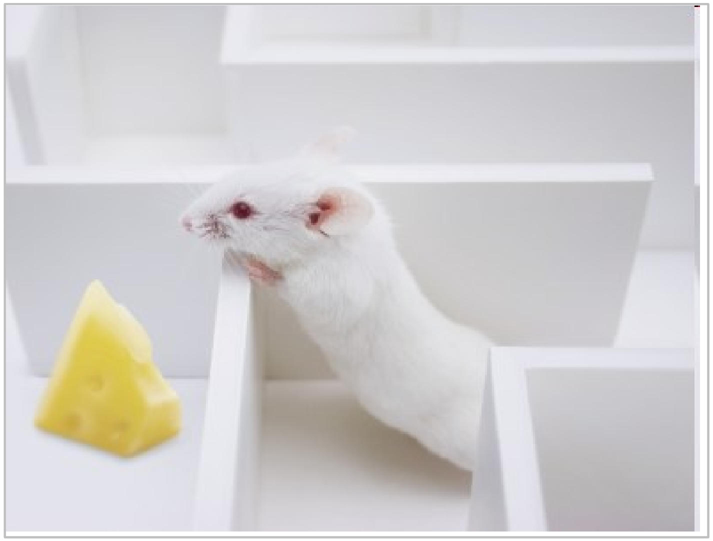
Trial and Error / Versuch und Irrtum

- Sie kennen Probleme, die nur oder am einfachsten mit Versuch und Irrtum gelöst werden können (Trial & Error)
- Sie kennen die Vorteile und Nachteile
- Sie wissen was ein Entscheidungsbaum ist
- Sie können einige bekannte Probleme erkennen
- Sie wissen was eine Zielfunktion ist
- Sie wissen wie Branch&Bound Verfahren funktionieren
- Sie wissen was Pruning bedeutet

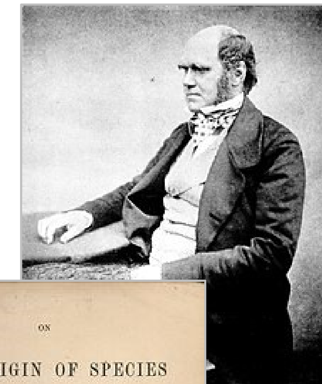


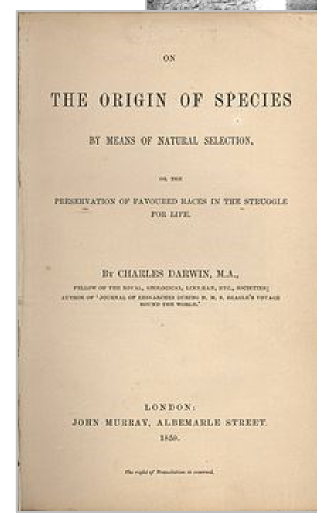
Basiert auf Material von:

Kurt Bleisch
Stephan Neuhaus
Karl Rege
Marcela Ruiz
Jürgen Spielberger

Trial and Error / Versuch und Irrtum

- Eigentlich bessere Bezeichnung wäre: Versuch und Bewertung
- Gem. Darwin die älteste Lösungsstrategie überhaupt:
  - Versuch = Mutation (genetische Variation)
  - Bewertung = Selektion (natürliche Selektion ≙ äussere Faktoren und sexuelle Selektion ≙ innerartlich)
- Charakteristik:
  - Trial & Error ist ziemlich rechen- und zeitintensiv
  - Nicht unbedingt beste Lösung als Resultat, es gibt zwei mögliche Resultate bei Trial & Error Ansatz:
    1. Beste Lösung
    2. Akzeptable Lösung (unter den gegebenen Rahmenbedingungen)





ON

THE ORIGIN OF SPECIES

BY MEANS OF NATURAL SELECTION,

OR THE

PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE.

BY CHARLES DARWIN, M.A.,

LONDON:
JOHN MURRAY, ALBEMARLE STREET.
1859.

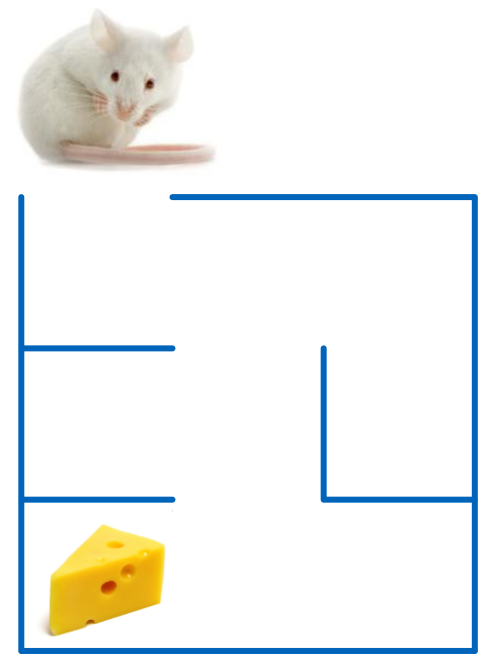
Charles Darwins Hauptwerk

Backtracking

Backtracking: Beispiel Labyrinth

Probleme:

- Maus sucht Käse
- Maus sucht Ausgang (auf dem kürzesten Weg)



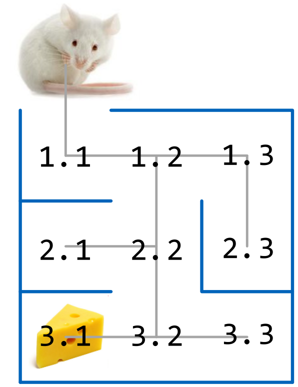
Einfacher Lösungsalgorithmus:

Diese Methode rekursiv aufrufen.

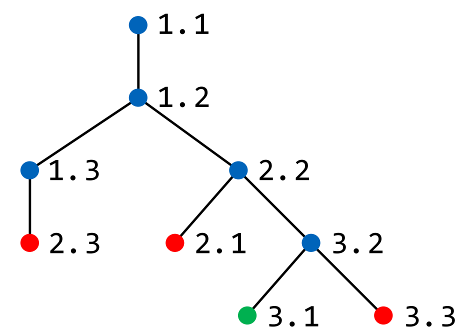
```
Gehe einen der Wege entlang...
   1) bis zur einer Verzweigung → Rekursion
   2) bis am Ziel → gefunden (Abbruch)
   3) bis Sackgasse → dann gehe zur nächsten
      Verzweigung zurück die noch einen nicht
      probierten Weg aufweist (Rücksprung aus
      Rekursion)
```

Das Labyrinth darf keine Zyklen enthalten. Für Zyklen muss dieser Algorithmus ergänzt werden.

Backtracking: Entscheidungsbaum,



- Es entsteht so ein virtueller Entscheidungsbaum. Jede Entscheidung (Verzweigung) entspricht darin einem Knoten.
- Teillösungen werden systematisch zu Gesamtlösungen erweitert bis Lösung gefunden ist oder Erweitern nicht mehr möglich ist (→ Sackgasse).
- Bei Sackgasse werden ein oder mehrere Schritte rückgängig gemacht. Von dort aus wird versucht, eine Lösung zu finden.
- Aus Sackgassen einen Weg zurück finden wird als Backtracking bezeichnet.
- Zeitkomplexität: $O(z^n)$; z ≙ Verzweigungsgrad; n ≙ Tiefe.



Backtracking: Rekursiver Pseudocode

Zürcher Hochschule für Angewandte Wissenschaften

zh aw School of Engineering

- Lösung: Bsp. Pfad durch das Labyrinth
- Mit dem Vorwärtsgehen im Entscheidungsbaum wird die Teillösung erweitert.

```
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;
}
```

Falls das Labyrinth Zyklen enthält, wird damit ein «im Kreis gehen» verhindert.

Dt. anschliessend

Jeden anschliessenden Knoten ein Mal überprüfen.

Beim Zurückgehen müssen die Knoten wieder «freigegeben» werden.

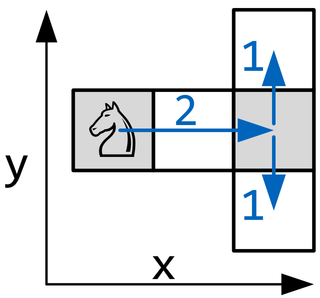
- Es müssen systematisch alle möglichen Erweiterungen durchprobiert werden.

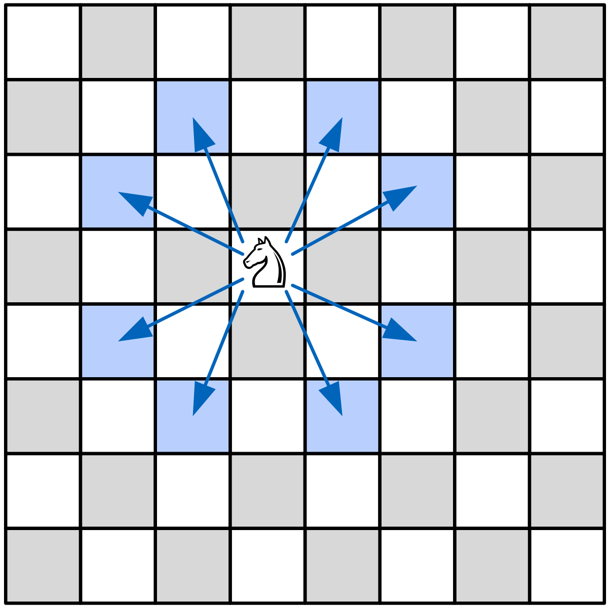
Springerproblem

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





Vierte mögliche Bewegung in x und y Richtung zur nächsten Position.

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

Springerproblem: Datenstrukturen und Methoden

- Idee: Mittels Backtracking alle möglichen Wege absuchen.
- Zeitkomplexität mit Backtracking (n = Anz. Spalten & Zeilen): $O(8^{n \cdot n})$
  $8 \cdot 8 \cdot 8 \ldots \cdot 8$ bei 64 Felder → $8^{64} = 6.3 \cdot 10^{57}$
- Datenstruktur:

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

  Der int-Wert in Feld soll angeben, in welchem Zug das Feld besucht wurde.
  Wird mit 0-Werten initialisiert.

- Methode:

```
boolean versuchen(int x, int y, int nr)
```

  x, y: Koordinaten des Feldes
  nr: Nummer des Zuges (≥1)

Springerproblem: Algorithmus

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

Position erlaubt?

```
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 ermitteln.

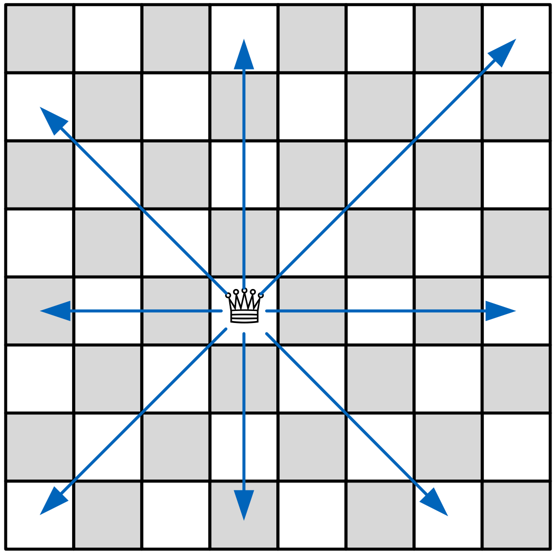
8 Damenproblem

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.



8 Damenproblem: Datenstrukturen und Methoden

- Idee: Mittels Backtracking alle Kombinationen ausprobieren.
- Zeitkomplexität: O(n!)

Anz. Spalten.

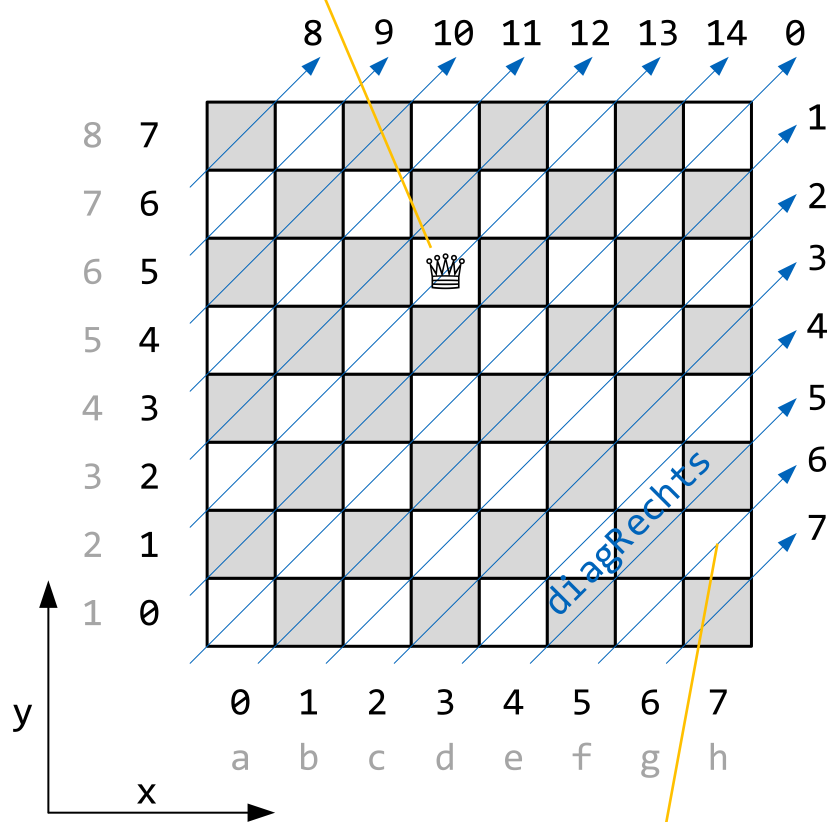
```
int[] dameInDerSpalte = new int[n];
static boolean[] reihe = new boolean[n];
```

Anz. Diagonalen.

```
int diagN = 2*n - 1;
boolean[] diagLinks = new boolean[diagN];
links = (x + y) % diagN;
```

Berechnung der Nr. der Diagonale.

```
boolean[] diagRechts = new boolean[diagN];
rechts = (diagN + x - y) % diagN;
```




```
(15 + x – y) % 15
(15 + 7 – 1) % 15 → 6
```

8 Damenproblem: Datenstrukturen und Methoden

```java
// testet ob Position möglich ist
// Wert in einem der 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]);
}
```

```java
// 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;
}
```

Konditionaloperator: Falls val = true, dann ergibt der Ausdruck y sonst -1.

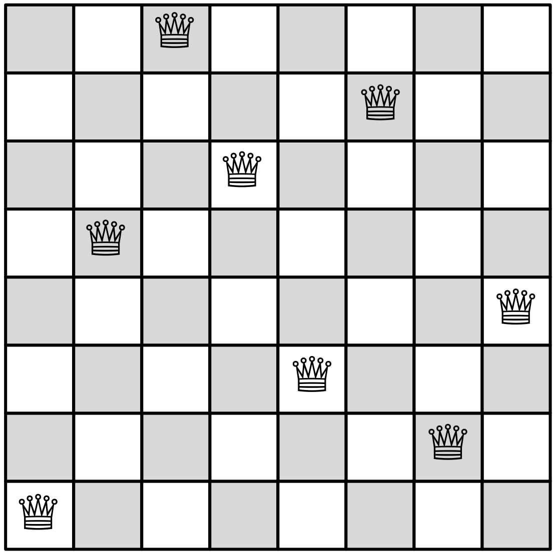
8 Damenproblem: Algorithmus

```
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);
                return (versuchen(x + 1));
                setzeDame(x, y, false);
            }
        }
        return false;
    }
}
```

Spalte die ausprobiert wird.

Lösung gefunden.

Dame in Spalte x in allen Reihen ausprobieren.



Hier fehlt noch die Ausgabe der Lösung…

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 von ihm geraubten Gegenstände maximal wird.

Rucksackproblem: Gegenstände

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 5 Gegenstände von unterschiedlicher Grösse und den angegebenen Werten.
- Die Bezeichnungen der Gegenstände werden im Programm in Indizes umgewandelt: 0 bis 4

Rucksackproblem: Pseudocode

- Idee: Mittel Backtracking alle möglichen Varianten ausprobieren.
- Zeitkomplexität: $O(2^n)$ (Erklärung folgt).

```
void teste (Gegenstand k) {
   teste (k + 1) // ohne Gegenstand k
   falls 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
   }
}
```

Teste alle Varianten ohne Gegenstand k.

Teste alle Varianten mit Gegenstand k.

Rucksackproblem: Datenstrukturen

```
double[] volume = {1, 2,  7,  8,  9};
double[] wert =   {2'000, 3'000, 10'000, 11'000, 17'000};
Set<Integer> maxRucksack;
final double MAXV = 17;
double maxW = 0;
```

Das Set enthält die Gegenstände zur besten gefundenen Lösung.

Maximales Volumen des Rucksacks.

Wert der besten gefundenen Lösung.

Rucksackproblem: Algorithmus

```
static public void test(Set<Integer> rucksack, int k, double aktW, double aktV) {
   double newV;
   if (k < volumen.length) {
      test(rucksack, k + 1, aktW, aktV);
      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);
      }
   }
}
```

Suche das Maximum ohne Gegenstand k.

Ab hier: Suche das Maximum mit Gegenstand k.

Die Lösung ist gültig, ist es grösser als die bisherigen Lösungen?

Der Gegenstand muss wieder entfernt werden, da die selbe Methode für denselben Gegenstand mehrfach aufgerufen wird.

Rucksackproblem: Aufwandsbetrachtung

Frage: Auf wie viele Arten kann ich 1, 2, 3, …, k unterscheidbare Gegenstände auswählen (jeden Gegenstand nur einmal)?

- 1 Gegenstand: {}, {0} → 2 Arten
- 2 Gegenstände: {}, {0}, {1}, {0, 1} → 4 Arten
- 3 Gegenstände : {}, {0}, {1}, {0, 1} , {2}, {0, 2}, {1, 2}, {0, 1, 2} → 8 Arten
- 4 Gegenstände : {«3 Kugeln»}, {4}, {4, «3 Kugeln»} → 16 Arten
- ….
- k Kugeln: $2 \cdot 2 \cdot 2 \cdot 2 \ldots \cdot 2 \rightarrow 2^k$

Aufwand: $O(2^n)$

Rucksackproblem: Anwendungen

Transportunternehmen:

Optimale Beladung eines Lastwagens bei gegebenen Maximalgewicht und unterschiedlichen Speditionsgebühren: Optimierung von Gewicht und Gebühren.

Reederei:

Optimale Beladung eines Schiffes mit unterschiedlichen Containern: Optimierung von Volumen und Transportkosten.

Kofferproblem:

Optimale Beladung des Reisekoffers (max. 20kg) für einen Flug.

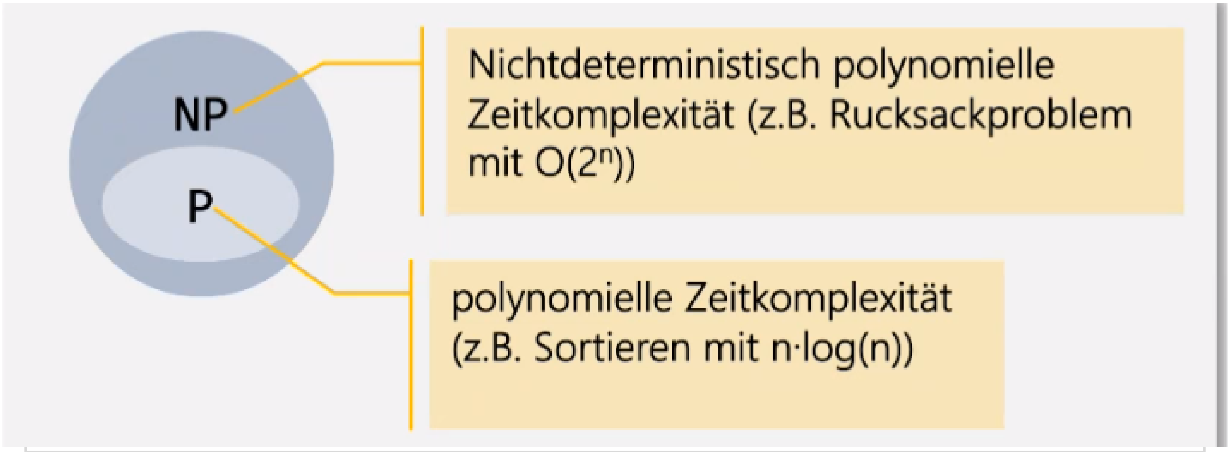
Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Komplexität der Probleme

Komplexität der Probleme: Erschöpfende Suche

- Das Rucksackproblem ist ein Beispiel von einer Klasse von algorithmischen Problemen, bei denen keine bessere Lösung bekannt ist, als das Ausprobieren sämtlicher Möglichkeiten: Versuch & Irrtum.
- Durchsuche aller Möglichkeiten → erschöpfende Suche (Engl. exhaustive search)
- Der Aufwand bei diesen Algorithmen ist meist $O(k^n)$ oder $O(n!)$, entsprechend der Anzahl möglicher Kombinationen oder Permutationen.

- Frage: Gibt es eine Ordnung/Klassifikation in der Komplexität von Problemen?



Komplexität der Probleme: Genereller rekursiver Backtracking Algorithmus

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

Zeitkomplexität: $O(z^n)$; z ≙ Verzweigungsgrad; n ≙ Tiefe.

Komplexität der Probleme:
Beispiele von Optimierungsproblemen

- 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

Komplexität der Probleme: 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 → $8^{64} = 6.3 \cdot 10^{57}$ 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. Grössenvergleich zum Springerproblem:
  - Alter des Universum: $5 \cdot 10^{17}$ s
  - Masse der Sonne $2 \cdot 10^{33}$ g
  - Anzahl Atome im sichtbaren Weltall: 10^{77}

➢ Wir können nicht alle Kombinationen ausprobieren

Suchverfahren mit Zielfunktion

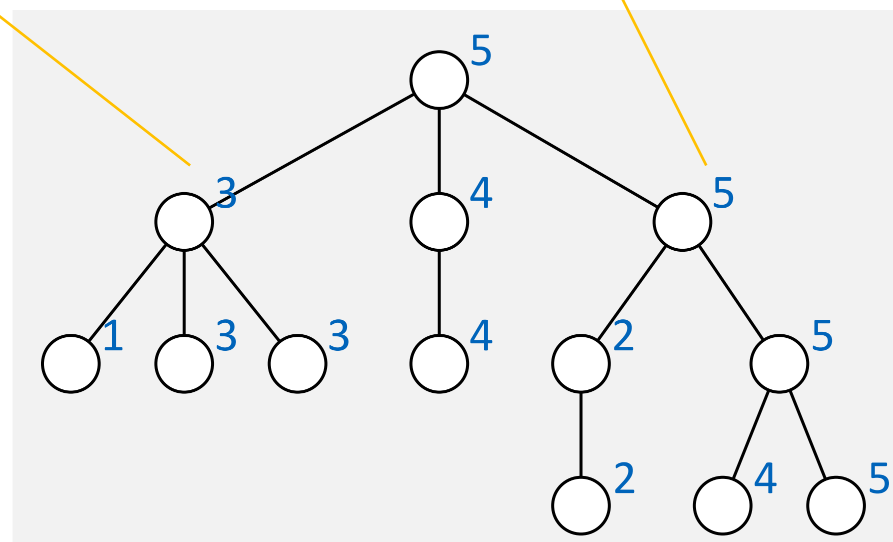
Zielfunktion

- Erste Idee: Umgehen der kombinatorischen Explosion
  - Man wählt nur die Lösung aus, die zum Ziel führt.
  - Man berechnet zu jedem Knoten im Entscheidungsbaum den (besten) **Zielwert**, den man über diesen Knoten erreichen kann.
- Diese Funktion wird als **Zielfunktion f(v)** bezeichnet

v = Vertex

Dieser Teilbaum liefert als besten Wert einen Zielwert von 5.

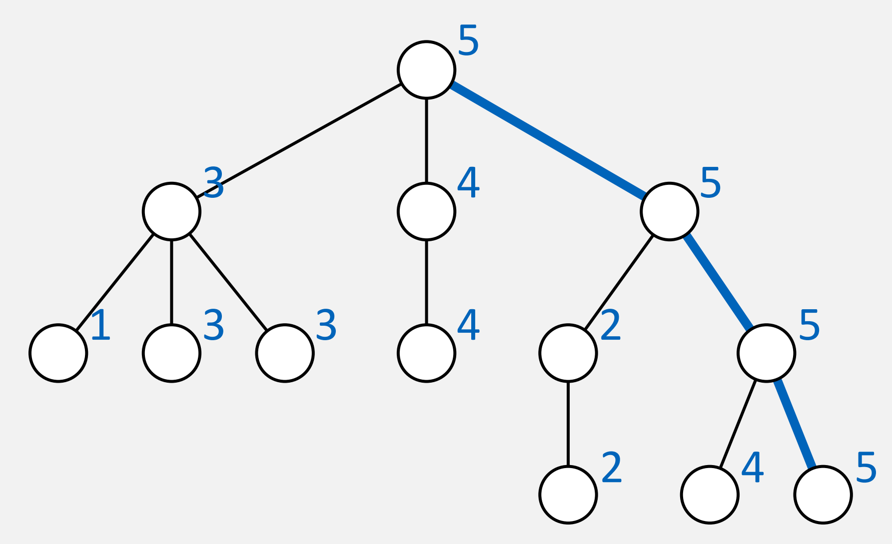
Dieser Teilbaum liefert als besten Wert einen Zielwert von 3.



Zielfunktion: Algorithmus

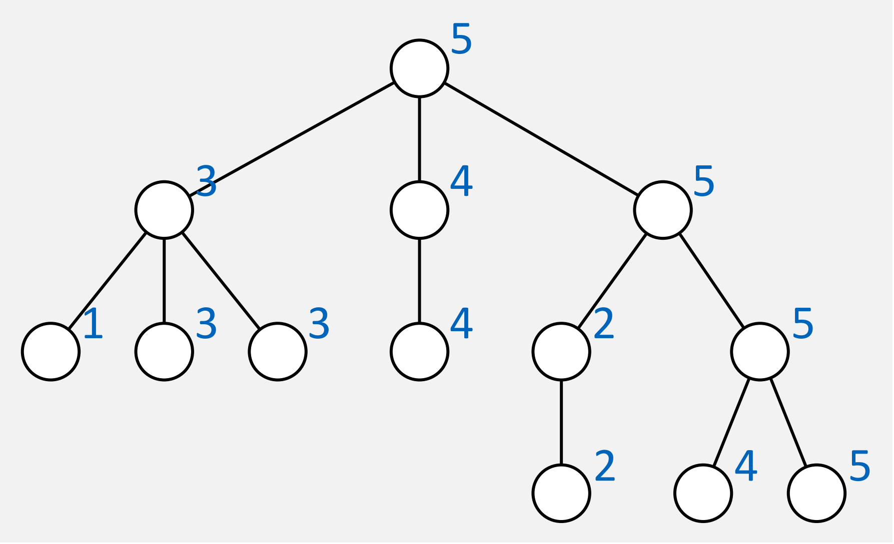
- Für jeden Knoten:
  - Berechne zu jedem Nachfolgeknoten im Entscheidungsbaum die **Zielfunktion f(v)**.
  - Gehe der Kante entlang (wähle die Teillösung aus), die zum Knoten mit dem höchsten Zielfunktionswert führt.
- Problem gelöst:
  - keine kombinatorische Explosion.
  - sehr effizienter Algorithmus : ~ Log(n)

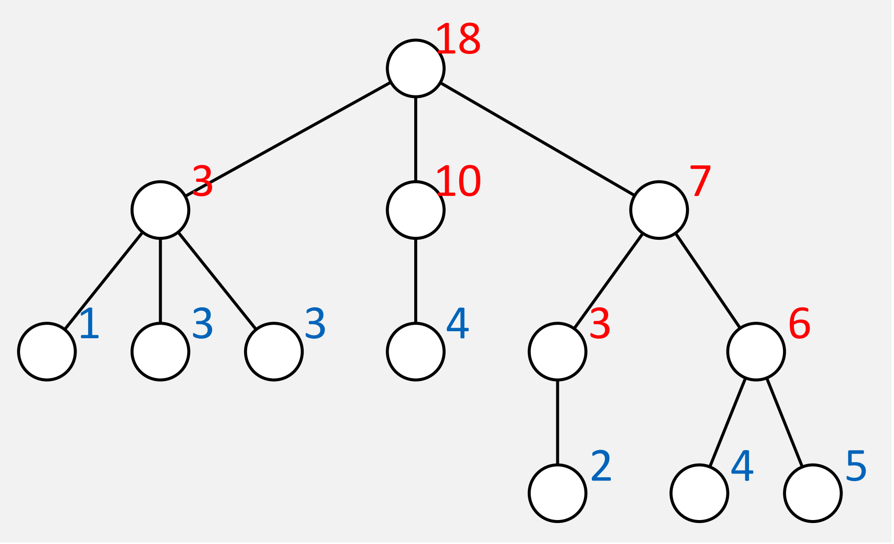
Aber:
Zur Berechnung der Zielfunktion muss das Problem meist schon gelöst sein, d.h. z.B. der Entscheidungsbaum unterhalb des Knotens vollständig durchlaufen sein → wir haben nichts gewonnen.



Zielfunktion: Geschätzte obere Schranke

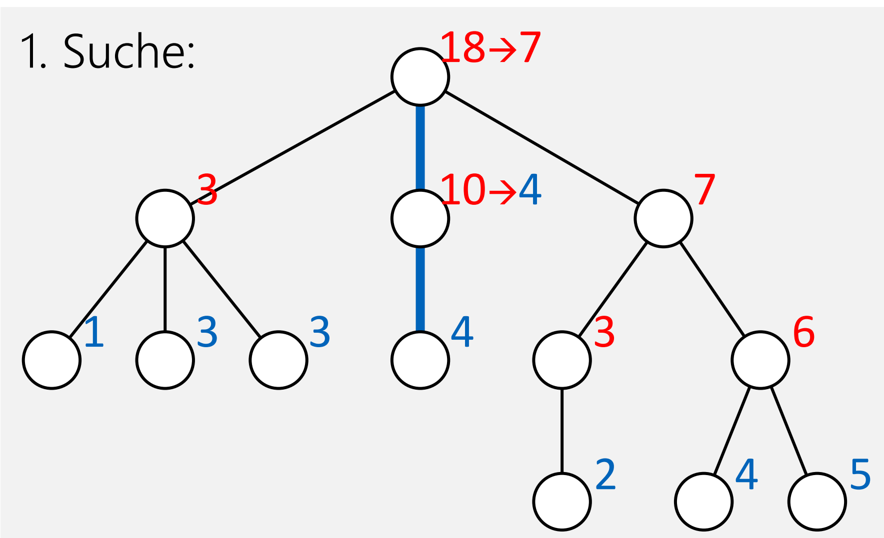
- Verbesserung der Idee: Es wird nicht die exakte Zielfunktion, sondern eine einfacher zu bestimmende Funktion verwendet, ohne dass der gesamte Teilbaum untersucht werden muss: «Score der Lösung», Fitness Function, Kostenfunktion.
- Es wird eine Bound-Funktion b bestimmt, die immer bessere Werte liefert als die exakte Zielfunktion f: obere Schranke b(v) >= f(v).

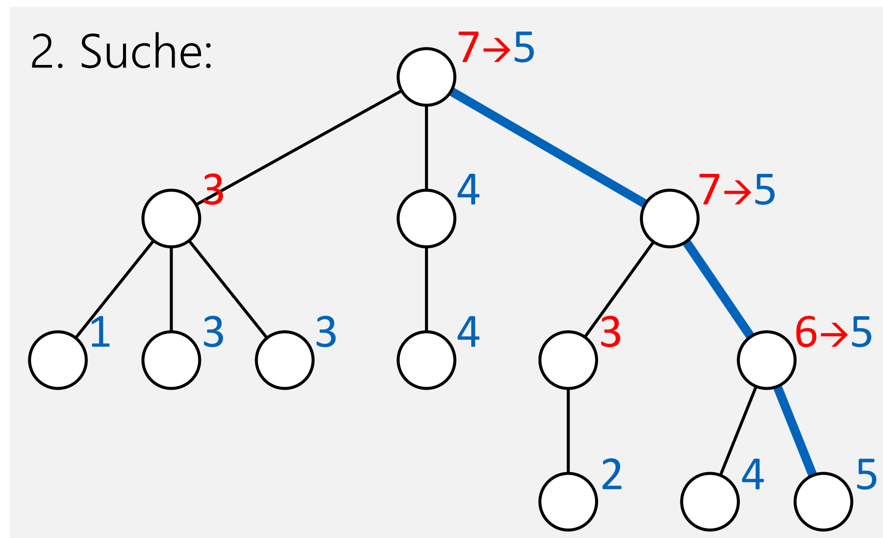




Zielfunktion: Best-first Search

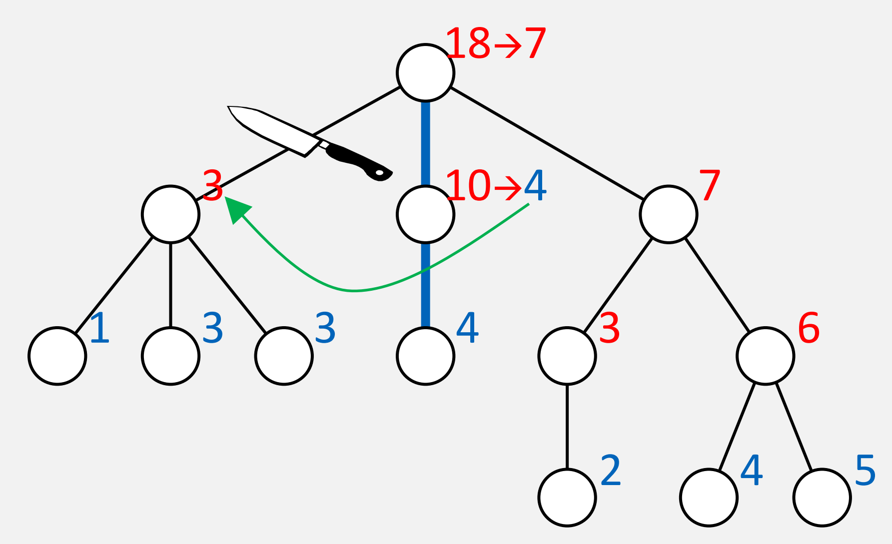
- **Best-first Search**: Gehe dem Pfad mit dem höchsten Bound-Wert zuerst entlang.
- Korrigiere den b(v)-Wert des betrachteten Knotens anhand der Bound-Werte der darunter liegenden Knoten bzw. des erreichten Ergebnisses. Der Bound-Wert ist das Maximum der Bound Werte der direkten Nachfolger.
- Ist der gefundene Wert höher als die Bound-Werte, dann haben wir die Lösung gefunden.
- Kombination aus Breitensuche und Tiefensuche.



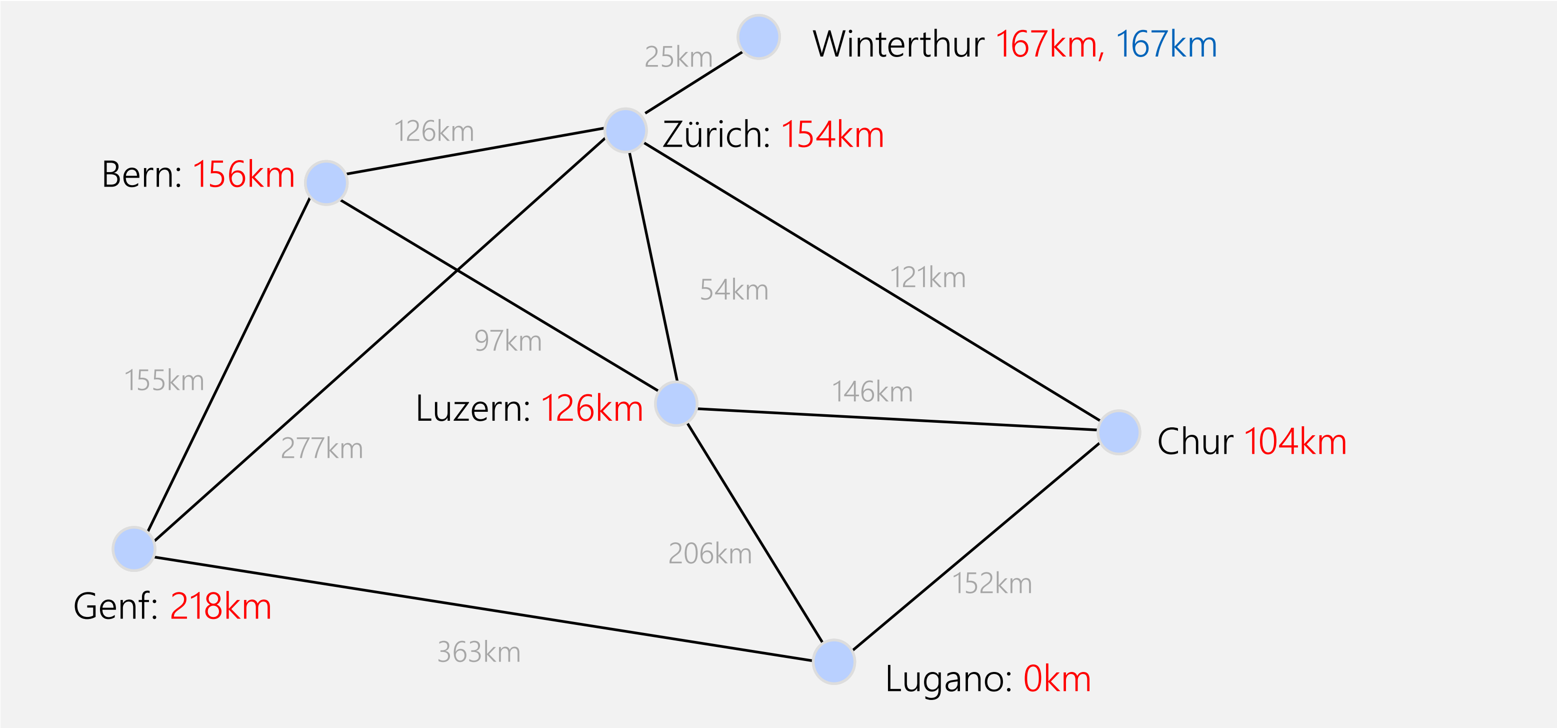


Zielfunktion: Abschneiden (Cutoff, Pruning)

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

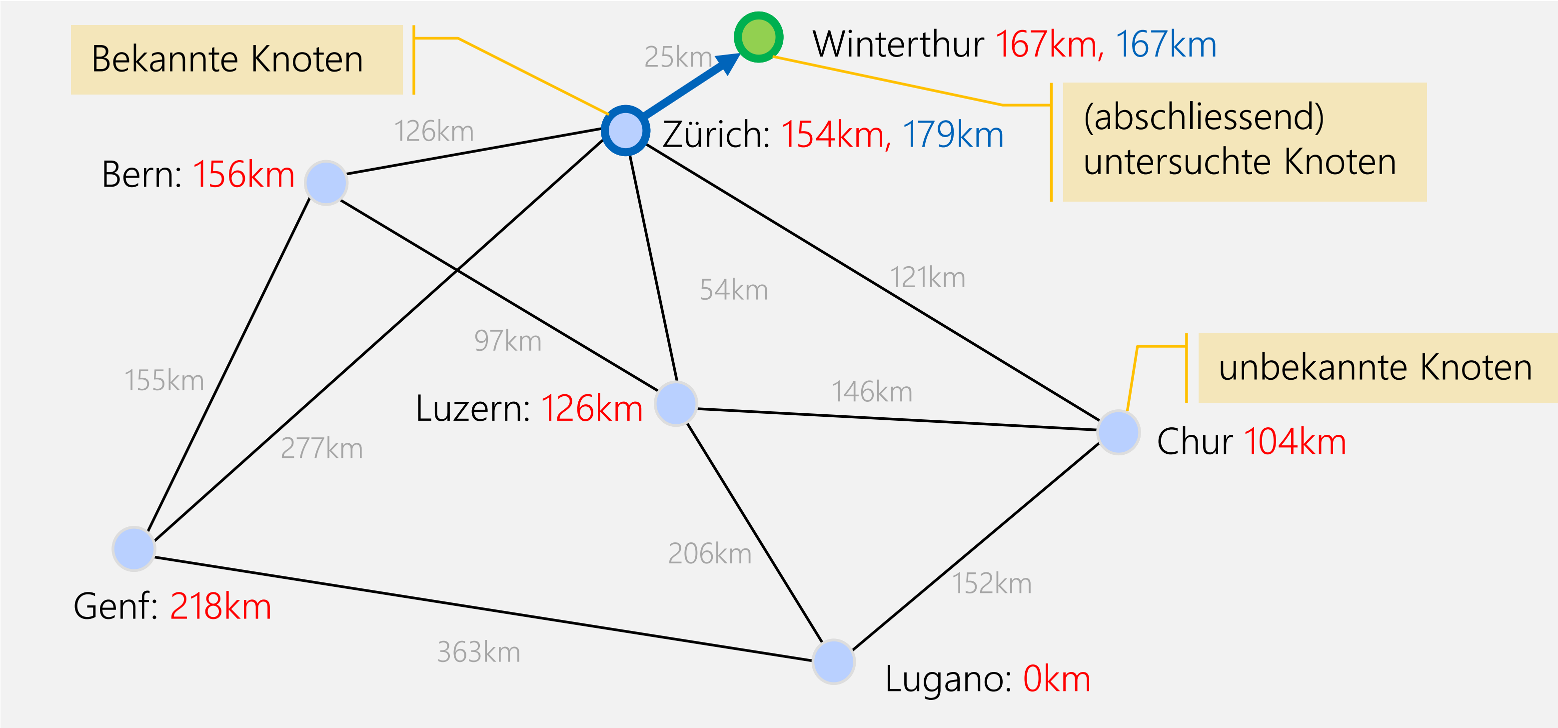


Zielfunktion: Beispiel A*-Algorithmus



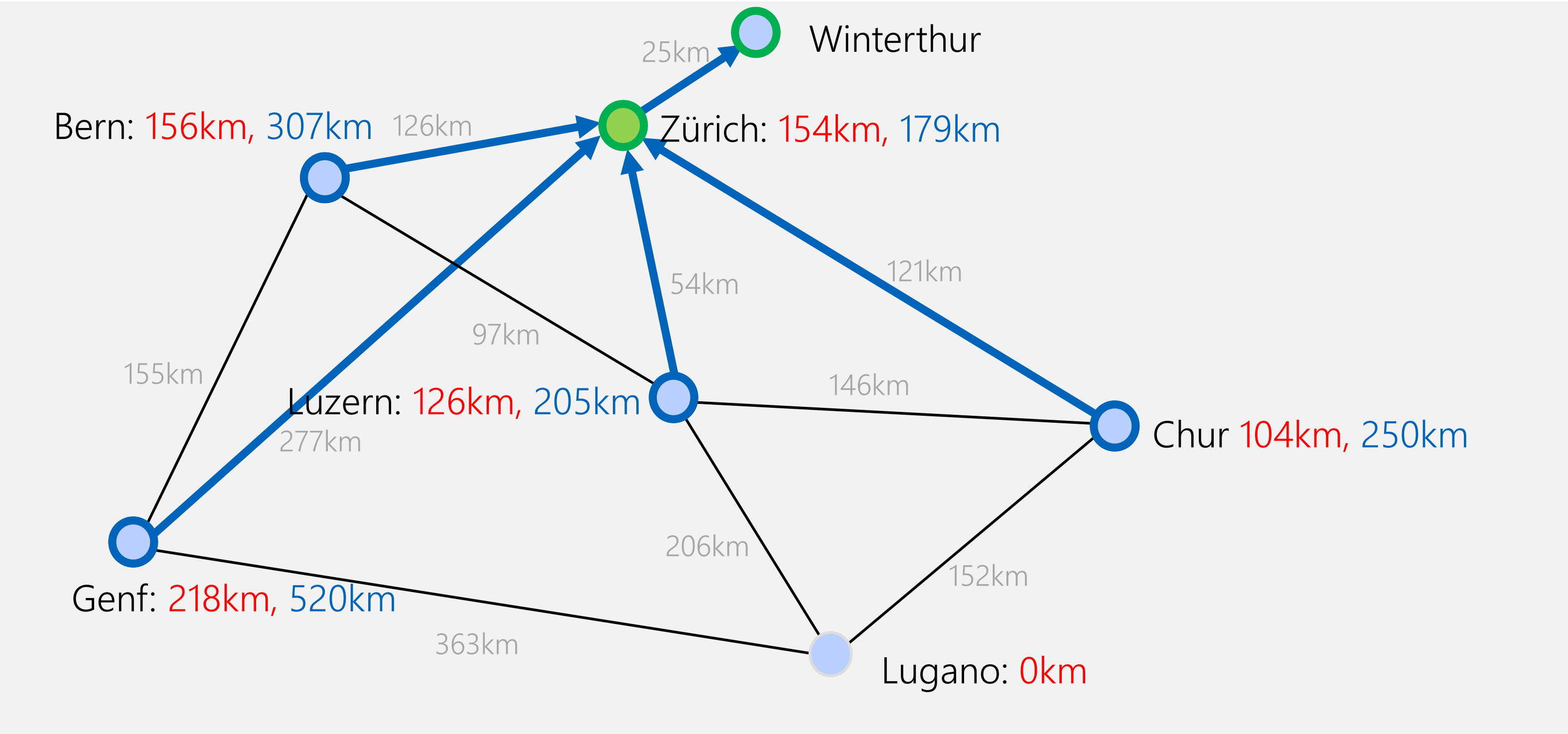
- Wir suchen hier den kürzesten Weg zwischen Winterthur und Lugano, die Bound-Funktion soll daher den potentiell besten Wert je Ort zeigen → wir verwenden den Luftweg zwischen den Orten.
- Hier wird, im Gegensatz zum Upper-Bound im Beispiel vorher, der Lower-Bound verwendet. Am Knoten geben wir **h(x)**, das ist der Luftweg bis zum Ziel, und f(x) = g(x) + **h(x)** mit g(x) = «Distanz vom Start bis x» an.

Zielfunktion: Beispiel A*-Algorithmus



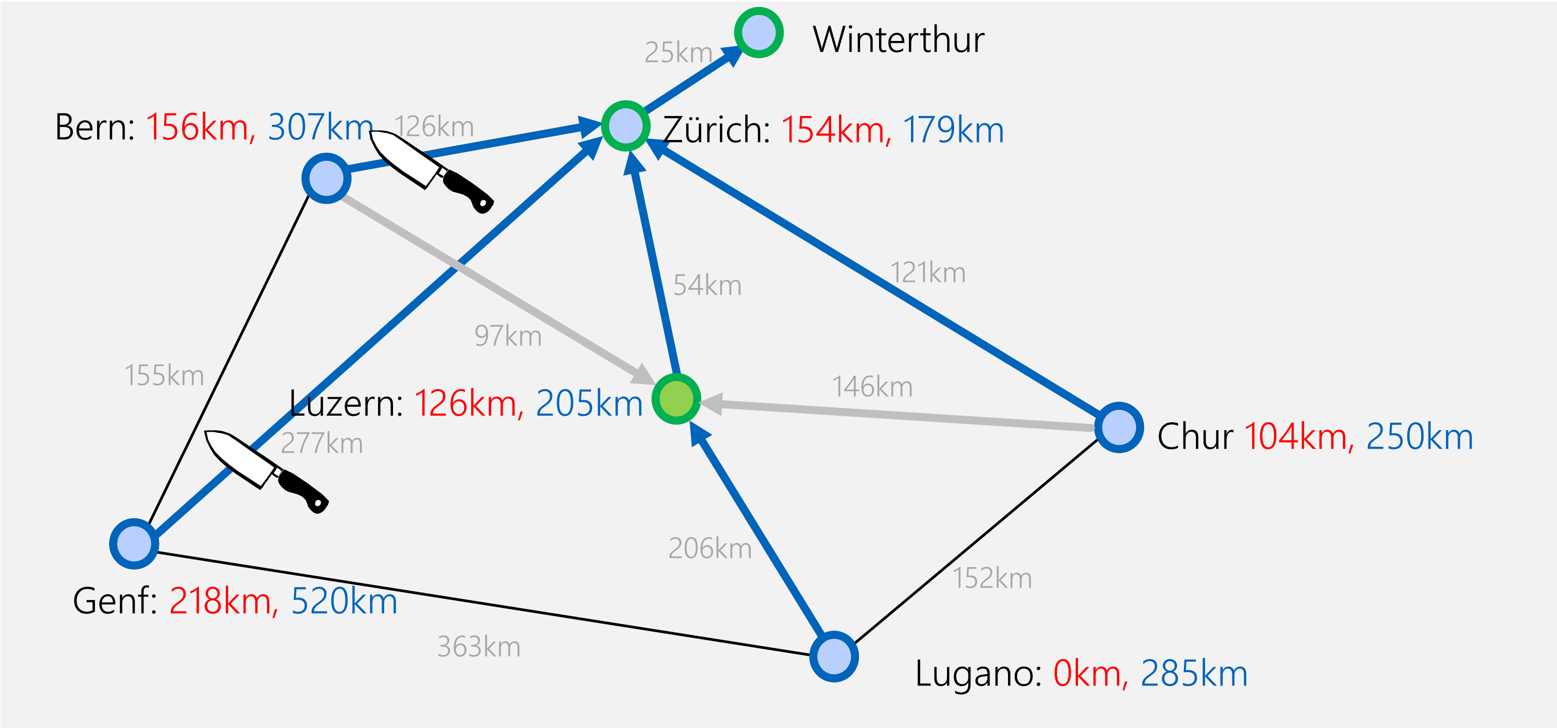
- Wir betrachten zunächst alle nicht untersuchten Nachbarnoten von Winterthur: das ist nur Zürich.
- Für Zürich berechnen wir die minimal erwarteten Kosten: 25km + 154km = 179km, das ist der f-Wert. Da es nur ein Knoten ist, bleiben diese Kosten aber ohne weitere Bedeutung.

Zielfunktion: Beispiel A*-Algorithmus



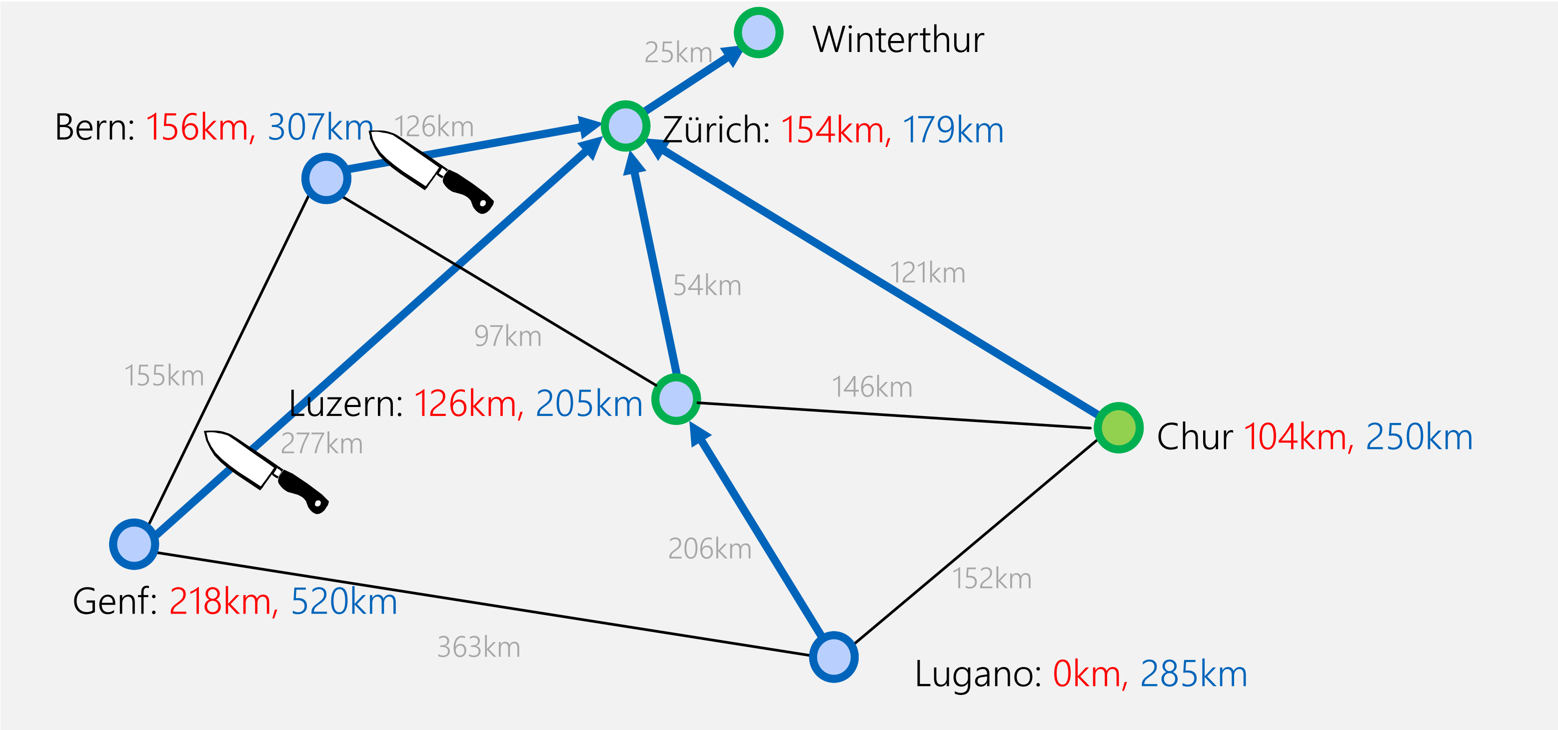
- Wir betrachten nun alle nicht untersuchten Nachbarknoten von Zürich und berechnen deren f-Werte (Bern, Genf, Luzern und Chur, wobei z.B. Chur: 25 + 121 + 104 = 250).
- Wir wählen den vielversprechendsten, nicht untersuchten Knoten aus (f-Werte): Luzern mit 205km und untersuchen dessen nicht untersuchten Nachbarknoten Bern, Chur und Lugano.

Zielfunktion: Beispiel A*-Algorithmus



- Chur wird nicht angepasst, da der neue Weg (f-Wert) länger ist: 25 + 54 + 146 + 104 = 329 > 285.
- Auch für Bern keine Anpassung: 25 + 54 + 97 + 156 = 332 > 307
- Lugano erhält den f-Wert von 25 + 54 + 206 = 285km, bisher der kürzeste Weg.
- Der Weg von Zürich via Bern und Genf kann mittels Pruning entfernt werden, da diese bereits schlechter sind als die bisherige Lösung (520 > 307 > 285).

Zielfunktion: Beispiel A*-Algorithmus



- Im nächsten Schritt wird nun Chur als bester Kandidat untersucht. Dafür müssen die noch nicht untersuchten Nachfolgeknoten betrachtet werden: nur Lugano.
- Es ergibt sich kein besser f-Wert für Lugano 25 + 121 + 152 = 298 > 285, es muss nichts angepasst werden.
- Der nächstbessere Kandidat ist nun Lugano → die Suche ist abgeschlossen

Zielfunktion: Branch&Bound

- Branch: Aufteilen der Lösung in mehrere, einfachere Teilschritte (z.B. mit einem Baum).
- Bound: Durch das Festlegen von Schranken Teillösungen als nicht optimal kennzeichnen und nicht mehr verfolgen (abschneiden),

- Mittels Branch&Bound lässt sich das Problem der kombinatorischen Explosion eindämmen. Es ist ein allgemeines Verfahren, das an das jeweilige Problem angepasst werden muss.
- 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 grosse Obergrenze) → es können keine/wenige Teilbäume abgeschnitten werden → kombinatorische Explosion

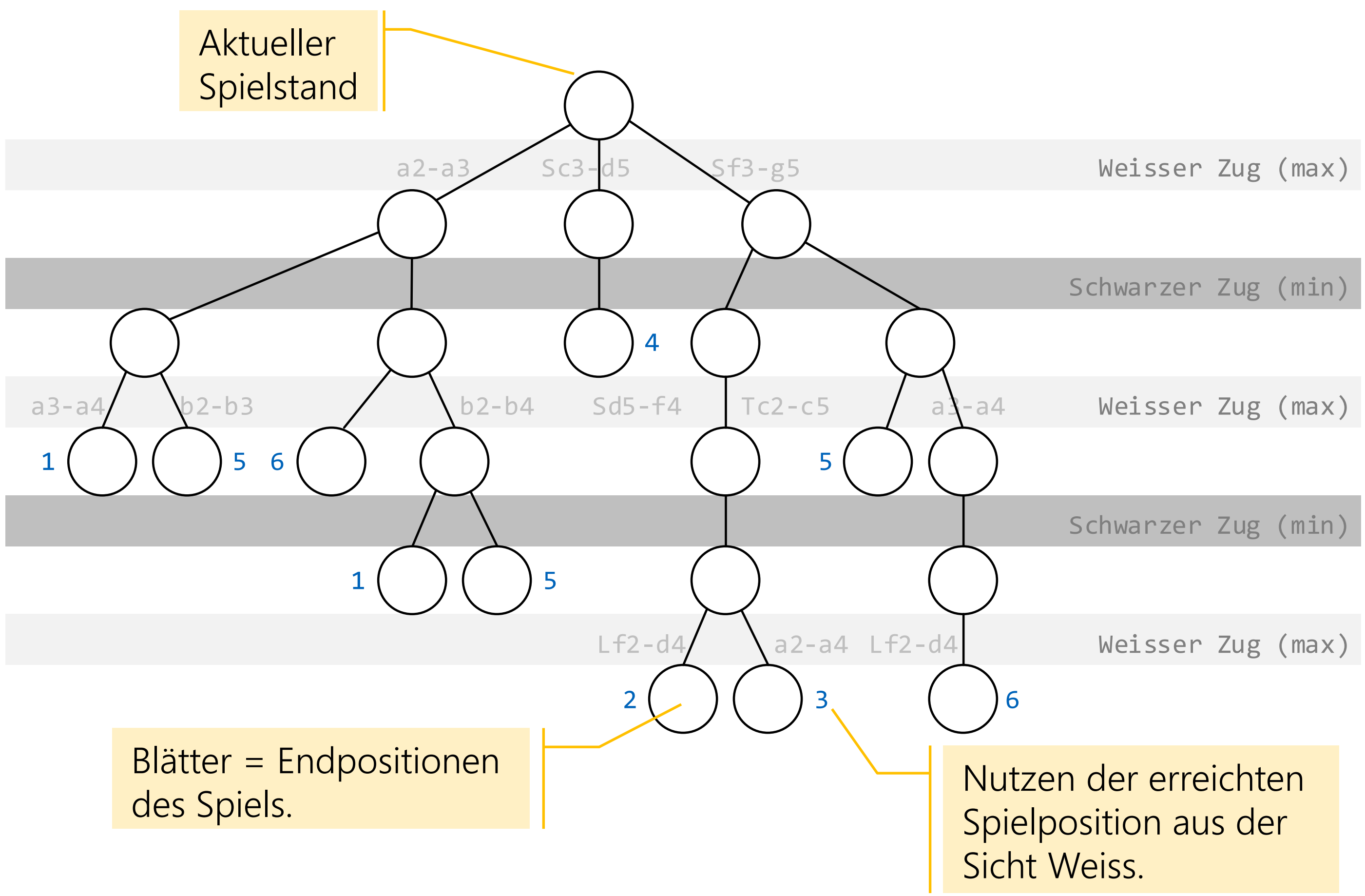
Zielfunktion: Beispiel Schach

- Entscheidungsbaum: alle möglichen Züge.
- Berechne für jede Endposition einen b(v)-Wert (auch Bewertungsfunktion bezeichnet).

- Figuren Werte:
  - König 100; Dame 9; Turm 5; Springer/Läufer 3.5; Bauer 1
- Positionswerte:
  - Beherrschung des Zentrums
  - Schutz des Königs und der restlichen Figuren
  - Bedrohung der gegnerischen Königs&Figuren
  - …

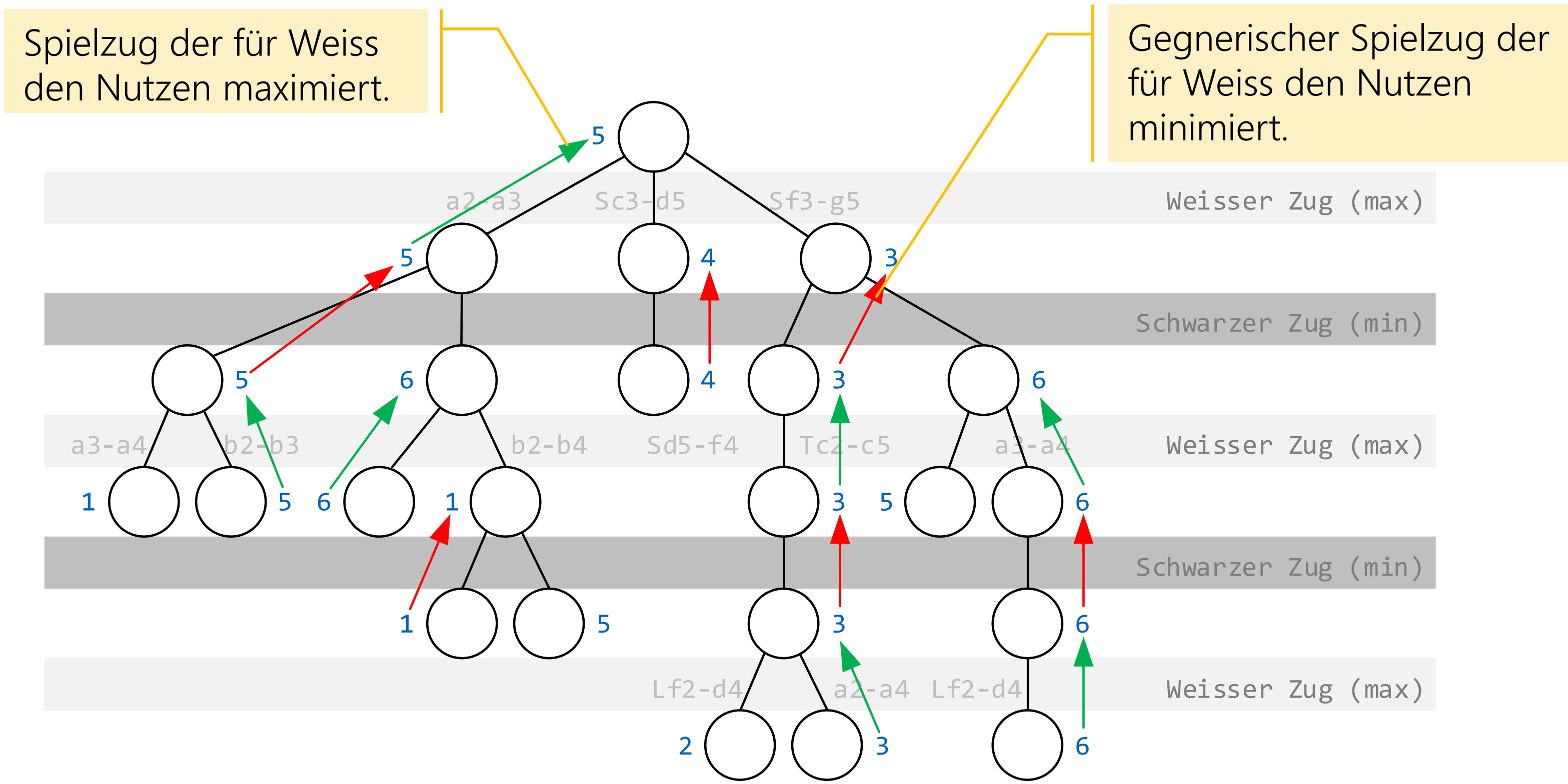
Zielfunktion: Beispiel Schach

- Jeder Zug führt zu einer anderen Spielposition, die bewertet werden kann. Aber: es kommen beide Spieler abwechselnd an die Reihe.
- Schwarz wird einen Gegenzug machen, der das b(v) **möglichst minimiert**.



Zielfunktion: Beispiel Schach Minimax-Alg.

- Bei den Blättern beginnend, wird jeweils der maximale resp. minimale Nutzen als «Weg» gewählt.



Zielfunktion: Minimax-Alg. Pseudocode

```
Tiefensuche durch den Spielbaum (erstellt den Baum)
Wende Bewertungsfunktion auf Endpositionen (Blätter) an
Für alle inneren Knoten von unten nach oben:
    Falls beim inneren Knoten Schwarz am Zug war:
        Wähle das kleinste b(v) für den inneren Knoten.
    Falls beim inneren Knoten Weiss am Zug war:
        Wähle das grösste b(v) für den inneren Knoten.
Wähle an der Wurzel den Zug der den höchsten b(v) verspricht.
```

Zielfunktion: Rekursiver Minimax-Alg. Pseudocode

Zürcher Hochschule für Angewandte Wissenschaften

School of Engineering

```
// Aufruf der Methode
mimimax(currentPosition, 3, true);
```

Aktuelle Spielposition

```
public static int minimax(position, depth, maximizingPlayer) {
   if depth = 0 or gameOver { return b(position); }
   if maximizingPlayer {
      int maxEval = -infinity;
      for each child of position // alle möglichen Züge
         maxEval = max(maxEval, minimax(child, depth – 1, false));
      return maxEval;
   }
   else {
      int minEval = +infinity;
      for each child of position // alle möglichen Züge
         minEval = min(minEval, minimax(child, depth – 1, true);
      return minEval;
   }
}
```

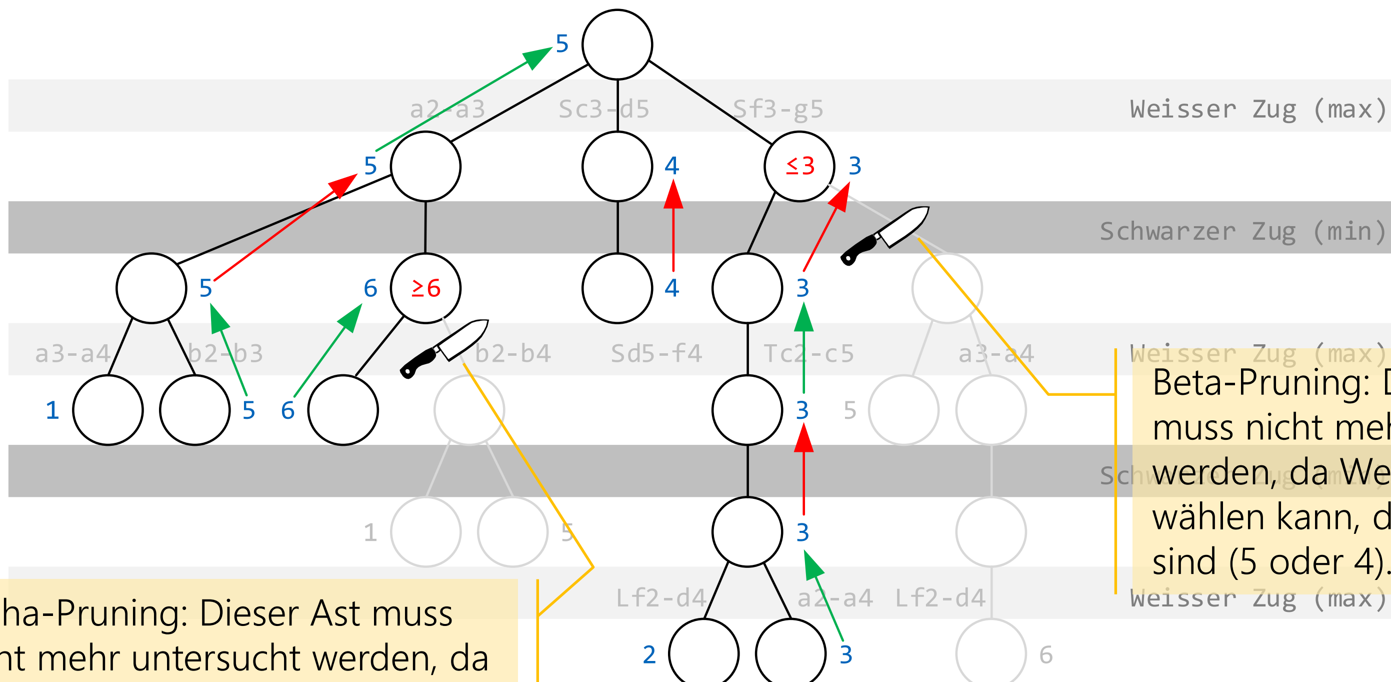
Zielfunktion: Der Horizont Effekt

- Bei jeder Stellung ca. 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.
  - 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.

Minimax Algorithmus, Alpha-Beta-Pruning

- Das Alpha-Beta-Pruning ist eine optimierte Variante des Minimax-Suchverfahrens:
  - Der Minimax-Algorithmus analysiert den vollständigen Suchbaum.
  - Das Alpha-Beta-Pruning ignoriert alle Knoten, bei denen bereits bei der Suche feststeht, das dieser Ast das Ergebnis nicht beeinflussen kann.

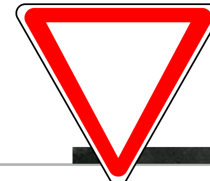


Beta-Pruning: Dieser Ast muss nicht mehr untersucht werden, da Weiss Optionen wählen kann, die besser als 3 sind (5 oder 4).

Alpha-Pruning: Dieser Ast muss nicht mehr untersucht werden, da Schwarz eine Option wählen kann, die besser als 6 ist (5).

Zusammenfassung

- Entscheidungsbaum
- Backtracking
- Labyrinth
- Springer
- 8 Damen
- Rucksack
- Kombinatorische Explosion: Suchfunktionen mit Zielfunktion
- Pruning
- Minimax-Algorithmus
- Alpha-Beta-Pruning
- Tic-Tac-Toe



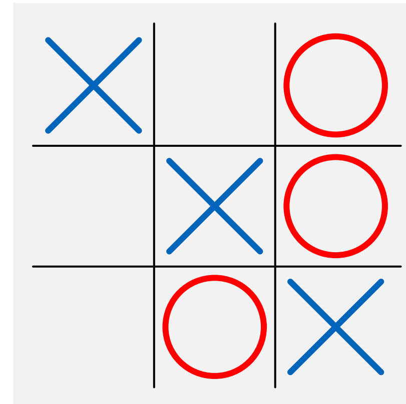
Kontrollfragen Lektion xx nicht vergessen – heute mit Rapunzel

Nerd-Zone

Tic-Tac-Toe

Tic-Tac-Toe

- Spiel, bei dem der ganze Entscheidungs-baum berechnet werden kann (Grösse: 3 x 3).
- Ziel: Wer zuerst drei X oder drei O in einer Reihe, Spalte oder Diagonalen hat, hat gewonnen. Im Bild hat Blau gewonnen.



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: O(n!). Bereits bei einer Feldgrösse von 4x4 wird Ihr PC ca. 60'000'000 mal länger brauchen... zu lange!

Tic-Tac-Toe: Datenstrukturen

Anzahl Diagonalen

Anzahl Markierungen der n-ten Spalte je Spieler (0 oder 1)

Besetzte Felder (FREE, HUMAN oder COMPUTER)

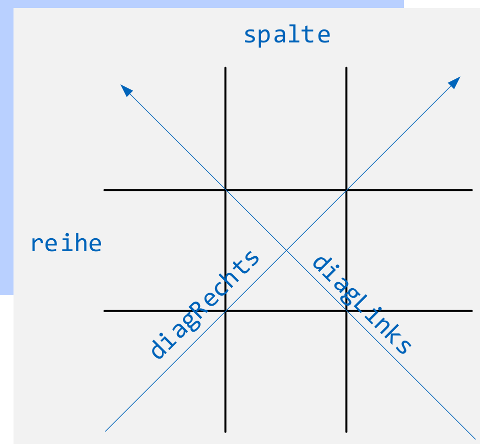
```
static int n = 3; // muss grösser 2 sein, sonst gewinnt immer der Erste
static int diagN = 2 * n -1;

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;
}
```

Tic-Tac-Toe: Hilfsmethoden

Methode überprüft, ob jemand und wer gewonnen hat.

```
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;
   }
   for (int y = 0; y < diagN; y++) {
      if (diagLinks[COMPUTER][y] == n) return COMPUTER_WIN;
      if (diagLinks[HUMAN][y] == n) return HUMAN_WIN;
      if (diagRechts[COMPUTER][y] == n) return COMPUTER_WIN;
      if (diagRechts[HUMAN][y] == n) return HUMAN_WIN;
   }
   return UNCLEAR;
}
```

Algorithmus

WARNING! NERD ZONE

Wir haben einen Gewinner

Muss Upper-Bound oder Lower-Bound verwendet werden?

Position noch frei

Maximieren

Minimieren

```
public static int tryMove(int side, int moveNo) {
   int score, bX = -1, bY = -1, status;
   if (moveNo == n * n) return UNCLEAR;
   else {
      status = win();
      if (status != UNCLEAR) return status;
      score = (side == COMPUTER)?HUMAN_WIN:COMPUTER_WIN;
      for (int x = 0; x < n; x++) {
         for (int y = 0; y < n; y++) {
            if (board[x][y] == FREE) {
               setze(side,x,y);
               int val = tryMove((side + 1) % 2, moveNo + 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;
}
```