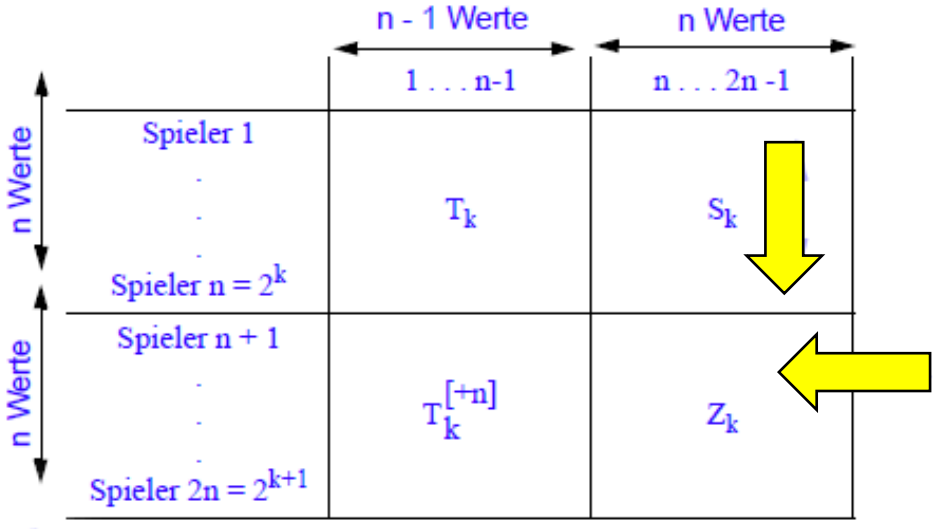
- Aufgabe: Konstruktion von Turnierspielplänen für eine vorgegebene Anzahl von Spielern, sodass
  - ⇨ Jeder Spieler mit jedem Spieler zusammentrifft
  - ⇨ Jeder einzelne Spieler nur einmal am Tag spielt
- Der Einfachheit halber nehmen wir an,
  die Anzahl der Spieler n sei immer eine 2er-Potenz
  - ⇨ $n = 2^k$
- Folgerung: Mindestens n-1 Spieltage notwendig
  - ⇨ Jeder der n Spieler spielt gegen n-1 andere Spieler

Divide-and-Conquer: Beispiel Turnierplan I

- Zur Zerlegung des Problems nötig:
  - ⇨ Für die kleinste Problemgröße **sofort** die Lösung angeben können!
  - ⇨ Aus einem bekannten Turnierplan T_k für $n = 2^k$ Spieler muss eindeutig ein Turnierplan T_{k+1} für $m = 2 * n = 2^{k+1}$ konstruiert werden können
    - auf diese Weise kann das Prinzip der Rekursion angewandt werden
- Also: Plan für $n = 2^k$ ist bekannt
  - ⇨ Aufgabe: Plan für $n = 2^{k+1}$ ermitteln
- Dazu wird ein Spielplan (aller Spieler) als Matrix M notiert, wobei der Eintrag m_{ij} den Gegner des Spielers i am Tag j bestimmt.
  - ⇨ Wie kann man an Hand der Matrix erkennen, dass Spielplan korrekt ist?

Divide-and-Conquer: Beispiel Turnierplan II

- Die Rekursion (k → k+1) basiert auf dem folgenden Muster:



- Die **Matrix** $T_k[^{+n}]$ ist die ursprüngliche Matrix Tk mit jeweils um den Wert n erhöhten Elementen, d.h. strukturell dieselbe Lösung wie T_k.
- **Z_k** ist eine (n x n)-Matrix, die durch zyklisches Verschieben der **Zeile** (1, 2,..., n) erzeugt wird, und zwar zyklischer Links-Shift um eine Position.
- **S_k** ist eine (n x n)-Matrix, die durch zyklisches Verschieben der **Spalte** (n+1,..., 2n) erzeugt wird, und zwar zyklischer Unten-Shift um eine Position.
- Wie sieht der Turnierplan für T_2 also für $n = 2^2 = 4$ Spieler aus?

Divide-and-Conquer: Beispiel Turnierplan III

- Das Ergebnis für T_1 für $n = 2^1 = 2$

	Tag 1
Spieler 1	2
Spieler 2	1

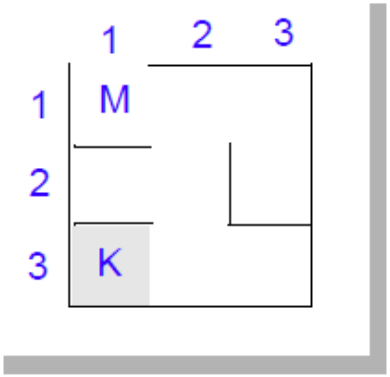
- Das Ergebnis T_2 für $n = 2^2 = 4$ Spieler
  und $2 * n - 1 = 3$ Tage sieht damit wie folgt aus:

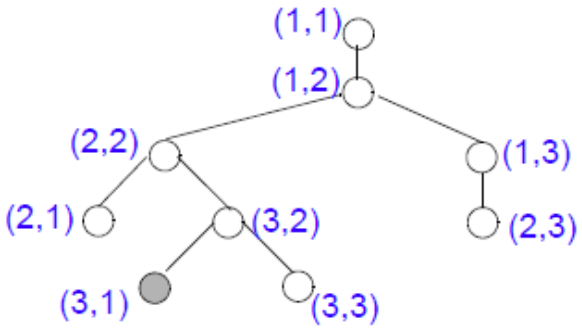
	Tag 1	Tag 2	Tag 3
Spieler 1	2	3	4
Spieler 2	1	4	3
Spieler 3	4	1	2
Spieler 4	3	2	1

Backtracking

Basiert auf Versuch-und-Irrtum-Prinzip (trial and error)

- Backtracking („Zurückverfolgen“) für Such- und Optimierungsprobleme:
  - ⇨ Backtracking realisiert eine allgemeine systematische Suchtechnik, die einen vorgegebenen Lösungsraum komplett bearbeitet.
  - ⇨ I.d.R. gerät man bei der Suche in „Sackgassen“ und geht dann zur nächsten noch nicht bearbeiteten Abzweigung „zurück“, bis man alle Verzweigungen abgearbeitet hat.
- Im folgenden wird das Prinzip des Backtracking am Beispiel einer Labyrinth-Suche demonstriert
  - ⇨ Aufgabe: Wie kommt die Maus zum Käse?
  - ⇨ In dem Labyrinth ist ein Startpunkt **M** der Maus zusammen mit dem der Ort **K** des Käses





Backtracking: Allgemeines Schema

```
static void backtrack (Konfiguration K) {

   for ( /* jede direkte Nachfolger-Konfiguration K' von K */ )  {

      if ( /* K' ist eine Lösung */ )  {
      // Gib K' aus
      } // if

      backtrack (K');

   } // for

} // backtrack

public static void main (String[ ] args)  {

   backtrack (K1);

} // main
```

Einsatz des Backtracking

- Die beschriebenen Eigenschaften machen Backtracking für einige Anwendungsgebiete besonders geeignet:
  - ⇨ In Spielprogrammen wie Schach, Dame usw. entsprechen Konfigurationen den Aufstellungen der Figuren, die Nachfolger den Spielzügen. (→ Spielstrategie implementieren)
  - ⇨ In logischen Programmen/Aussagen können Erfüllbarkeitstests mit Hilfe von Backtracking-Strategien realisiert werden. Die Konfigurationen stellen Programmzustände dar, die Nachfolger stellen mögliche Programmabläufe bzw. Nachfolgezustände im Programm dar.
  - ⇨ Generell wird Backtracking in der Optimierungstheorie zur Lösung von Planungs- und ähnlichen Problemen eingesetzt.

Weitere Beispielanwendungen für Backtracking

- Labyrinthartige Suchprobleme
- Im Grunde alle Probleme, die sich durch vollständiges Durchsuchen des Lösungsraumes auf trial-and-error Basis lösen lassen.
  - ⇨ Z.B. Generieren von Spielelösungen
  - ⇨ Bewerten von Spielzügen z.B. für Schachprogramme

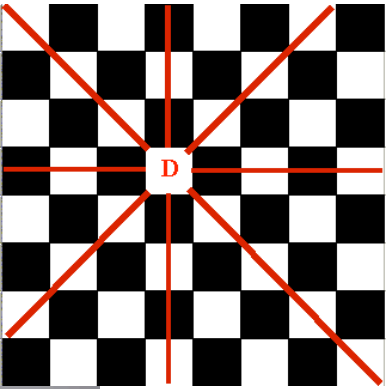
	3							
			1	9	5			
		8					6	
8				6				
4			8					1
				2				
	6					2	8	
			4	1	9			5
							7	

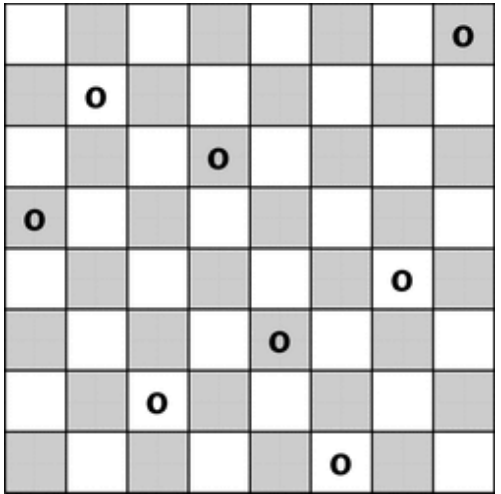
Varianten des Backtracking

- Nachteil: Hoher Rechenaufwand – schlechtes Laufzeitverhalten
- Daher oft modifizierte Backtracking-Varianten:
  - ⇨ Algorithmus markiert/bewertet Lösungen, aus denen nach dem Lauf die beste ausgewählt wird.
  - ⇨ Statt alle Lösungen auszugeben, wird nach der ersten gefundenen Lösung abgebrochen oder es wird eine maximale Rekursionstiefe vorgegeben.
    - Schwierigkeitsgrad entspricht der Anzahl Vorausschauschritte
- *Branch-and-Bound*-Varianten vermeiden frühzeitig Sackgassen.
  - ⇨ Es werden nur Zweige verfolgt, die eine Lösung prinzipiell zulassen. Dazu müssen die Zweige vorher geeignet abgeschätzt werden.

Backtracking-Beispiel: n-Damen-Problem I

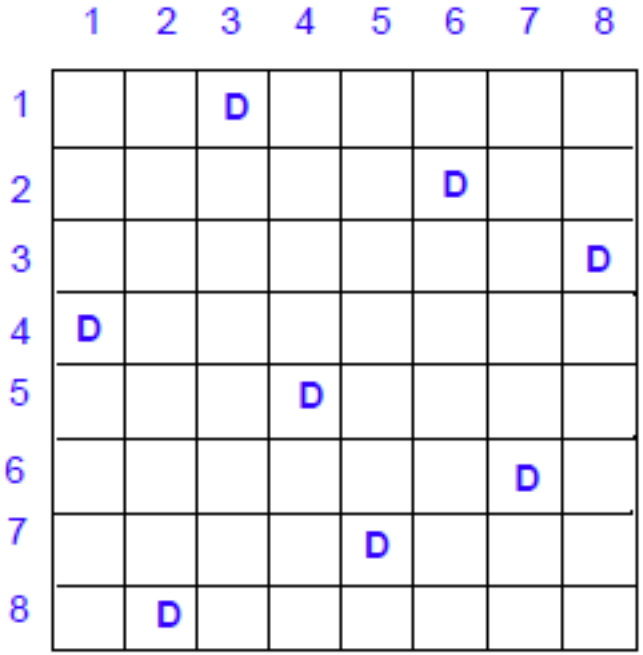
- Es sind alle Konfigurationen von n Damen auf einem n x n Schachbrett zu bestimmen, bei denen keine Dame eine andere Dame „bedroht“.
  - ⇨ Farbe ist irrelevant
- Eine Lösung besitzt die folgenden Eigenschaften:
  - ⇨ Keine Zeile kann mehr als eine Dame enthalten. Da genau n Damen unterzubringen sind, muss jede Zeile genau eine Dame enthalten.
  - ⇨ Analog muss jede Spalte genau eine Dame enthalten.
  - ⇨ Entfernen einer Dame aus einer Konfiguration, die Bedrohungs-frei ist, führt wiederrum zu einer Bedrohungs-freien Konfiguration.

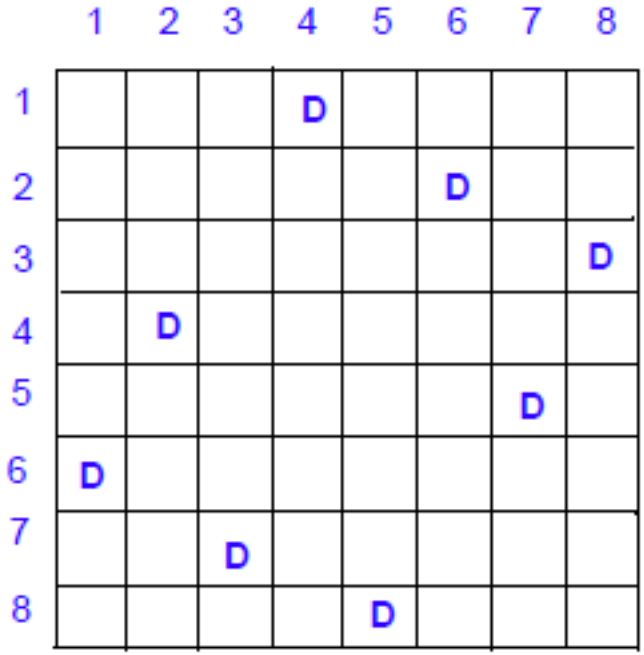




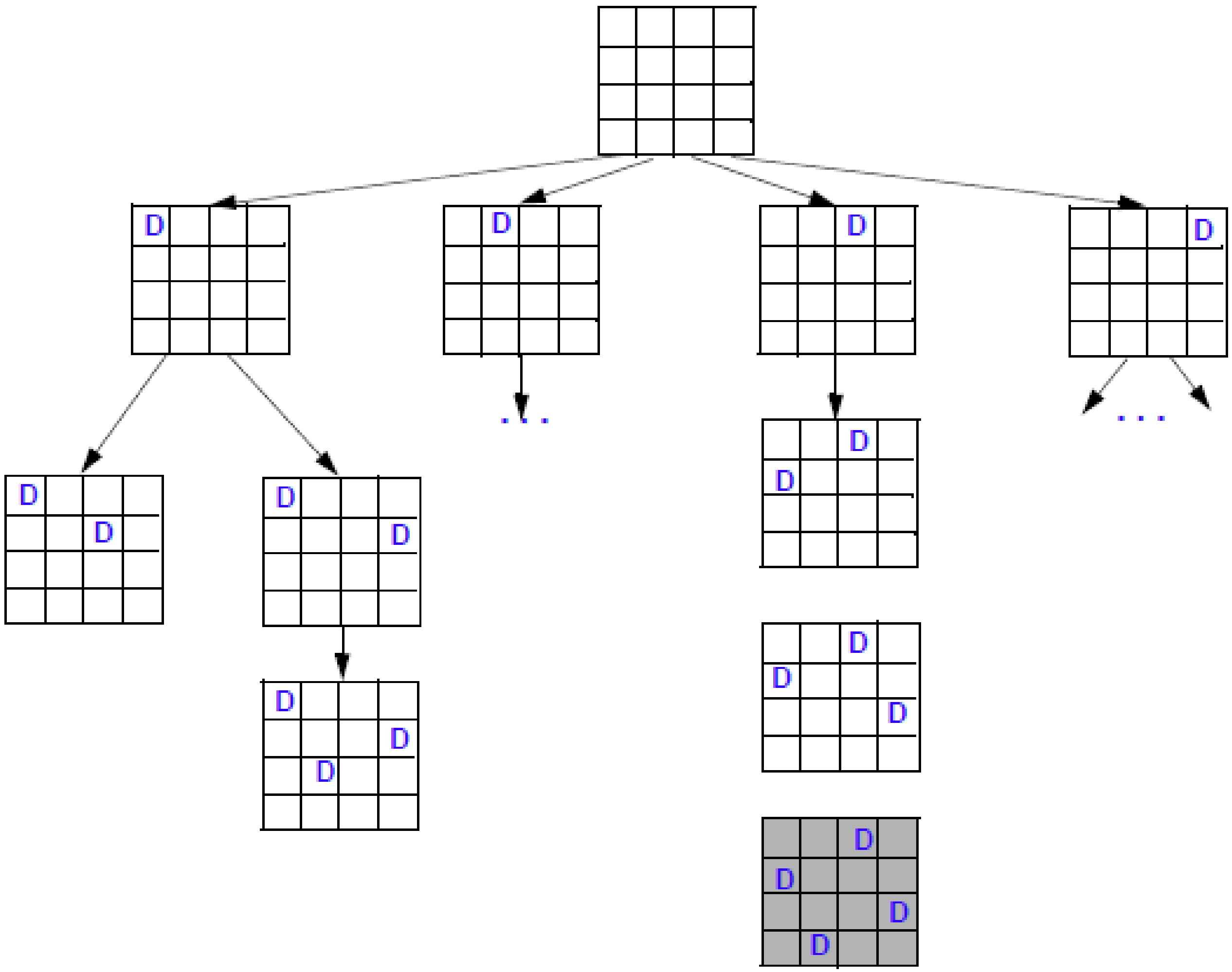
Backtracking-Beispiel: 8-Damen-Problem II

- Bei 8 Damen existieren 92 Lösungen





Backtracking-Beispiel: 4-Damen-Problem



Backtracking: Lösung N-Damen-Problem

```
static void  platziereDameInZeile (int zeile) {
    // Für eine feste Spalte wird eine Damenzuordnung gesucht.
    for (int spalte = 0; spalte < anzDamen; spalte++) {
        // Platziere Dame auf Position (zeile, spalte), falls keine Bedrohung
        if ( !testeBedrohung (zeile, spalte) ) {
           // Setze Dame auf Feld (zeile, spalte) und passe Hilfs-Strukturen an
           . . .
           if (zeile == anzDamen - 1)
             ausgabe ();                          // Alle Damen erfolgreich verteilt
           else
             platziereDameInZeile (zeile + 1); // Nächste Dame auf Folge-Zeile verteilen

           /* Programmkontrolle erreicht diese Stelle, wenn Rekursion für aktuelle Spalte
           über alle (möglichen) Folge-Zeilen abgeschlossen ist, d.h. alle Lösungen (if-
           Fall) bzw. alle Sackgassen (else-Fall) wurden gefunden bzw. durchlaufen.
           In diesem Fall soll Backtracking angewendet werden, also im Suchraum
           zurückgegangen werden. Dazu wird die letzte (aktuelle) Dame
           vom Feld (zeile, spalte) entfernt */
            . . .
        } // Keine Bedrohung
    } // for spalte

} // platziereDameInZeile
```

Dynamische Programmierung

- Vereint Aspekte der drei bisher vorgestellten Algorithmenmuster:
  - ⇨ Vom Greedy-Ansatz: Wahl optimaler Teillösungen
  - ⇨ Vom Divide-and-Conquer und Backtracking: rekursive Vorgehensweise basierend auf einem Konfigurationsbaum
  - ⇨ Während Divide-and-Conquer-Verfahren unabhängige Teilprobleme durch rekursive Aufrufe lösen, werden bei dynamischer Programmierung abhängige Teilprobleme optimiert gelöst, indem mehrfach auftretende Teilprobleme nur einmal gelöst werden.
- Der Anwendungsbereich der dynamischen Programmierung sind Optimierungsprobleme analog zu Greedy-Algorithmen.
  - ⇨ Im Gegensatz zu den Greedy-Algorithmen werden Verfahren der dynamischen Programmierung verwendet, wenn optimale Lösungen benötigt werden und Greedy diese nicht liefern kann.

Fibonacci-Folge

- Der italienische Mathematiker Leonardo von Pisa (Filius Bonacci) fragte sich eines Tages, **wie viele Kaninchen** in einem eingezäunten Gehege pro Jahr geboren werden, wenn man davon ausgeht das:
  - ⇨ jeden Monat ein Paar ein weiteres Paar erzeugt
  - ⇨ Kaninchen zwei Monate nach der Geburt geschlechtsreif sind
  - ⇨ alle Kaninchen unsterblich sind.
- Mit F_n wird die Anzahl der Kaninchen Paare nach n Monaten beschrieben. Für die entsprechende Folge gilt dann:
- $F_0 = 0$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$
- Diese Zahlen werden Fibonacci-Zahlen genannt.
  - ⇨ Die ersten Fibonacci-Zahlen lauten: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Dynamische Programmierung der Fibonacci-Folge

- Die zugrundeliegende Idee ist sehr einfach.
  - ⇨ Statt jeden Rekursions-Aufruf unabhängig voneinander zu betrachten, wird nun ein Gedächtnis verwendet.
  - ⇨ Auf diese Weise wird ein Wert höchstens einmal berechnet.
- Dazu muss das rekursive Programm so umgewandelt werden,
  - ⇨ dass jeder berechnet Wert (letzte Aktion) gespeichert wird und
  - ⇨ die bereits gespeicherten Werte (erste Aktion) geprüft werden, um Neuberechnung zu vermeiden.
- Daher kann man in diesem Fall das ursprünglich verzweigte Rekursions-Programm nahezu automatisch in eine effiziente Variante umwandeln, die nur noch einen linearen Rechenaufwand erfordert.

Beispiel: Fibonacci-Folge

```
static int knownFibos [ ] = new int [n];

static int fibo (int i) {

   if (knownFibos [i] != 0)
     return knownFibos [i];

   if (i < 0)
     return 0;

   int temp = i;  // wichtig für die Eingabewerte 0 und 1
   if (i > 1)
     temp =  fibo (i-1) + fibo (i-2) ;
   knownFibos [i] = temp;

   return temp;

}  // fibo
```