# 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
- Sie wissen was eine Zielfunktion ist
- Sie wissen wie Branch&Bound Verfahren funktionieren
- Sie wissen was Pruning bedeutet

# Versuch und Irrtum

- Eigentlich bessere Bezeichnung wäre: Versuch und Bewertung
- Die älteste Lösungsstrategie überhaupt
- Natur:
  - Mutation = Versuch
  - Selektion = Bewertung
- Resultat nach 4 Milliarde Jahren: der Mensch

- Schlussfolgerungen
  - Trial & Error ist ziemlich rechen- und zeitintensiv
  - Nicht unbedingt beste Lösung als Resultat
  - Zwei mögliche Resultate bei Trial & Error Ansatz
    - *beste Lösung*
    - *akzeptable Lösung (unter den gegebenen Rahmenbedingungen)*

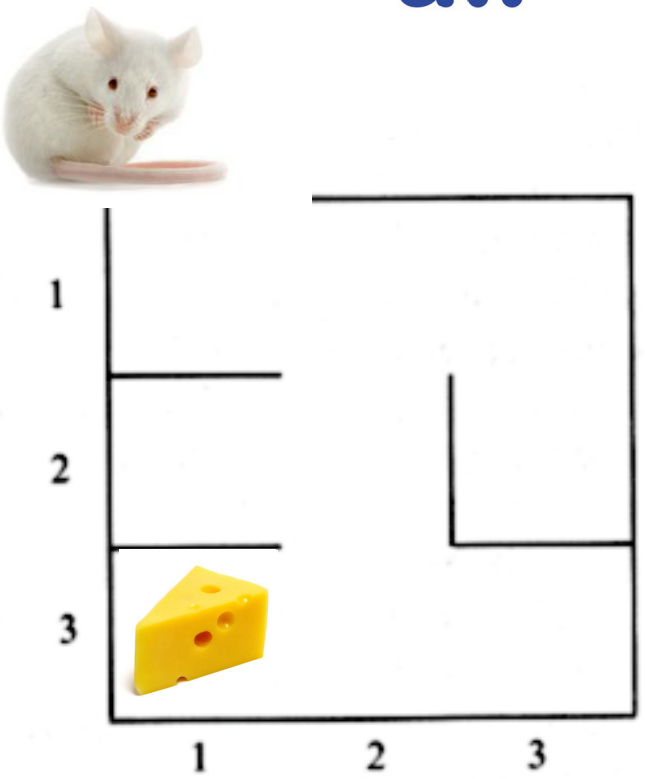
# Labyrinth

# Beispiel: Labyrinth

- Probleme:
  - Maus sucht Käse
  - Katze sucht Maus
  - Maus sucht Ausgang (auf dem kürzesten Weg)

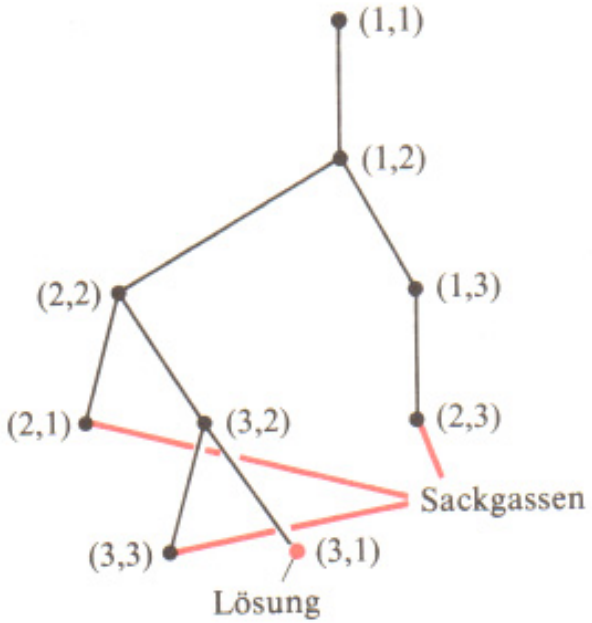
- Lösungsalgorithmus

  gehe einen Weg entlang bis
  Verzweigung
    gehe einem der möglichen Wege entlang
      *i) bis am Ziel*
      *ii) oder nicht mehr weiter (Sackgasse)*
        gehe zur Verzweigung zurück
        versuche noch nicht probierten Weg



# Entscheidungsbaum, Backtracking

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




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

- Bei Sackgasse werden ein 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: Bsp. 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
      nehme Erweiterung zurück
   nein ->
      keine Lösung möglich, Abbruch
```

- Es müssen systematisch alle möglichen Erweiterungen durchprobiert werden.

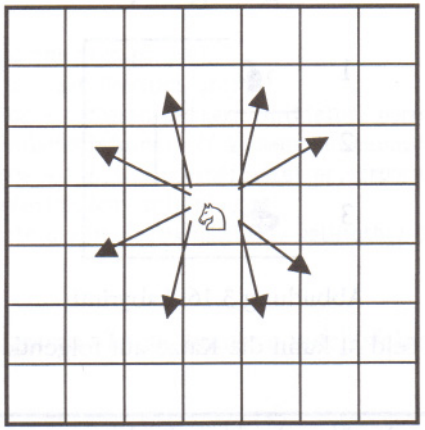
# rekursive Suche im Labyrinth: Pseudocode

```
boolean search (Node currentNode) {
  mark currentNode;
  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;
}
```

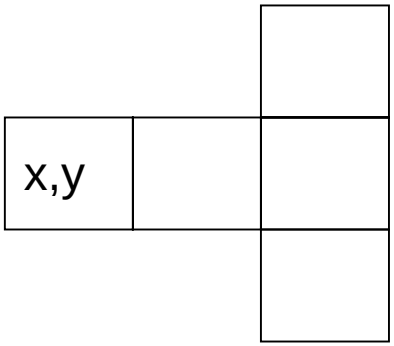
# Springerproblem

# Springerproblem

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



- Bewegung des Springers
  - zwei Felder in beliebige Richtung und ein Feld in dazu senkrechter Richtung




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

# 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)

# Der Algorithmus

```
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;
}
```

Position erlaubt

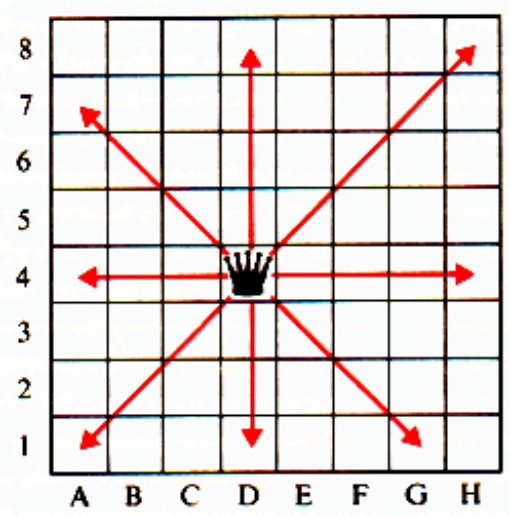
Lösung gefunden

neue Position

# 8 Damenproblem

# Das 8 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



# Datenstrukturen und Methoden

- `int[] dameInDerSpalte = new int[n];`
- `static boolean[] reihe = new boolean[n];`
- `int diagN = 2*n -1;`
- `boolean[] diagLinks = new boolean[diagN];`

| | | |
|---|---|---|
| 1,5 | | |
| | 2,4 | |
| | | 3,3 |

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

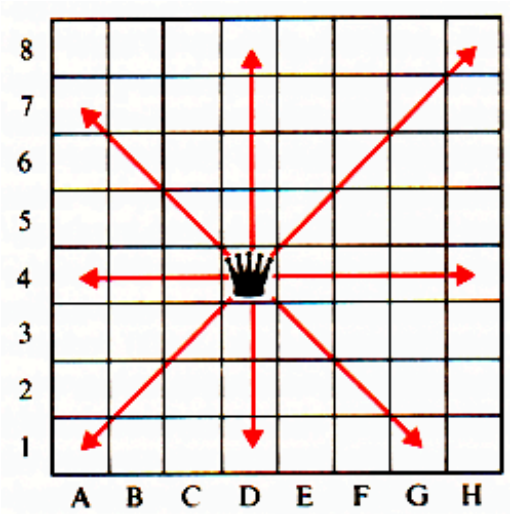
- `boolean[] diagRechts = new boolean[diagN];`

| | | |
|---|---|---|
| | | 3,5 |
| | 2,4 | |
| 1,3 | | |

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

Spalte

Reihe



# Zwei Hilfsmethoden

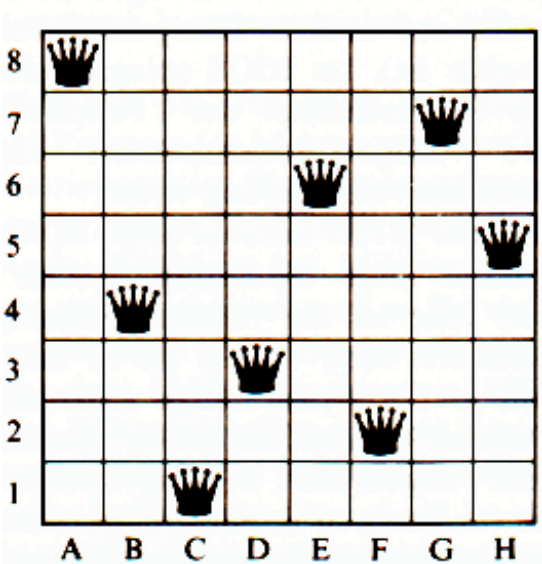
```
// testet ob Position möglich ist
// Wert in 3 Arrays true -> Position besetzt

public static boolean gueltigeDamePosition(int x, int y) {
  return !(reihe[y] || diagLinks[(x + y) % diagN]
            || diagRechts[(diagN + x - y) % diagN]);
}


// setzt/löscht die Dame von der Position
public static void setzeDame(int x, int y, boolean val) {
  reihe[y] = val;
  diagLinks[(x + y) % diagN] = val;
  diagRechts[(diagN + x - y) % diagN] = val;
  dameInDerSpalte[x] = (val)?y:-1;
}
```

# Der Algorithmus

```
public static 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;
  }
}
```



# Rucksackproblem

# Das Rucksackproblem

- **Das Rucksackproblem**
  Ein Dieb, der eine Wohnung ausraubt, findet K verschiedene Gegenstände unterschiedlicher Grösse und unterschiedlichen Werts, hat aber nur einen 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 auswählen sollte, so dass der Gesamtwert der vom ihm geraubten Gegenstände maximal wird.

# Gegenstände in der Wohnung

| | | | | | |
|---|---|---|---|---|---|
| **Volumen:** | **1l** | **2l** | **7l** | **8l** | **9l** |
| **Wert:** | **2'000** | **3'000** | **10'000** | **11'000** | **17'000** |

- Beispiel: Der Rucksack besitzt ein Fassungsvermögen von 17l, in der Wohnung befinden sich diese 5 Gegenstände von unterschiedlicher Grösse und den angegebenen Werten.
- Die Bezeichnungen der Gegenstände werden im Programm in Indizes umgewandelt: 0 bis 4

# Das Rucksackproblem - Pseudocode

- systematisch alle möglichen Varianten ausprobieren: Trial & Error

```
void teste (Gegenstand k) {
   teste k + 1 // ohne Gegenstand k
   fall Gegenstand k noch Platz
        füge Element k zu der Menge hinzu
        falls neues Maximum speichere das
        teste k+1 // mit Gegenstand k
        nehme Element k aus der Menge weg
   }
}
```

# Rucksackproblem - Trial&Error Algorithmus

```
double[] volumen= {1,2,7,8,9};
double[] wert = {2, 3, 10, 11, 17};
Set<Integer> maxRucksack;
final double MAXV = 18;
double maxW = 0;

    static   public void test(Set<Integer> rucksack, int k, double aktW, double aktV) {
      if (k < volumen.length) {
         test(rucksack, k + 1, aktW, aktV);
         double newV = aktV + volumen[k];
         if (newV <= MAXV) {
            rucksack.add(k);
            double newW = aktW + wert[k];
            if (newW > maxW) {
               maxRucksack = new HashSet<Integer>(rucksack);
               maxW = newW;
            }
            test(rucksack, k + 1, newW, newV);
            rucksack.remove(k);
         }
      }
    }
```

teste zuerst ohne Gegenstand k

teste ob ok

füge hinzu

neues Maximum?

nehme weg

# Aufwandsbetrachtung: Statistik

Frage: auf wie viele Arten kann ich 1,2,3,..,K unterscheidbare Gegenstände auswählen, jeden Gegenstand nur einmal

Statistiker sprechen von Kugeln und Fächer :0, 1, 2, 3,…,K

- #1 Kugel: <0>: 1
- #2 Kugeln: <0>,<1>,<1,0>: 3
- #3 Kugeln: <0>,<1>,<1,0>, <2> <2,0>,<2,1>,<2, 1,0>: 7
- #4 Kugeln: <"3 Kugeln">, <4>, <4,"3 Kugeln">: 15
- ….
- #K Kugeln:  2 * #(K-1 Kugeln) + 1 -> 2*2*2*2,...

Aufwand: $O(2^n)$

# Anwendungen des Rucksackproblems

Transportunternehmer:

- Optimale Beladung eines Lastwagens bei gegebenen Maximalgewicht und unterschiedlichen Speditionsgebühren (Wert).

Reederei-Besitzer:

- Optimale Beladung eines Schiffes mit unterschiedlichen Containern (Wert), bei gegebenen Volumen (Gewicht).

# Erschöpfende Suche

- Das Rucksackproblem ist ein Beispiel von einer Klasse von algorithmischen Problemen, bei denen keine bessere Lösung bekannt ist, als Ausprobieren sämtlicher Möglichkeiten: Versuch & Irrtum

- Durchsuche aller Möglichkeiten -> **erschöpfende Suche** (*exhaustive search*)

- Der Aufwand bei diesen Algorithmen ist meist $O(2^n)$ oder $O(n!)$, entsprechenden der Anzahl möglichen Kombinationen (oder Permutationen).

- Frage: gibt es eine Ordnung/Klassifikation in den Komplexität von Problemen?

# Algorithmus & Probleme

# Optimierungsprobleme

- Transportwesen
  - optimale Beladung eines LKWs (Gewicht, Wert der Ware)
- 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

# Die kombinatorische Explosion

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

- z.B. beim Springer max. 8 mögliche Positionen pro Zug ->
- Abschätzung: 8*8*8 … *8  bei 64 Felder -> $8^{64} = 6.3 * 10^{57}$
  - hier wird aber vernachlässigt, dass die Anzahl möglicher Züge mit der Zeit abnimmt.

- Die Anzahl der möglichen Fälle (Kombinationen) wächst $k^n$ oder n!

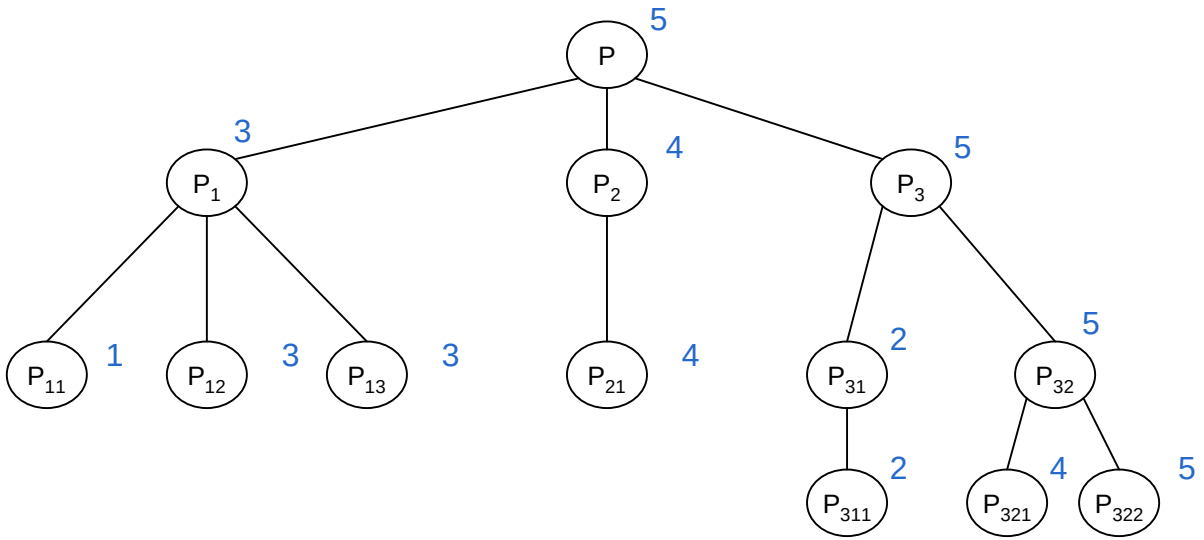
- schon für relativ kleine Werte von n (n ~ 10-100) dauert die Berechnung meist zu lange.
- Alter des Universum: $5*10^{17}$ sek
  Masse der Sonne $2*10^{33}$ g
  Anzahl Atome im Weltall: $10^{77}$

# Suchverfahren mit Zielfunktion

# Die Zielfunktion

- Umgehen der kombinatorischen Explosion:
  - man wählt nur die Lösung aus, die zum Ziel führt
  - man berechnet zu jedem Knoten im Entscheidungsbaum den *Zielwert*, den man über diesen Knoten erreichen kann

- Diese Funktion wird als Zielfunktion f(v) bezeichnet



# Algorithmus mit Zielfunktion

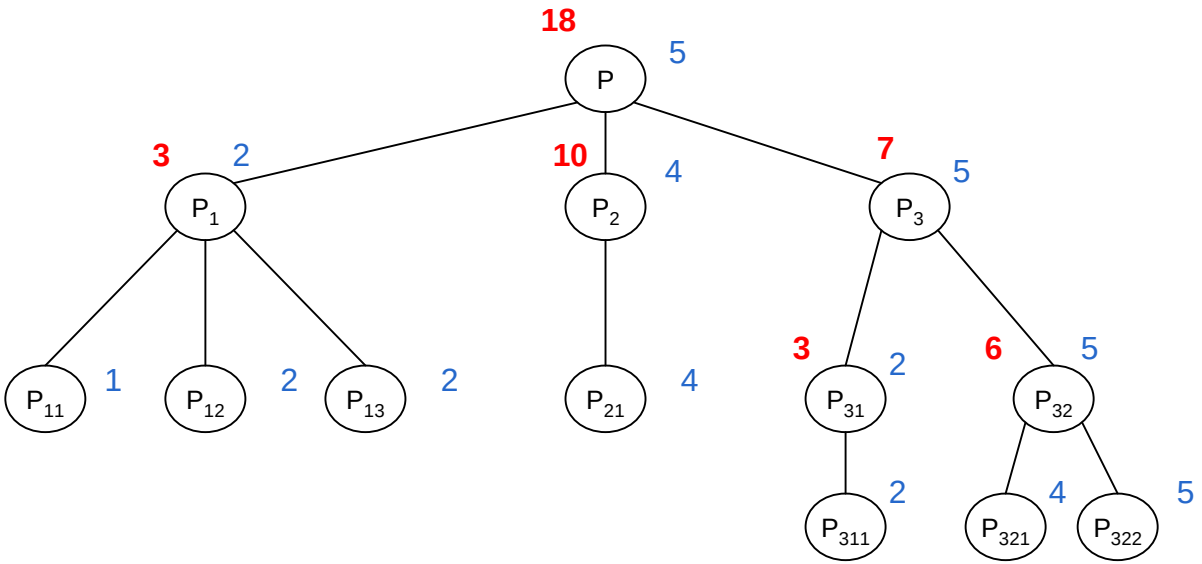
- für jeden Knoten
  - berechne zu jedem Nachfolgeknoten im Entscheidungsbaum die Zielfunktion f(v) , Kostenfunktion c(v) = 1/f(v)
  - gehe der Kante entlang (wähle die Teillösung aus), die zum Knoten mit dem höchsten Zielfunktionswert führt

- Problem gelöst
  - keine kombinatorische Explosion
  - sehr effizienter Algorithmus : ~ Log(n)

- Aber: zur Berechnung der  Zielfunktion muss das Problem meist schon gelöst sein, d.h. z.B. der Entscheidungsbaum unterhalb des Knotens vollständig durchlaufen sein -> wir haben nichts gewonnen.

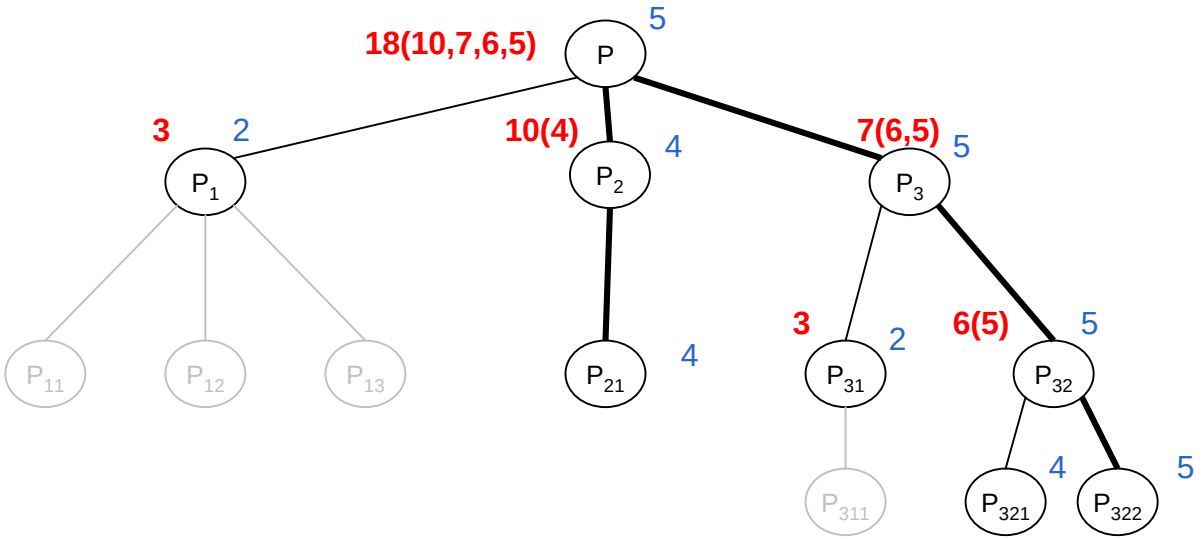
# Geschätzte obere Schranke

- es wird nicht die exakte Zielfunktion selber, sondern eine einfacher zu bestimme Funktion
- "Score der Lösung", Fittness Function, Kostenfunktion
- Es wird eine Funktion bestimmt, die immer bessere Werte liefert als die exakte Zielfunktion: ***obere Schranke*** b(v) >= f(v)
- Bound Funktion: Schranke



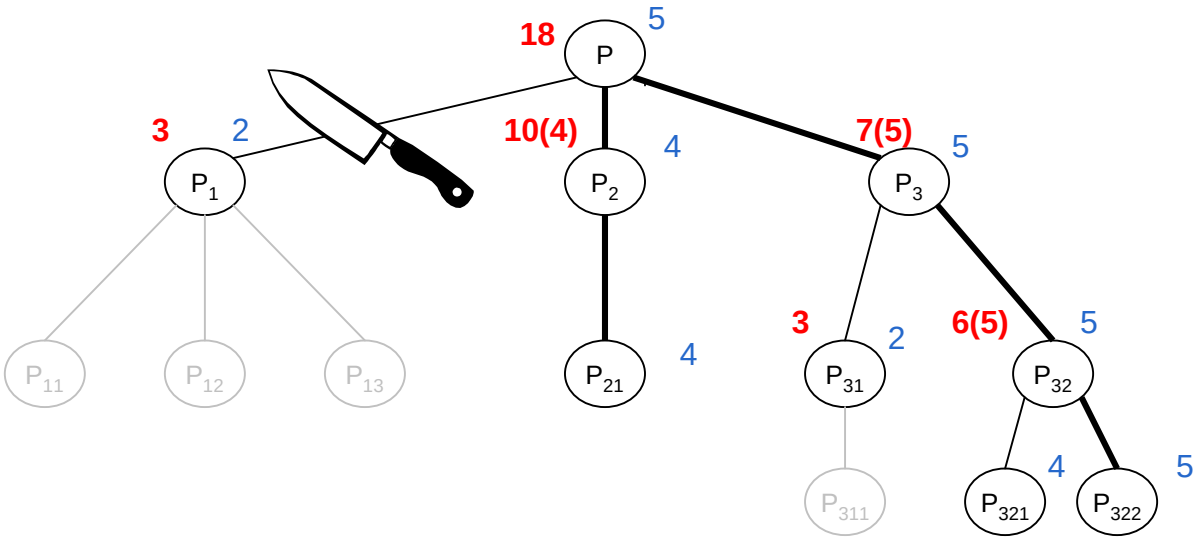
# Algorithmus: Bestfirst Search

- gehe dem Pfad mit dem höchsten Bound-Wert **zuerst** entlang
  - Bestfirst Search
- **korrigiere den b(v)-Wert des betrachteten Knotens** anhand der Bound-Werte der darunter liegenden Konten bzw. des erreichten Ergebnisses; Bound Wert ist das Maximum der Bound Werte der direkten Nachfolger.
- Kombination aus Breitensuche und Tiefensuche

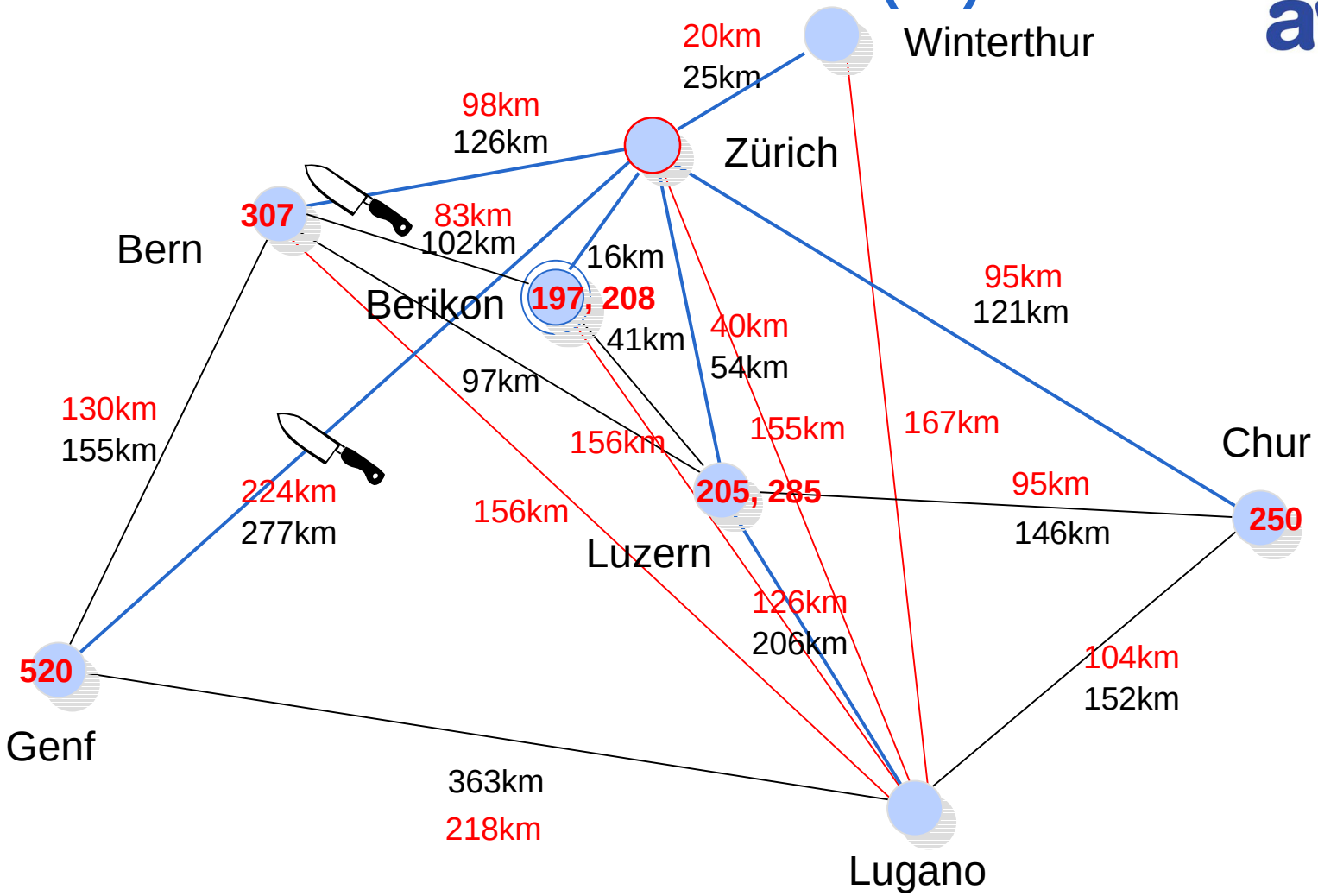


# Abschneiden (Cutoff, Pruning)

- Falls eine bereits gefundene Lösung besser ist, als die mittels der oberen Schranke b(v) geschätzte bestmögliche Lösung über einen Knoten, dann muss dieser Teilbaum nicht mehr betrachtet werden.
- Das "Abschneiden" eines Astes im Entscheidungsbaum wird als **Pruning** bezeichnet



# Bound Funktion für Routensuche (A*)



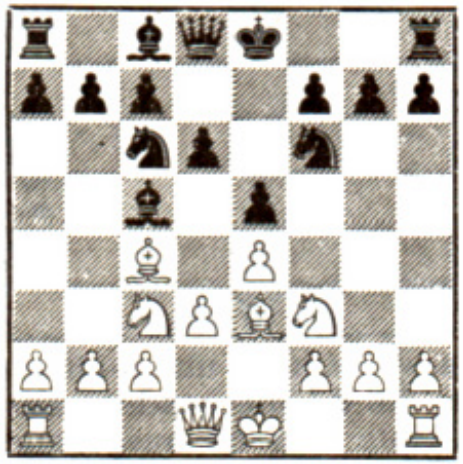

- Hier wird eine zu optimistische untere Schranke berechnet
- Die kürzeste Strecke kann nicht besser sein als die Luftlinie

http://www.luftlinie.org

# Betrachtungen Branch&Bound

- mittels Branch&Bound lässt sich das Problem der kombinatorischen Explosion eindämmen
- Branch&Bound Verfahren setzen eine mit vernünftigem Aufwand berechenbare b(v) Funktion 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

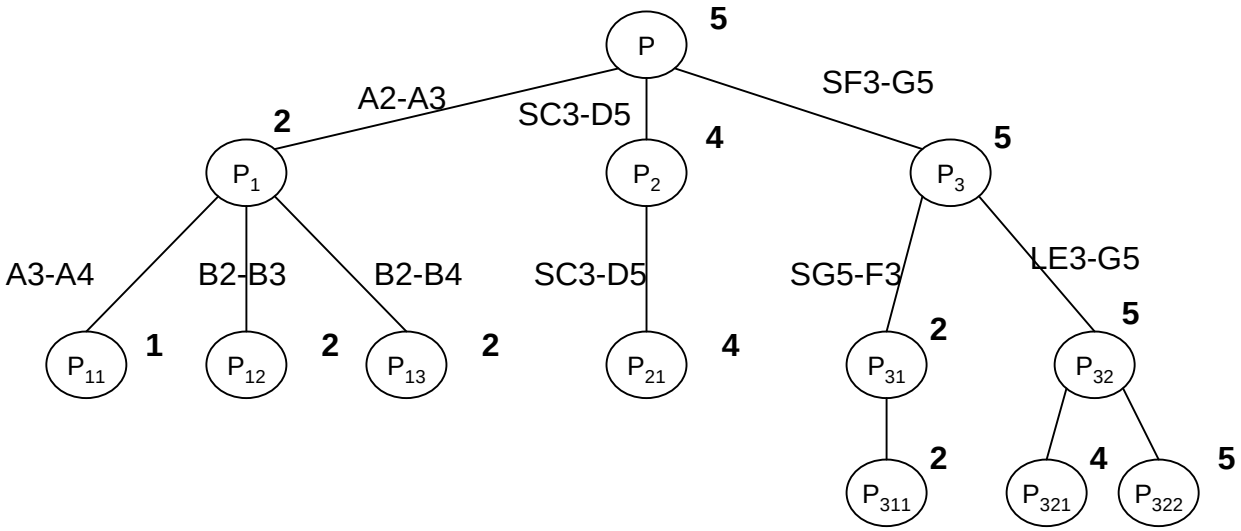
# Beispiel Schach



- 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 der gegnerischen Königs&Figuren
  - ...
- Wichtig: b(v) ist obere Schranke, d.h. zu optimistisch

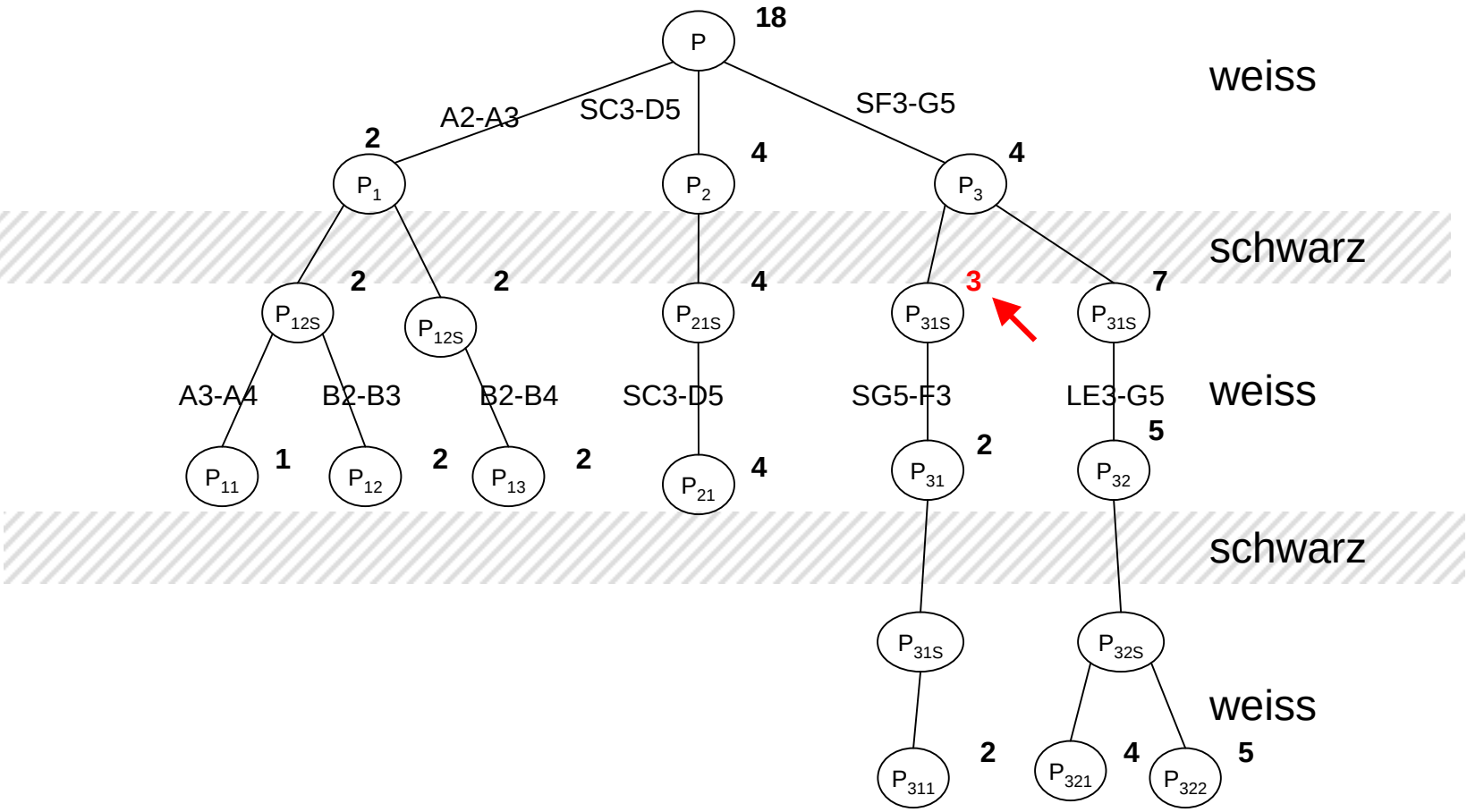
# Beispiel Schach

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



# Der 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 Optimierungsebene
  - 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.

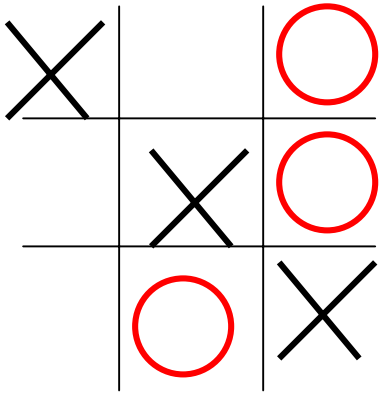
- Lösung:
  - die ausgewählte Lösung (und nur diese) wird noch ein paar Stufen weiter ausgewertet.

# Tic-Tac-Toe

# Tic-Tac-Toe

School of Engineering

- Spiel, bei dem der ganze Entscheidungs-baum berechnet werden kann



- Ziel: zuerst eine 3 X oder O in einer Reihe, Spalte oder Diagonalen

Besonderes:

- es gibt keine Gewinnstrategie -> wenn keiner einen Fehler macht, dann immer unentschieden

- Sämtliche möglichen Züge können vorausberechnet werden

- Es sind jedoch 549'946 rekursive Anrufe nötig, um den ersten Zug zu bestimmen

# Datenstrukturen

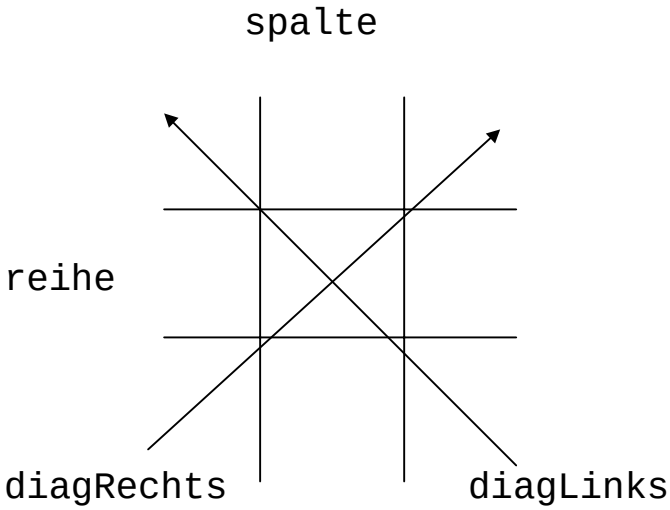
```
static int n = 3;
static int diagN = 2*n -1;

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;
```



# 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;
  }
```

# Algorithmus

```
public static int tryMove(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 = tryMove((side+1)%2,draw+1);
          if ((side == COMPUTER) && (val > score)) {
              bX = x; bY = y; score = val;
          }
          else if ((side == HUMAN) && (val < score)) {
              bX = x; bY = y; score = val;
          }
          loesche(side,x,y);
        }
      }
    }
  }
  bestX = bX, bestY = bY;
  return score;
}
```

gewonnen

gültige Position

maximiere

minimiere

# Zusammenfassung

- Versuch und Irrtum
  - Entscheidungsbaum
  - Backtracking
  - Beispiele
    - *Labyrinth*
    - *Springer*
    - *8 Damen*
    - *Rucksack*
  - kombinatorische Explosion
  - Zielfunktion
  - obere Schranke
  - Bestfirst Search
  - Pruning
  - Minimax Problem
  - alphabeta Pruning
  - Tic-Tac-Toe