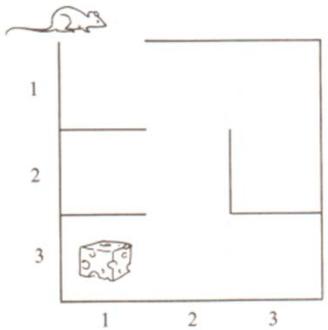
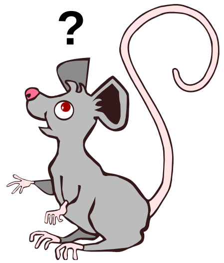
Trial & Error, Backtracking, Branch&Bound

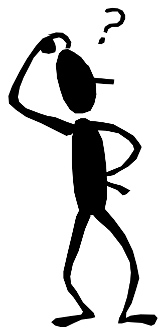




- Sie kennen Probleme, die nur/am einfachsten mit Versuch und Irrtum gelöst werden können (Trial & Error)
- Sie kennen die Vorteile und Nachteile dieses Verfahrens
- Sie wissen was ein Entscheidungsbaum ist
- Sie können einige bekannte Probleme (Labyrinth, Springer Problem) selbständig lösen
- Sie wissen was eine Zielfunktion ist
- Sie wissen wie Branch & Bound Verfahren funktionieren
- Sie wissen was Pruning bzw. cut off bedeutet

Versuch und (Erkennen des) Irrtum(s)

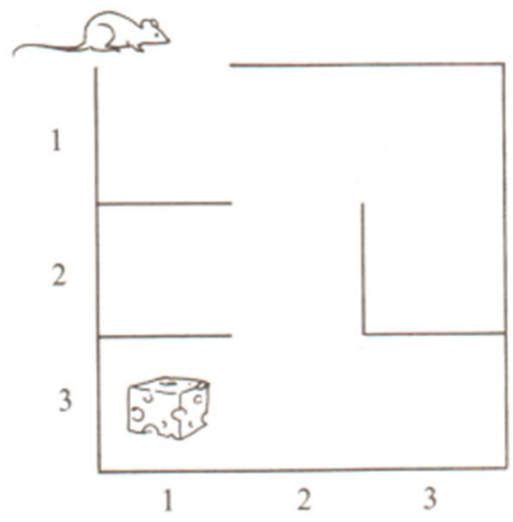
- eigentlich besser wäre: Versuch und Bewertung
- die älteste Lösungsstrategie überhaupt
- Natur:
  - Mutation = Versuch
  - Selektion = Bewertung



- Resultat nach 3 Milliarden Jahren: der Mensch
- Schlussfolgerungen
  - Trial & Error ist ziemlich rechenintensiv
  - Nicht unbedingt beste Lösung als Resultat
  - Zwei mögliche Resultate bei Trial & Error Ansatz
    - beste Lösung
    - akzeptable Lösung (unter den gegebenen Rahmenbedingungen)

Beispiel: Labyrinth

- Problem:
  - Maus sucht Käse
  - Katze sucht Milch/Maus
  - Maus sucht Ausgang (so schnell wie möglich)



- Lösungsalgorithmus

  gehe einen Weg entlang bis Verzweigung

  gehe *einem* der möglichen Wege entlang

  i) bis am Ziel

  ii) oder nicht mehr weiter (Sackgasse)

  gehe zur (letzten) Verzweigung zurück

  versuche noch nicht probierten Weg

  bis es keine nicht probierten Wege mehr gibt

Theseus und der Minotaurus

- Die Frau von Minos, König von Kreta, verliebte sich in einen weissen Stier. Sie ließ sich ein hölzernes Kuhgestell bauen und eine Kuhhaut darüberspannen, um dann in dieses Gestell zu kriechen und sich in diesem mit dem weißen Stier zu vereinigen. Aus dieser Vereinigung ging der **Minotaurus** hervor, eine Gestalt mit menschlichem Körper und dem Kopf eines Stieres.
- Minos ließ für den Minotaurus ein Gefängnis, das Labyrinth in Knossos, erbauen.
- Später musste Athen alle neun Jahre sieben Jünglinge und sieben Jungfrauen als Tributzahlung nach Kreta schicken, die von Minos zum Minotaurus ins Labyrinth geschickt und so diesem geopfert wurden.
- **Theseus**, König von Athen, löste das Problem, indem er sich selbst auf den Weg nach Kreta machte.
- Theseus konnte den Minotaurus besiegen und das Labyrinth wieder verlassen.
- Die kretische Prinzessin Ariadne, die sich in den kühnen Recken verliebt hatte, hatte ihm geholfen, indem sie ihm den bekannten **Ariadnefaden** mitgegeben hatte, mit dem er den Weg aus dem Labyrinth wieder fand.

Theseus schleppt den toten Minotaurus aus dem Labyrinth, um 440-430 v. Chr

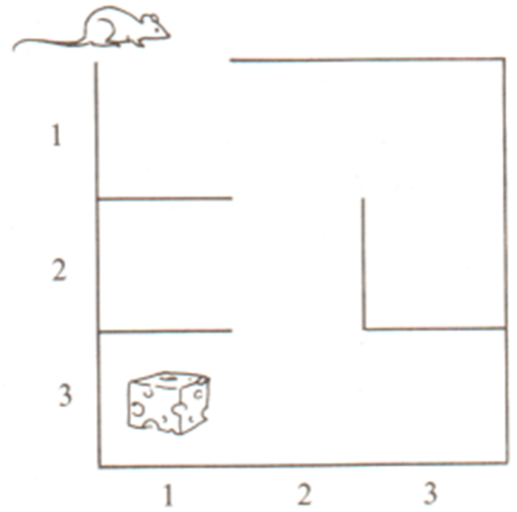
Entscheidungsbaum, Backtracking

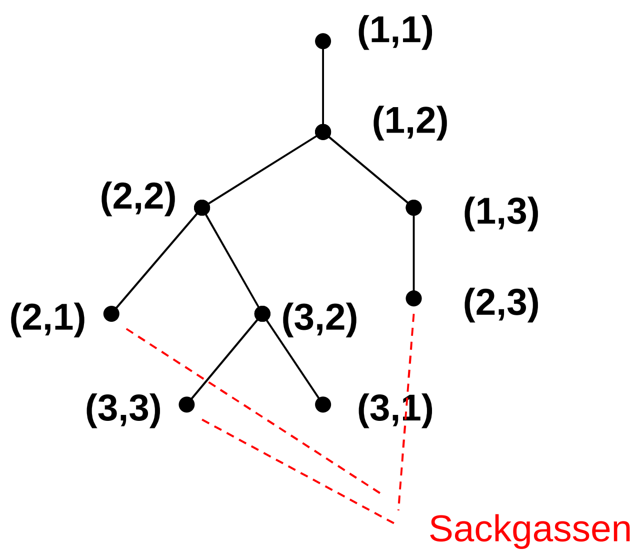
- Es entsteht so ein *virtueller* **Entscheidungsbaum**. Jede Entscheidung (Verzweigung) entspricht darin einem Knoten.

- **Teillösungen** werden systematisch zu Gesamtlösungen **erweitert,** bis eine **Lösung gefunden** oder **Erweitern nicht mehr möglich** ist (→Sackgasse).

- Bei Sackgasse werden einer oder mehrere **Schritte rückgängig gemacht** und von dort aus wird versucht, eine Lösung zu finden.

- Aus Sackgassen einen Weg zurück finden wird als **Backtracking** bezeichnet.





Vorgehen bei Versuch und Irrtum Verfahren

Lösung: Beispiel: Pfad durch das Labyrinth

Mit dem Vorwärtsgehen im Entscheidungsbaum wird die Teillösung erweitert.

solange Lösung nicht gefunden

Erweiterung der bestehenden (Teil-)Lösung möglich?

ja →

füge Erweiterung hinzu

überprüfe, ob erweiterte Lösung zum Ziel führt

ja →Lösung gefunden, Abbruch
(oder Weitersuchen, falls weitere Lösungen gesucht)

nein→nehme Erweiterung zurück

nein →

keine Lösung möglich, Abbruch

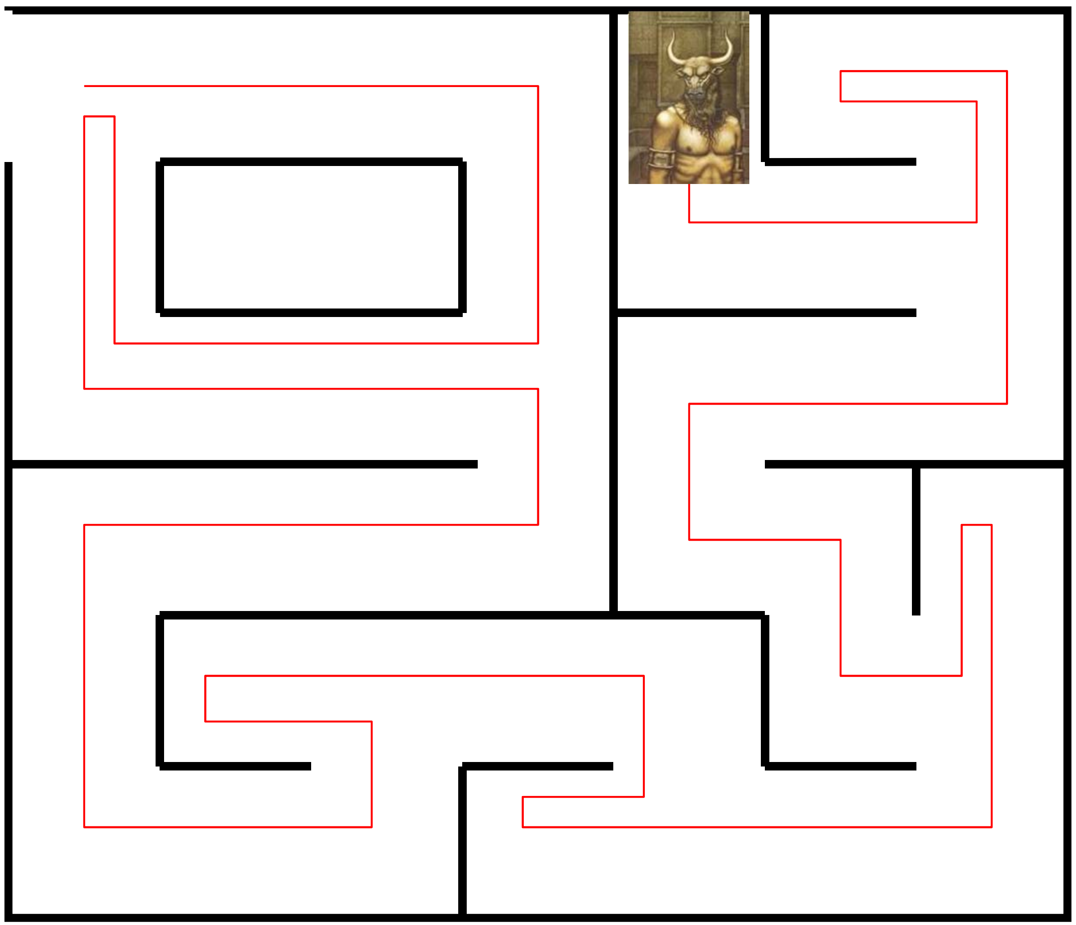
- Es müssen systematisch alle möglichen Erweiterungen durchprobiert werden.

Rekursive Suche im Labyrinth: Pseudocode

Das Labyrinth kann als Graph beschrieben werden

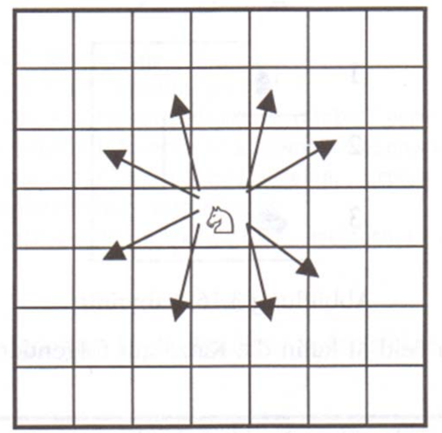
```
boolean search (Node currentNode) {
  mark currentNode;   // wegen Zyklen
  if (currentNode == goal) return true;
  else {
    for all nodes n adjacent to currentNode {
       if (!(marked(n)) {
          if (search(n)) return true;
       }
    }
  }
  unmark currentNode;
  return false;
}
```

Der Weg durch das Labyrinth



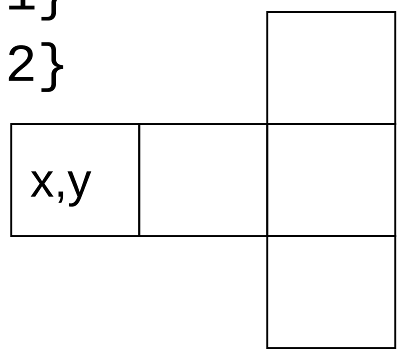
Springerproblem

- Aufgabe
  – von einem bestimmten Schachfeld aus soll ein Springer nacheinander sämtliche Felder des Schachbretts genau einmal besuchen.



- Bewegung des Springers
  – zwei Felder in die eine Richtung und ein Feld in dazu rechtwinkliger Richtung

- `springerX = {2, 2, 1,-1,-2, -2, -1, 1}`
- `springerY = {1,-1, 2, 2, 1, -1, -2,-2}`



Springerproblem: Datenstrukturen und Methoden

- Datenstruktur
  - `int[][] schachbrett new int[n][n];`

- int-Wert in Feld soll angeben, in welchem Zug das Feld besucht wurde.

- werden mit 0 initialisiert

- Methode: `boolean versuch(int x, int y, int nr)`

- x, y : Koordinaten des Feldes
- nr : Nummer des Zuges (>=1)

Springerproblem: Der Algorithmus

Position erlaubt

```
public static boolean gueltigePosition(int x, int y) {
    return (0 <= x) && (x < n) && (0 <= y) && (y < n) &&
        schachbrett[x][y] == 0;
}

public static boolean versuchen(int x, int y, int nr) {
    schachbrett[x][y] = nr; // Feld besetzen
    if (nr == n*n) return true;
    else {
      for (int versuch = 0; versuch < springerX.length;
               versuch++) {
        int xNeu = x + springerX[versuch];
        int yNeu = y + springerY[versuch];
        if (gueltigePosition(xNeu, yNeu)) {
          if (versuchen(xNeu,yNeu,nr+1)) return true;
        }
      }
    }
   schachbrett[x][y] = 0; // Feld freigeben
   return false;
}
```

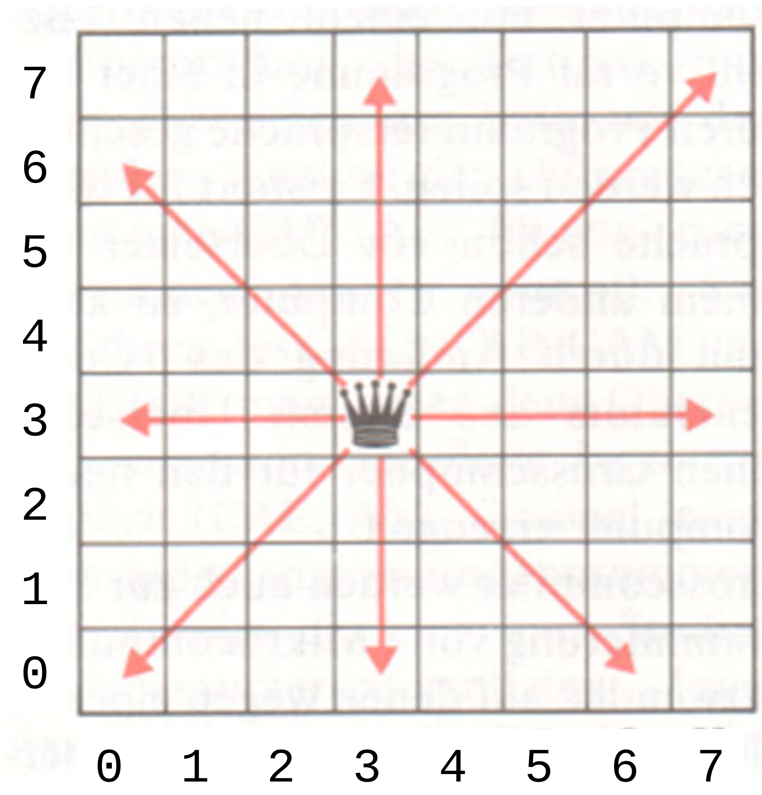
Lösung gefunden

neue Position

Frage: Wie müsste dieser Algorithmus geändert werden, damit *alle* möglichen Lösungen ausgegeben werden?

Das Acht-Damenproblem

- Historisches
  - wurde von C.F. Gauss (1777-1855) gestellt
- Aufgabe
  - es soll eine Stellung für acht Damen auf einem Schachbrett gefunden werden, so dass keine zwei Damen sich gegenseitig schlagen können.
- Bewegung der Dame
  - horizontal, vertikal und diagonal



Acht-Damenproblem: Datenstrukturen und Methoden

- `int[] dameInDerSpalte = new int[n];`
- `boolean[] zeileFrei = new boolean[n];`
- `int diagN = 2*n -1;`
- `boolean[] hauptDiagFrei = new boolean[diagN];`

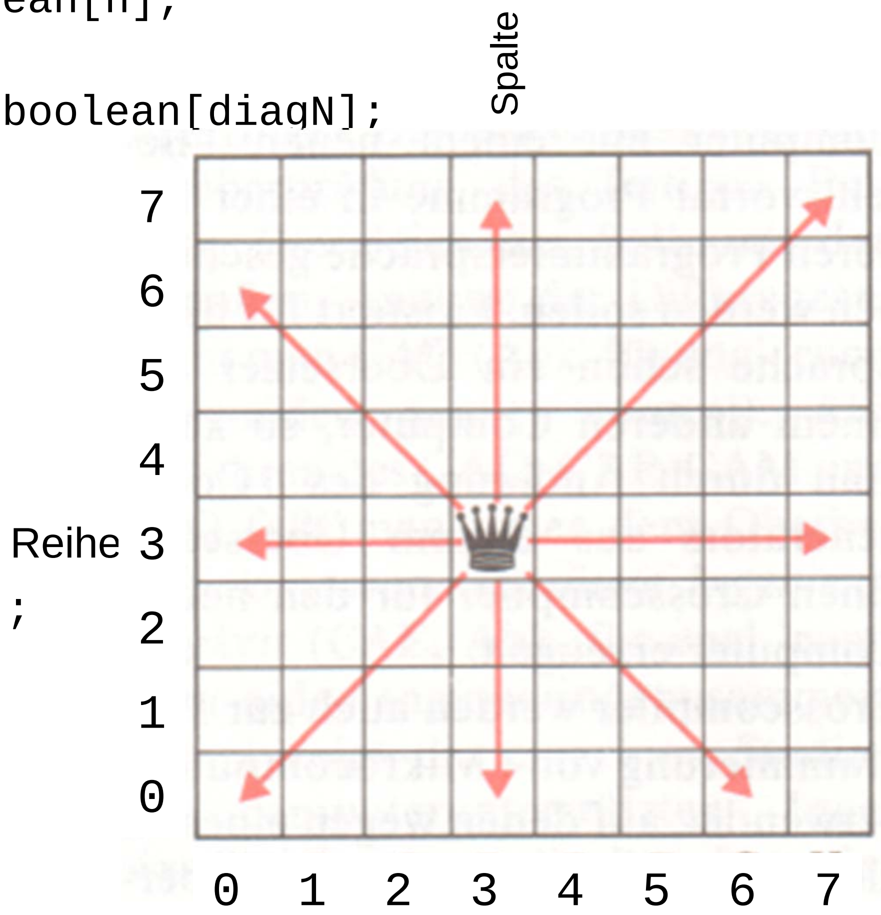
1,5		
	2,4	
		3,3

- `hauptDiag = (x + y) % diagN;`

- `boolean[] nebenDiagFrei =`
  `new boolean[diagN];`

		3,5
	2,4	
1,3		

- `nebenDiag = (diagN+x-y)%diagN;`

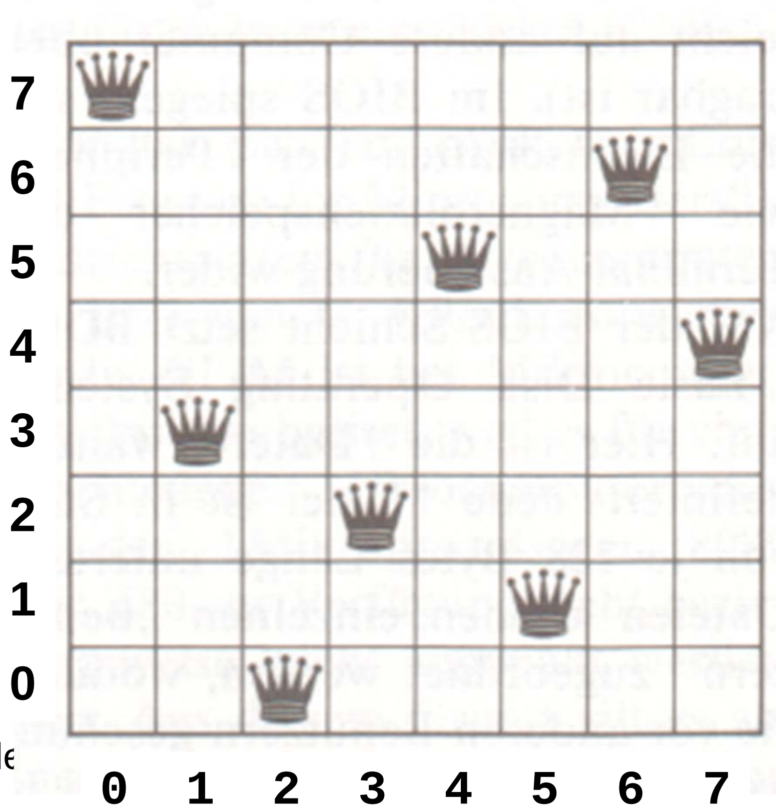


Acht-Damenproblem: Zwei Hilfsmethoden

```
// testet ob Position möglich ist
// Wert in 3 Arrays true -> Position frei

boolean gueltigeDamePosition(int x, int y) {
  return (zeileFrei[y] && hauptDiagFrei[(x + y) % diagN]
            && nebenDiagFrei[(diagN + x - y) % diagN]);
}


// setzt/löscht die Dame von der Position
void setzeDame(int x, int y, boolean val) {
  zeileFrei[y] = !val;
  hauptDiagFrei[(x + y) % diagN] = !val;
  nebenDiagFrei[(diagN + x - y) % diagN] = !val;
  dameInDerSpalte[x] = val? y:-1;
}
```

Acht-Damenproblem: Der Algorithmus

```
boolean versuchen(int x) {
  if (x == n) return true;
  else {
    for (int y = 0; y < n; y++) {
      if (gueltigeDamePosition(x,y)) {
        setzeDame(x,y,true);
        if (versuchen(x+1)) return true;
        setzeDame(x,y,false);
      }
    }
    return false;
  }
}

 void main() {
    versuchen(0);
 }
```



Genereller rekursiver Backtracking Algorithmus

```
public static boolean versuchen(int k) {
    if (LösungGefunden) return true;
    else {
      for all l in ErweiterungVonTeilloesungen {
        if (moeglicheErweiterung(l)) {
          fuegeZuLoesungHinzu(l);
          if (versuchen(k+1)) return true;
          nehmeVonLoesungWeg(l);
        }
      }
      return false;
    }
  }
```

Die kombinatorische Explosion

- Das Problem der Versuch-und-Irrtum-Methode ist, dass meist **alle möglichen Kombinationen** ausprobiert werden müssen.

- z.B. beim Springer max. 8 mögliche Positionen pro Zug →
- 8*8*8 … *8 bei 64 Feldern → $8^{64} = 6.3E57$

- Die Anzahl der möglichen Fälle (Kombinationen) wächst mit 2^n oder n!, d.h. exponentiell

- schon für relativ kleine Werte von n (n ~ 10-100) dauert die Berechnung meist zu lange.

Zuordnungsprobleme: Das Problem der stabilen Heirat

- Es seien n Männer und n Frauen gegeben.
- Jeder Mann und jede Frau haben eine Präferenzliste.
- Aufgabe**:** Finden einer stabilen Konstellation von Paaren.
- Eine Konstellation heisst *instabil*, falls es zwei Leute gibt, die nicht miteinander verheiratet sind, sich aber jeweils ihrem Partner vorziehen.
- Eine Konstellation ist *stabil*, wenn sie nicht instabil ist.

Das Problem der stabilen Heirat

Fritz:	Vreni , Susi , Moni , Jenny , Angi
Kurt:	Vreni , Jenny , Moni , Angi , Susi
Max:	Moni , Susi , Jenny , Vreni , Angi
Roli:	Susi , Vreni , Moni , Angi , Jenny
Sepp:	Vreni , Moni , Susi , Jenny , Angi

Angi:	Sepp , Kurt , Max , Roli , Fritz
Jenny:	Sepp , Roli , Max , Kurt , Fritz
Moni:	Roli , Fritz , Sepp , Kurt , Max
Susi:	Sepp , Max , Kurt , Roli , Fritz
Vreni:	Sepp , Roli , Fritz , Max , Kurt

Fritz	-	Vreni
Kurt	-	Jenny
Max	-	Moni
Roli	-	Susi
Sepp	-	Angi

stabil ?

Fritz	-	Angi
Kurt	-	Jenny
Max	-	Susi
Roli	-	Moni
Sepp	-	Vreni

stabil ?

Stabile Heirat: Datenstrukturen

- Jede Frau führt eine geordnete Liste mit allen Männern.
- Jeder Mann führt eine geordnete Liste mit allen Frauen.
- Eine Map (später) *couples* mit allen Paaren.
- Eine Liste *freeList* mit allen freien Männern.

Zu Beginn sind alle Männer in der *freeList* und *couples* ist leer.

Stabile Heirat: Algorithmus (Pseudocode)

```
Map findStableMarriage() {
  while (!freeList.isEmpty()) {
    man = freeList.getFirst();
    woman = man.topPick();   // die Traumfrau
    if (woman.likes(man)) {  // true falls auf Liste
      husband = couples.getHusband(woman);
      if (husband != null) freeList.add(husband);
      couples.marry(woman, man);
      freeList.remove(man);
      // Lösche alle mit tieferer Präferenz
      woman.removeAllBelow(man);
    }
    man.remove(woman);
  }
  return couples;
}
```

Man kann zeigen, dass immer eine stabile Lösung gefunden werden kann.

Das Resultat ist im Allgemeinen anders, wenn die Rollen von Frauen und Männern vertauscht werden.

Das Resultat ist Mann-optimal, d.h. unter allen möglichen stabilen Paaren bekommt der Mann damit seine Lieblingspartnerin.

Stabile Heirat: Beispiel von vorher

fritz proposes to vreni
  she accepts
  but she still prefers: sepp , roli ,
kurt proposes to vreni
   she refuses
max proposes to moni
  she accepts
  but she still prefers: roli , fritz , sepp , kurt ,
roli proposes to susi
  she accepts
  but she still prefers: sepp , max , kurt ,
sepp proposes to vreni
  she accepts (dumping fritz)
  but she still prefers: nobody
kurt proposes to jenny
  she accepts
  but she still prefers: sepp , roli , max ,
fritz proposes to susi
   she refuses
fritz proposes to moni
  she accepts (dumping max)
  but she still prefers: roli ,
max proposes to susi
  she accepts (dumping roli)
  but she still prefers: sepp ,
roli proposes to vreni
   she refuses
roli proposes to moni
  she accepts (dumping fritz)
  but she still prefers: nobody
fritz proposes to jenny
   she refuses
fritz proposes to angi
  she accepts
  but she still prefers: sepp , kurt , max , roli

Fritz:	**Vreni , Susi , Moni , Jenny , Angi**
Kurt:	**Vreni , Jenny , Moni , Angi , Susi**
Max:	**Moni , Susi , Jenny , Vreni , Angi**
Roli:	**Susi , Vreni , Moni , Angi , Jenny**
Sepp:	**Vreni , Moni , Susi , Jenny , Angi**

Angi:	**Sepp , Kurt , Max , Roli , Fritz**
Jenny:	**Sepp , Roli , Max , Kurt , Fritz**
Moni:	**Roli , Fritz , Sepp , Kurt , Max**
Susi:	**Sepp , Max , Kurt , Roli , Fritz**
Vreni:	**Sepp , Roli , Fritz , Max , Kurt**

Resultat:

Fritz	**-**	**Angi**
Kurt	**-**	**Jenny**
Max	**-**	**Susi**
Roli	**-**	**Moni**
Sepp	**-**	**Vreni**

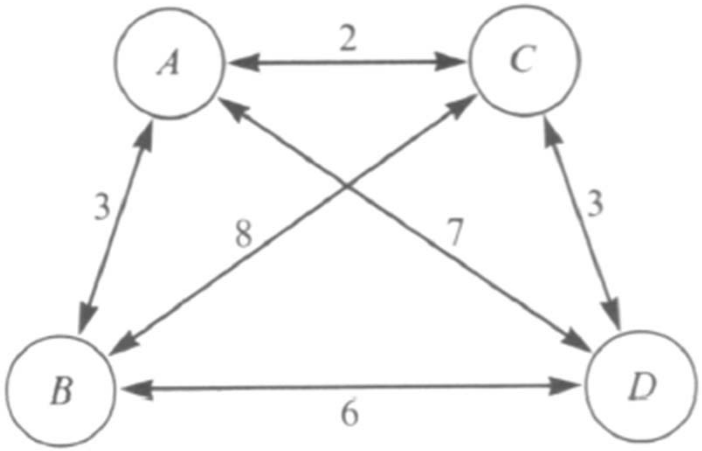
Optimierungsprobleme

- Transportwesen
  - optimale Beladung eines LKWs (Gewicht, Volumen, Wert der Ware)
  - Travelling Salesman
  - Standorte von Verteilzentren
- Schule:
  - Stundenplan (Pausenminimierung & Raumbelegung)
- Spiele:
  - bester Zug im Schach
- Gemeinsam:
  - es müssen sehr viele Möglichkeiten ausprobiert werden
  - es kann eine Art "Güte" der Teillösung bestimmt werden

Branch and Bound

- Branch and bound ist eine allgemeine Methode um Optimierungsprobleme zu lösen.
- Wir suchen das Minimum einer Zielfunktion f(S), wobei f über einer Menge von Lösungskandidaten definiert ist.
- 2 Schritte:
  1. Gegeben eine Menge von Kandidaten S. Finde n>1 Teilmengen S_i, sodass deren Vereinigung S ergibt. Dieser Schritt wird *branching* genannt. Dabei gilt min (f(S)) = min (min(f(S_1)), min(f(S_2)), ...) Daraus entsteht ein Baum.
  2. Berechne die untere und obere Schranke für f für die Teilmengen. Dieser Schritt wird *bounding* genannt.
- Die Grundidee ist die: Wenn die untere Schranke eines Knotens A grösser als die obere Schranke eines andern Knotens wird, dann kann der Knoten A abgeschnitten werden. Dieser Schritt heisst *pruning*.
  Manchmal verzichtet man auf die obere Schranke und nimmt einfach den erfolgversprechendsten Weg.

Branch and Bound am Beispiel Travelling Salesman



P_0 sei das Grundproblem mit folgender Distanztabelle

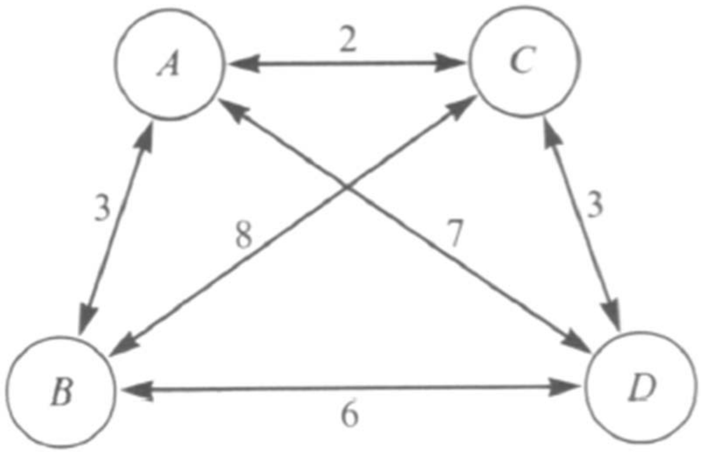
P_0	A	B	C	D
A	-	3	2	7
B	3	-	8	6
C	2	8	-	3
D	7	6	3	-

Zielfunktion = Weglänge
Als untere Schranke für die Zielfunktion kann die Summe der Längen der kürzesten Ankunftswege, bzw. die Summe der Längen der kürzesten Abgangswege zu/von allen Städten genommen werden. Das Maximum der beiden ist auch eine (engere) untere Schranke.
$d_{min}(P_0)=max((2+3+2+3), (2+3+2+3))=max(10,10)=10.$

Entsprechend für die obere Schranke:
$d_{max}(P_0)=min((7+8+8+7), (7+8+8+7))=min(30,30)=30.$

Aus: http://www.inf.hs-zigr.de/~wagenkn/TI/Komplexitaet/ReferateSS00/belitz.pdf

Branch and Bound am Beispiel
Travelling Salesman



•Die Stadt A kann nach B, C oder D verlassen werden.
•Daraus ergeben sich drei Unterprobleme.
•Die Wegmöglichkeiten, die nicht in Betracht kommen, werden nicht betrachtet.
•Die Wege B-A, C-B und D-B sind nicht möglich (Unterproblem P_1).

P_1 Unterproblem von A nach B

P_0	A	B	C	D
A	-	3	-	-
B	-	-	8	6
C	2	-	-	3
D	7	-	3	-

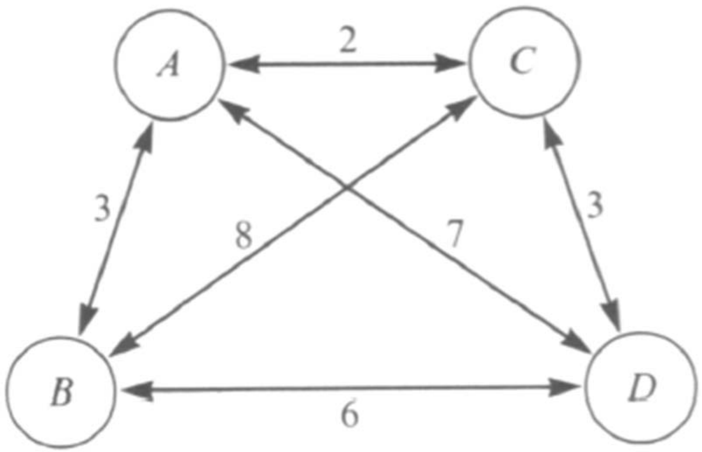
$d_{min}(P_1)$ = max ((2+3+3+3), (3+6+2+3))= max (11, 14) = 14
$d_{max}(P_1)$ = min ((7+3+8+6), (3+8+3+7))= min (24, 21) = 21

P_2 Unterproblem von A nach C

P_0	A	B	C	D
A	-	-	2	-
B	3	-	-	6
C	-	8	-	3
D	7	6	-	-

$d_{min}(P_2)$ = max ((3+6+2+3), (2+3+3+6))= max (14, 14) = 14
$d_{max}(P_2)$ = min ((7+8+2+6), (2+6+8+7))= min (23, 23) = 23

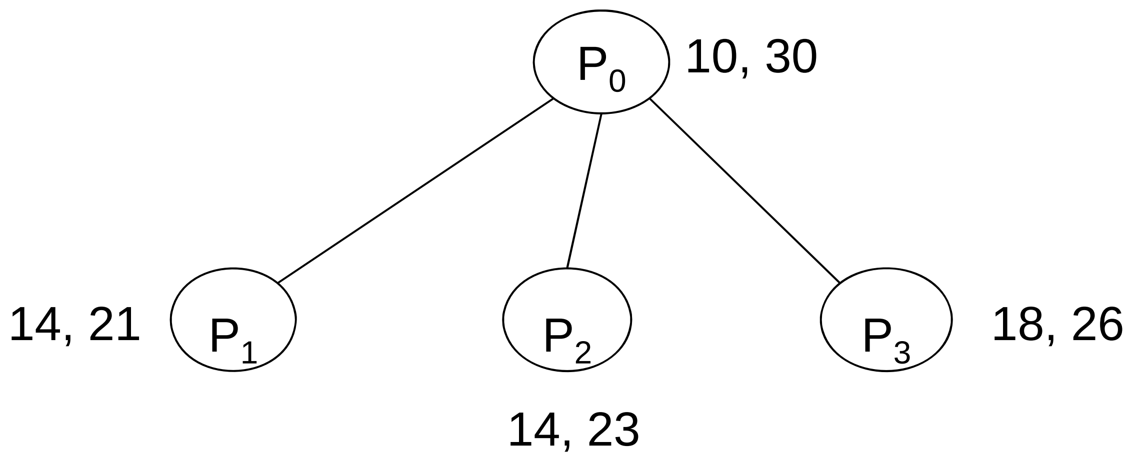
Branch and Bound am Beispiel Travelling Salesman



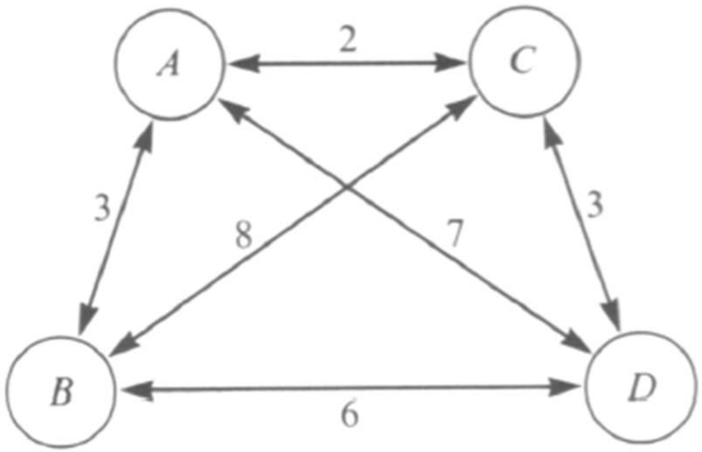
P_3 Unterproblem von A nach D

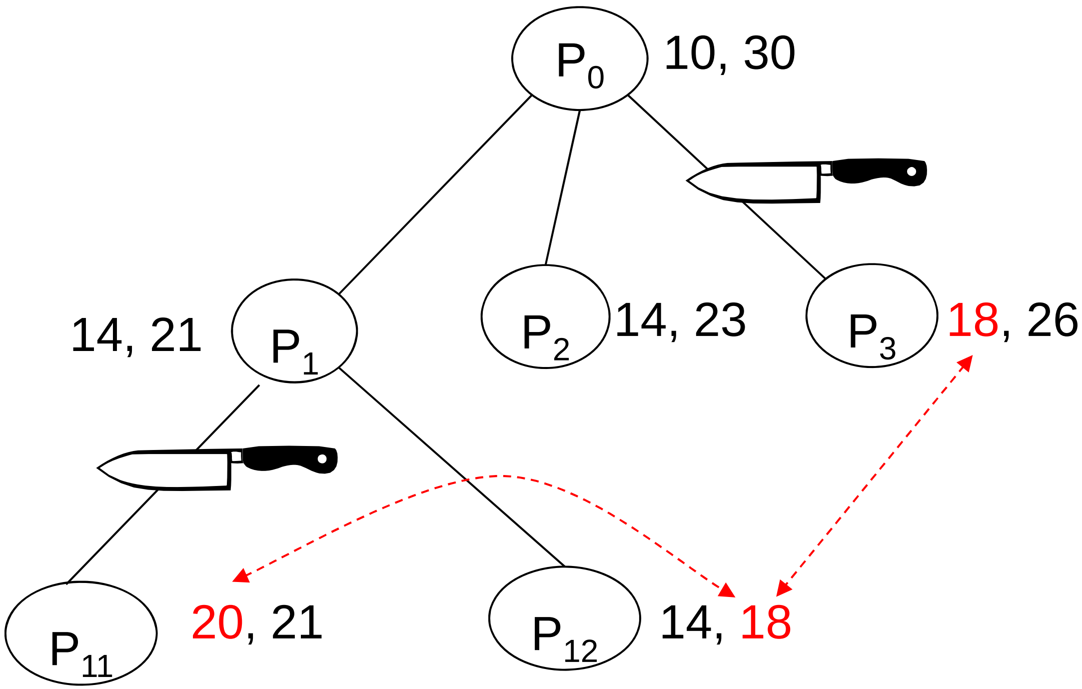
P_0	A	B	C	D
A	-	-	-	7
B	3	-	8	-
C	2	8	-	-
D	-	6	3	-

$d_{min}(P_3) = \max(18, 15) = 18$
$d_{max}(P_3) = \min(29, 26) = 26$



Branch and Bound am Beispiel
Travelling Salesman





P_{11} Unterproblem von A über B nach C

P_0	A	B	C	D
A	-	3	-	-
B	-	-	8	-
C	2	-	-	3
D	7	-	-	-

$\mathbf{d_{min}(P_{11}) = max\ (16, 20) = 20}$
$\mathbf{d_{max}(P_{11}) = min\ (21, 21) = 21}$

P_{12} Unterproblem von A über B nach D

P_0	A	B	C	D
A	-	3	-	-
B	-	-	-	6
C	2	-	-	-
D	7	-	3	-

$\mathbf{d_{min}(P_{12}) = max\ (14, 14) = 14}$
$\mathbf{d_{max}(P_{12}) = min\ (19, 18) = 18}$

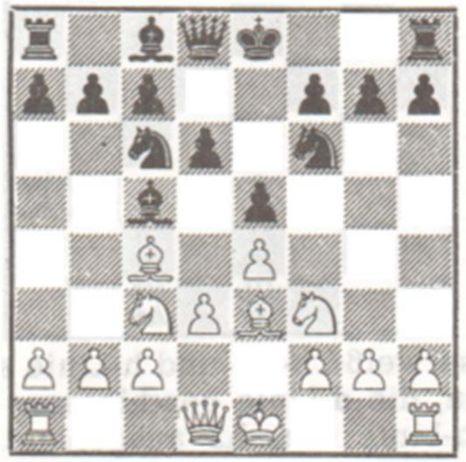
Betrachtungen Branch and Bound

- Mittels Branch and Bound lässt sich das Problem der kombinatorischen Explosion eindämmen

- Branch and Bound Verfahren setzen eine mit vernünftigem Aufwand berechenbare Schranke b(v) der Zielfunktion voraus.

- Problem der Bestimmung einer "guten" b(v) Funktion
  – genau → Berechnung ist zu teuer
  – ungenau (zu gross) → es können keine/wenige Teilbäume abgeschnitten werden → kombinatorische Explosion

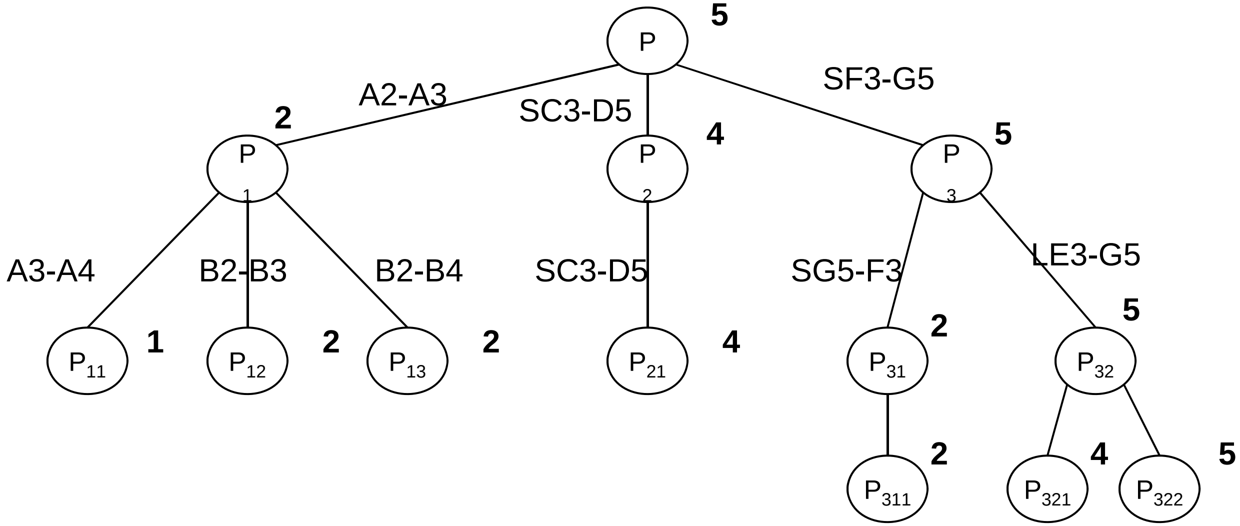
Minimax Algorithmus, Beispiel Schach



- Entscheidungsbaum
  - alle möglichen Züge
- Berechne für jede Position einen b(v) Wert
- Auch als Bewertungsfunktion bezeichnet, Score
- Figuren Werte:
  - König 100; Dame 9; Turm 5; Springer/Läufer 3.5; Bauer 1
- Position:
  - Beherrschung des Zentrums
  - Schutz des Königs und der restlichen Figuren
  - Bedrohung des gegnerischen Königs&Figuren
  - ...
- Wichtig: b(v) ist obere Schranke, d.h. zu optimistisch

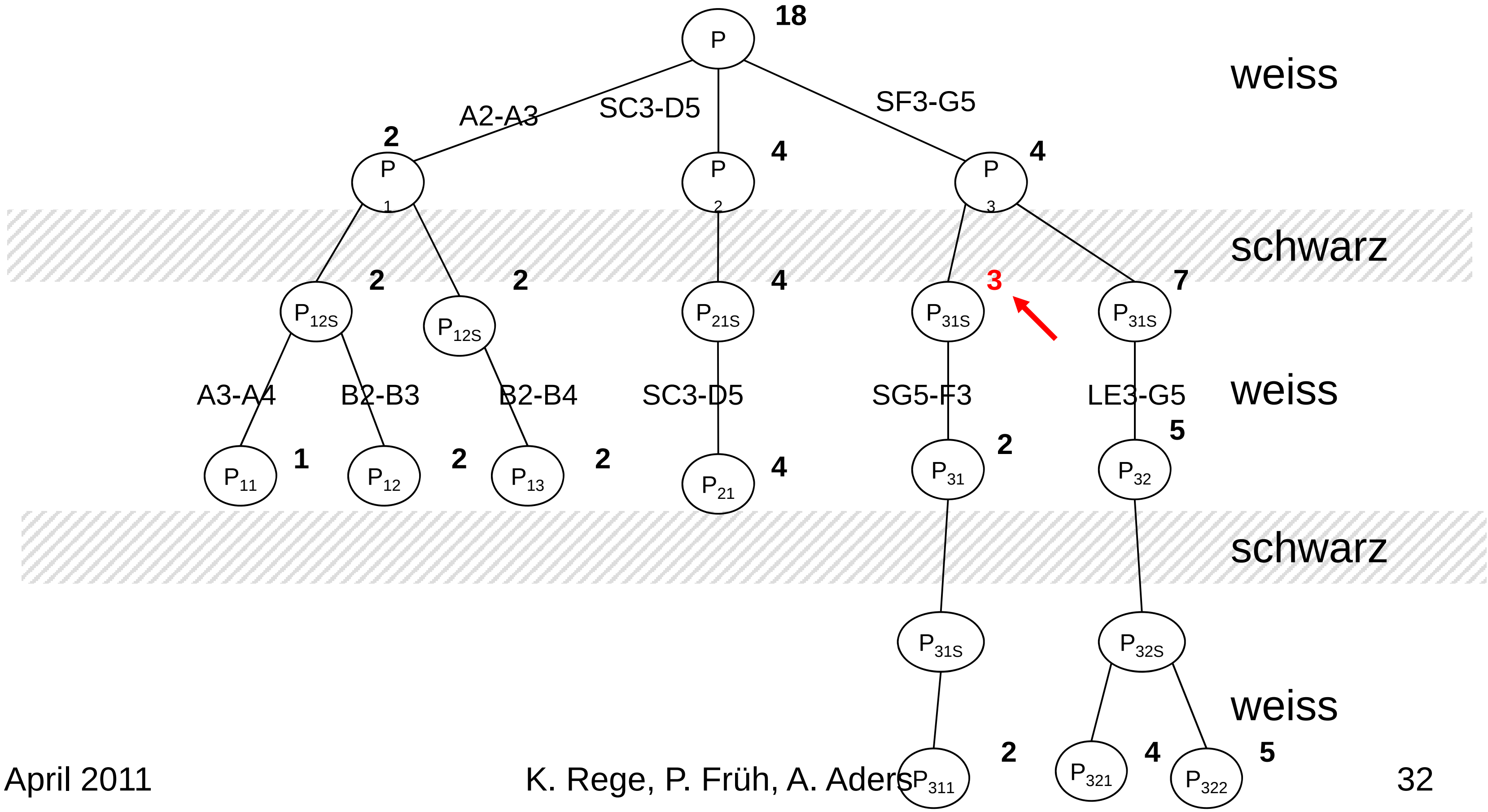
Minimax Algorithmus, Beispiel Schach

- Jeder Zug führt zu einer anderen Position, die bewertet werden kann.
- Aber: es kommen beide Spieler abwechselnd an die Reihe



Minimax Algorithmus, Beispiel Schach Gegenzug

- Schwarz wird (Annahme: Schwarz macht den besten Zug) einen Gegenzug machen, der das *"eigene"* b(v) möglichst minimiert



Minimax Algorithmus, Alphabeta-Pruning

- Der Algorithmus wechselt zwischen Maximieren und Minimieren des b(v) Werts ab.

- Falls mit Pruning gearbeitet wird: Alphabeta-Pruning
  - alpha: Grenze bei der Maximierungsebene
  - beta: Grenze bei der Minimierungsebene

- Bei jeder Stellung ca. 8-10 Züge möglich
  - Anzahl Stellungen 10^n
  - für jeden weiteren Zug-Gegenzug muss jeweils 100-mal länger gerechnet werden
  - es können nur eine begrenzte Anzahl Züge vorausberechnet werden: 8 bis 9

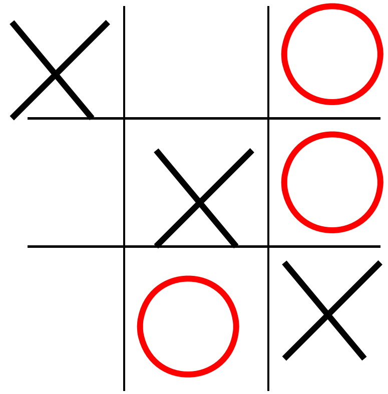
Der Horizont Effekt

- Die Berechnung muss nach n Zügen abgebrochen werden.
- Dies wird als der **Horizont** bezeichnet

- Problem:
  - gleich hinter dem Horizont kann sich die gefundene Lösung als schlecht erweisen.

- Lösung:
  - die ausgewählte Lösung (und nur diese) wird noch ein paar Stufen weiter ausgewertet.

Tic-Tac-Toe



- Spiel, bei dem der ganze Entscheidungs-baum berechnet werden kann
- Ziel: zuerst eine 3 X oder O in einer Zeile, Spalte oder Diagonale

Besonderes:

- es gibt keine Gewinnstrategie
→ wenn keiner einen Fehler macht, dann immer unentschieden
- Sämtliche möglichen Züge können vorausberechnet werden
- Es sind 549'946 jedoch rekursive Aufrufe nötig, um den ersten Zug zu bestimmen

Tic-Tac-Toe: Datenstrukturen

```
static int n = 3;
static int diagN = 2*n -1;

static int[][] spalte = new int[2][n];
static int[][] reihe = new int[2][n];
static int[][] diagLinks = new int[2][diagN];
static int[][] diagRechts= new int[2][diagN];
static int[][] board = {{-1,-1,-1},{-1,-1,-1},{-1,-1,-1}};

static final int COMPUTER_WIN = 2;
static final int UNCLEAR = 1;
static final int HUMAN_WIN = 0;

static final int COMPUTER = 1;
static final int HUMAN = 0;

static final int FREE = -1;
static int bestX, bestY;
```

Tic-Tac-Toe: Hilfsmethoden

```
public static void setze(int side, int x, int y) {
   spalte[side][x]++;
   reihe[side][y]++;
   diagLinks[side][(x + y) % diagN]++;
   diagRechts[side][(diagN + x - y) % diagN]++;
   board[x][y] = side;
 }

 public static void loesche(int side, int x, int y) {
   spalte[side][x]--;
   reihe[side][y]--;
   diagLinks[side][(x + y) % diagN]--;
   diagRechts[side][(diagN + x - y) % diagN]--;
   board[x][y] = FREE;
 }
```

```
public static int win() {
    for (int x = 0; x < n; x++ ) {
      if (spalte[COMPUTER][x] == n) return COMPUTER_WIN;
      if (spalte[HUMAN][x] == n) return HUMAN_WIN;
    }
    for (int y = 0; y < n; y++) {
      if (reihe[COMPUTER][y] == n) return COMPUTER_WIN;
      if (reihe[HUMAN][y] == n) return HUMAN_WIN;
    }
    if (diagLinks[COMPUTER][n-1] == n) return COMPUTER_WIN;
    if (diagLinks[HUMAN][n-1] == n) return HUMAN_WIN;
    if (diagRechts[COMPUTER][0] == n) return COMPUTER_WIN;
    if (diagRechts[HUMAN][0] == n) return HUMAN_WIN;

    return UNCLEAR;
  }
```

Tic-Tac-Toe: Algorithmus

```
public static int chooseMove(int side, int draw) {
  int score,bX=-1,bY=-1;
  if (draw == n*n) return UNCLEAR;
  else {
    score = win();
    if (score != UNCLEAR) return score;
    score = (side == COMPUTER)?HUMAN_WIN:COMPUTER_WIN;
    for (int x = 0; x < 3; x++ ) {
      for (int y = 0; y < 3; y++) {
        if (board[x][y] == FREE) {
          setze(side,x,y);
          int val = chooseMove((side+1)%2,draw+1);
          if (draw == 0) printBoard("board " + val);
          loesche(side,x,y);
          if ((side == COMPUTER) && (val > score)) {
              bX = x; bY = y; score = val;
          }
          else if ((side == HUMAN) && (val < score)) {
              bX = x; bY = y; score = val;
          }
        }
      }
    }
  }
  bestX = bX, bestY = bY;
  return score;
```

gewonnen

gültige Position

maximiere

minimiere

Zusammenfassung

- Versuch und Irrtum
- Entscheidungsbaum
- Backtracking
- Beispiele
  - Labyrinth
  - Springer
  - 8 Damen
- Kombinatorische Explosion
- Zuordnungsprobleme: Stabile Heirat
- Optimierungsprobleme
  - Branch and Bound
  - Traveling Salesman
- Minimax Problem
- alphabeta Pruning
- Tic-Tac-Toe