Algorithmen und Datenstrukturen

Vorlesung #05 – Alpha-Beta-Suche und *Branch-and-Bound*

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

16 · Mai · 2023

Themen der heutigen Vorlesung

- Minimax-Suchverfahren
- Alpha-Beta-Suche
- *Branch-and-Bound*
  - Begrenzung des Lösungsraums durch Schranken für Lösungswerte
  - Beispiel: 0/1 Knapsack

Minimax-Verfahren

- Wie können Lösungen in einem kompetitiven Setting gefunden werden?
- Dies kann z. B. ein Zwei-Personen Spiel sein. Wir betrachten Nullsummen-Spiele – was der Gewinn des einen, ist der Verlust des anderen.
- Die Spieler wählen abwechselnd einen Zug und haben gegensätzliche Ziele.
- Das **Minimax-Verfahren** durchläuft den Lösungsraum nach *Backtracking* Manier und trifft die Auswahl mit abwechselnden Kriterien, z. B.
- Wähle das Maximum der Bewertungen für Züge von Spieler **A** und Minimum für Spieler **B**. Bewertung aus Sicht von **A**.
- Dies Grundprinzip liegt vielen Programmen für Strategiespiele wie Schach zu Grunde.
- Dabei müssen die Züge nicht bis Ende durchgerechnet werden. Dann benutzt man nach einer bestimmten Zugtiefe eine Bewertung für die Spielstellung.

Minimax-Verfahren - Beispiel: Spiel *BestDifference*

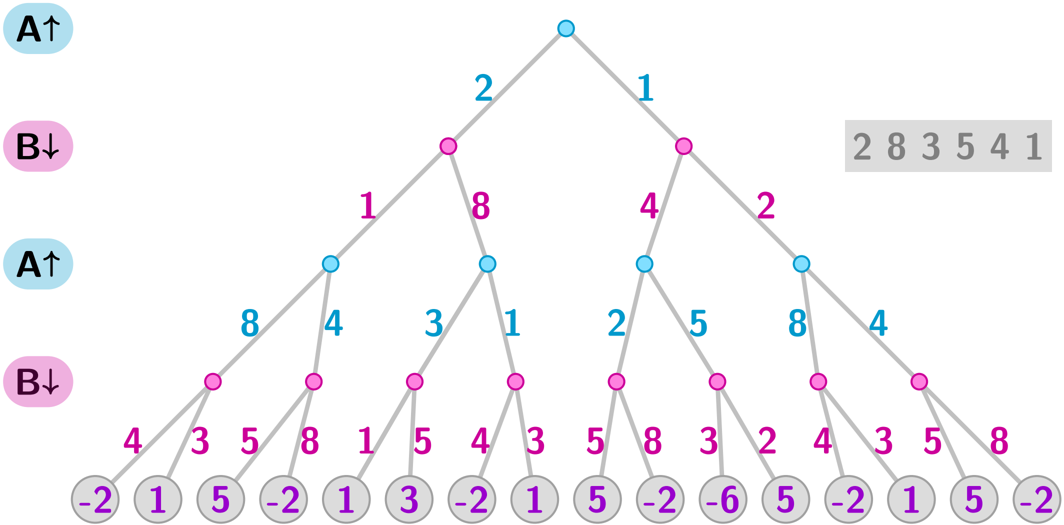
- Es sei eine Zahlenfolge geben, z. B. **2, 8, 3, 5, 4, 1**.
- Die Spieler **A** und **B** ziehen abwechselnd eine der beiden Zahlen vom Rand.
- Sobald nur noch zwei Zahlen übrig sind, wird die Differenz berechnet.
- Ziel von **A** ist, dass die Differenz groß und von **B**, dass sie klein ist.
- Mögliche Spielfolge für **2**, **8**, **3**, **5**, **4**, **1**,

  Spieler **A** ↑: **2** **3**

  Ergebnis: **5 - 4 = 1**

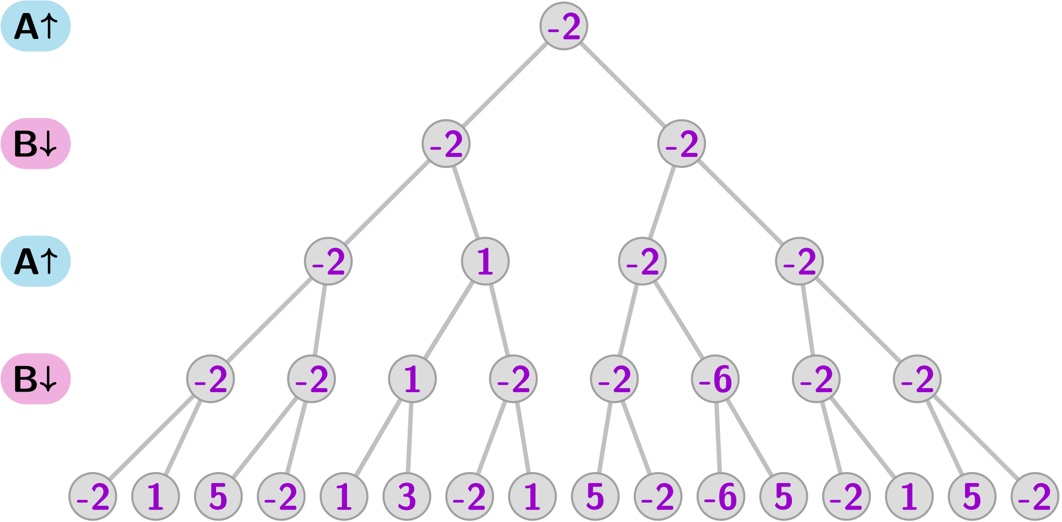
  Spieler **B** ↓: **8** **1**
- Kann **A** mehr Punkte erreichen? Gibt es eine Strategie für **B**, die ein kleineres Ergebnis erzielt?

Minimax Suchbaum für *BestDifference*



Wir nehmen die Spielzüge weg und propagieren die Bewertungen nach oben.

Minimax Suchbaum für *BestDifference*



Implementation des *BestDifference* Spiels: Grundelemente

```
public class Game {
  private int[] values;
  public static final int[]
      moves = {-1, 1};
  private int first, last;

  Game(int[] values) {
    this.values = values;
    first = 0;
    last = values.length-1;
  }

  public boolean isFinal() {
    return last - first == 1;
  }

  public int score() {
    return values[first] - values[last];
  }
```

Implementation des *BestDifference* Spiels: Grundelemente

```
public class Game {
  private int[] values;
  public static final int[]
      moves = {-1, 1};
  private int first, last;

  Game(int[] values) {
    this.values = values;
    first = 0;
    last = values.length-1;
  }

  public boolean isFinal() {
    return last - first == 1;
  }

  public int score() {
    return values[first] - values[last];
  }
```

```
  public void doMove(int move) {
    if (move < 0) {
      first++;
    } else {
      last--;
    }
  }

  public void undoMove(int move) {
    if (move < 0) {
      first--;
    } else {
      last++;
    }
  }

}
```

Minimax Implementation für *BestDifference*

```
public class MinMaxBestDifference {
  Game game;

  MinMaxBestDifference(int[] values) {
    game = new Game(values);
  }

  public int scorePlayerA() {
    if (game.isFinal())
      return game.score();
    int maxScore = Integer.MIN_VALUE;
    for (int move : Game.moves) {
      game.doMove(move);
      int score = scorePlayerB();
      game.undoMove(move);
      if (score > maxScore)
        maxScore = score;
    }
    return maxScore;
  }
```

```
  public static void main(String[] args) {
    int[] values = {2, 8, 3, 5, 4, 1};
    MinMaxBestDifference bd =
        new MinMaxBestDifference(values);
    int bestScoreA = bd.scorePlayerA());
  }

  public int scorePlayerB() {
    if (game.isFinal())
      return game.score();
    int minScore = Integer.MAX_VALUE;
    for (int move : Game.moves) {
      game.doMove(move);
      int score = scorePlayerA();
      game.undoMove(move);
      if (score < minScore)
        minScore = score;
    }
    return minScore;
  }
}
```

Das Minimax-Suchverfahren im Kontrast zu *Backtracking*

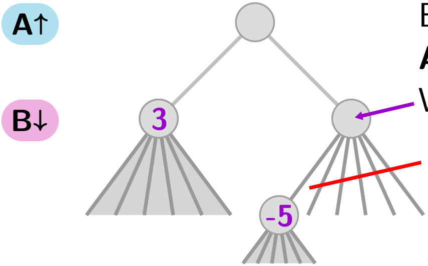
- Wie die Implementation zeigt, ist das Minimax-Suchverfahren sehr ähnlich zum *Backtracking*
- Als zusätzliches Grundelement kommt eine Bewertungsfunktion hinzu, hier `game.score()`.
- An Stelle des einfach rekursiven Aufrufs beim *Backtracking* steht beim Minimax der rekursiv alternierende Aufruf der gegenläufigen Auswahlverfahren.
- Die Minimax Methoden geben die Bewertung der jeweiligen Spielposition zurück während es beim *Backtracking* die Lösbarkeit ist.

Erweiterungen des Minimax-Suchverfahrens

- Bei komplexeren Spielen, wie Schach, können die Zugkombinationen nicht bis zum Spielende durchgerechnet werden.
- Daher wird Minimax nur bis zu einer bestimmten Zugtiefe durchgeführt.
- Dann wird die Güte der erreichten Spielposition bewertet.
- Häufig wird dabei mit einer dynamischen Zugtiefe gearbeitet, z. B. bei der Ruhesuche (*Quiescence search*).
- Allerdings werden bei Minimax viele überflüssige Auswertungen gemacht.
- Diese Beobachtung führt zu dem deutlich effizienteren Verfahren der Alpha-Beta-Suche.

Motivation der Alpha-Beta-Suche

- Die **Alpha-Beta-Suche** (*alpha-beta pruning*) basiert auf der folgenden Überlegung.
- Erste Zugmöglichkeit für **A** ergibt Bewertung 3.
- Bei nächster Zugmöglichkeit für **A** ergibt die erste Erwiederung von **B** Wert -5.
- In dieser Situation braucht **A** keine anderen Züge von **B** zu betrachten.



Bewertung kann aus Sicht von **A** nicht besser als -5 werden: Wert = $\min\{-5, \ldots\} \leq -5$

Weitere Züge brauchen nicht betrachtet zu werden.

Motivation der Schranken Alpha und Beta

- Im Beispiel stellt der Wert 3 eine **untere Schranke** dar:
- **A** hat eine Zugmöglichkeit gefunden, die mindestens die Bewertung 3 sichert. Es kann besser werden, falls **B** nicht optimal spielt.
- Züge von **A**, für die es eine Erwiderung von **B** gibt, die zu einer Bewertung unterhalb der Grenze führt, können verworfen werden.
- Für diese untere Schranke wird α verwendet und
- für die analoge obere Schranke in den Überlegungen von **B** die Variable β.

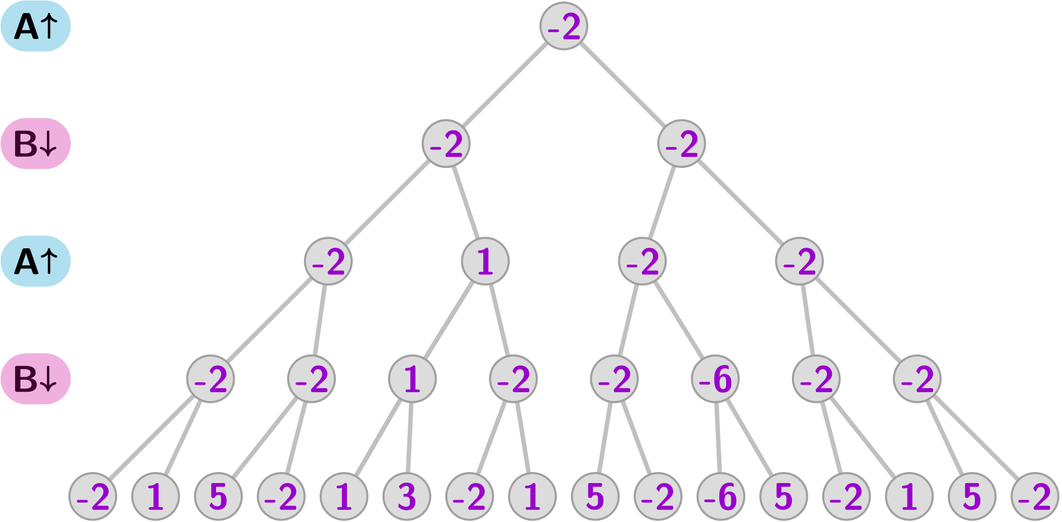
Das Alpha-Beta Intervall

- Knoten im Alpha-Beta-Suchbaum werden mit einem Intervall $[\alpha\ \beta]$ markiert. Es bedeutet: Bei der bisherigen Suche wurde gefunden:
  - für **A** eine Zugvariante mit Wert α,
  - für **B** eine Zugvariante mit Wert β
- Für die Suche unterhalb eines Knotens $[\alpha\ \beta]$ gilt:
- Findet **A** einen Zug mit Wert $\geq \beta$ ist: Abbruch (*beta-cutoff*).
  - **B** würde den zuvor gefunden Zug mit Wert $\leq \beta$ vorziehen.
  - Bei Gleichheit kann auch abgebrochen werden.
- Die analoge Überlegung für **B** führt zum *alpha-cutoff*.

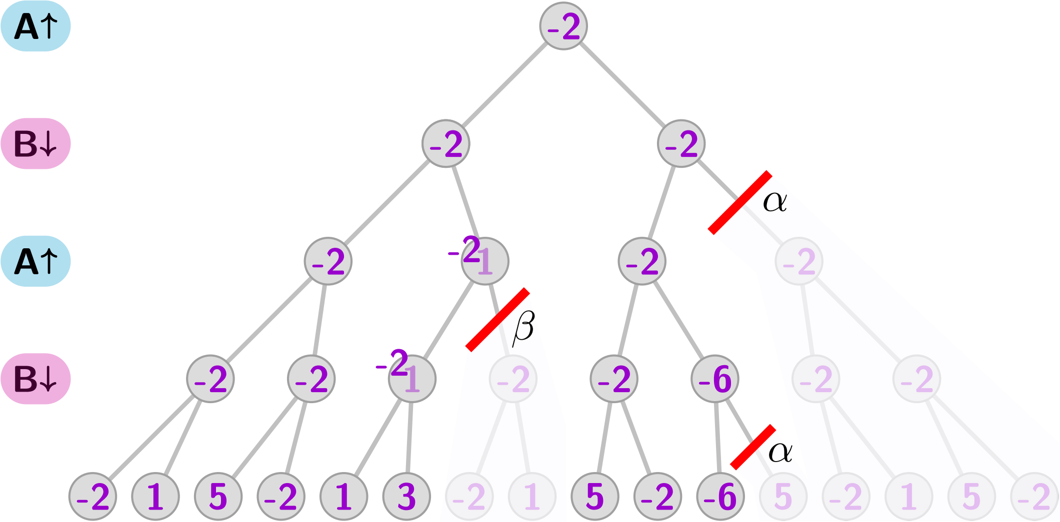
Illustration zum Alpha-Beta Intervall

- Für die abgeschnittenen Äste gilt:
  - mit Bewertung ≥ -5 würde **B** sie nicht wählen
  - mit Bewertung < 3 würde **A** nicht in diesen Zweig gehen

Minimax Suchbaum für das *BestDifference* Beispiel



Suchbaum mit Alpha/Beta Cutoff



Alpha-Beta Implementation für *BestDifference*

```
public int scorePlayerA(int alpha,
                        int beta) {
  if (game.isFinal()) {
    return game.score();
  }
  for (int move : Game.moves) {
    game.doMove(move);
    int score = scorePlayerB(alpha,
                             beta);
    game.undoMove(move);
    if (score > alpha) {
      alpha = score;
      if (alpha >= beta) break;
    }
  }
  return alpha;
}
```

Alpha-Beta Implementation für *BestDifference*

```
public int scorePlayerA(int alpha,
                        int beta) {
  if (game.isFinal()) {
    return game.score();
  }
  for (int move : Game.moves) {
    game.doMove(move);
    int score = scorePlayerB(alpha,
                             beta);
    game.undoMove(move);
    if (score > alpha) {
      alpha = score;
      if (alpha >= beta) break;
    }
  }
  return alpha;
}
```

```
public int scorePlayerB(int alpha,
                        int beta) {
  if (game.isFinal()) {
    return game.score();
  }
  for (int move : Game.moves) {
    game.doMove(move);
    int score = scorePlayerA(alpha,
                             beta);
    game.undoMove(move);
    if (score < beta) {
      beta = score;
      if (beta <= alpha) break;
    }
  }
  return beta;
}
```

Negamax Variante der Alpha-Beta Implementierung

- Der Code der Methoden `scorePlayerA()` und `scorePlayerB()` ist fast gleich.
- Die Methoden rufen sich wechselseitig auf.
- Für **A** maximieren und α Update, für **B** minimieren und β Update.
- In der **Negamax-Variante** werden beide Methoden zu einer vereint.
- Dafür wird die Bewertung so verändert, dass beide Spieler maximieren, jeweils aus ihrer Sicht. Die Bewertungsfunktion braucht dann also `player` als Argument.
- Die `negamax` Methode ruft sich selbst auf und führt dabei eine Negation der Spielbewertung durch und vertauscht die Rollen von α und β.

Negamax Implementation für *BestDifference*

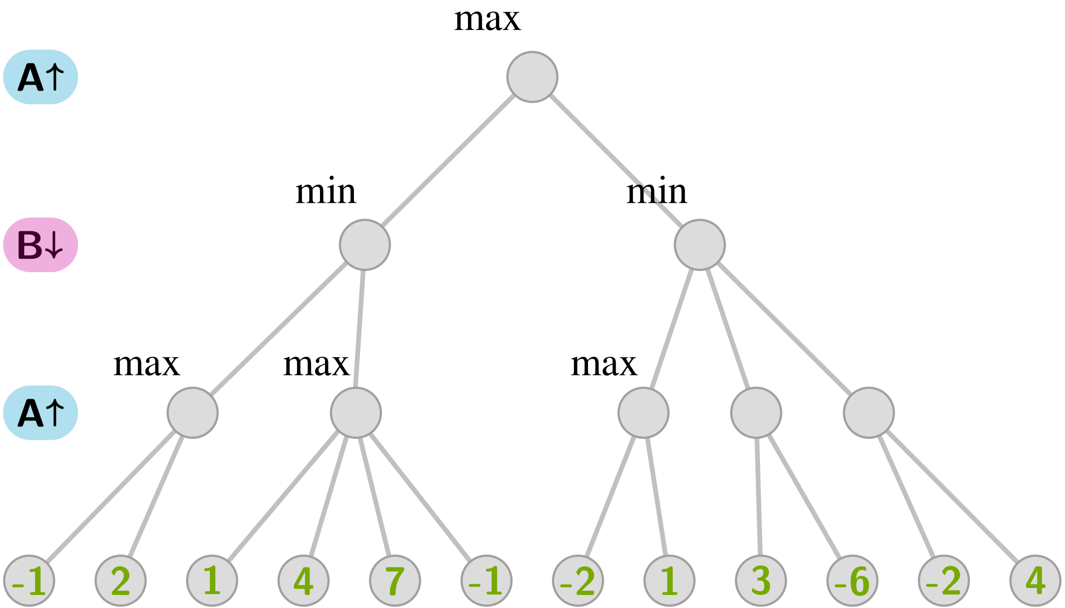
```
1  public int alphaBetaNegamax() {
2    return scorePlayer(1, -Integer.MAX_VALUE, Integer.MAX_VALUE);
3  }
4
5  public int scorePlayer(int player, int alpha, int beta) {
6    if (game.isFinal()) {
7      return player * game.score();
8    }
9    for (int move : Game.moves) {
10     game.doMove(move);
11     int score = - scorePlayer(-player, -beta, -alpha);
12     game.undoMove(move);
13     if (score > alpha) {
14       alpha = score;
15       if (alpha >= beta) break;
16     }
17   }
18   return alpha;
19 }
```

(Preisfrage: Warum `-Integer.MAX_VALUE` an Stelle von `Integer.MIN_VALUE` in Zeile 3?

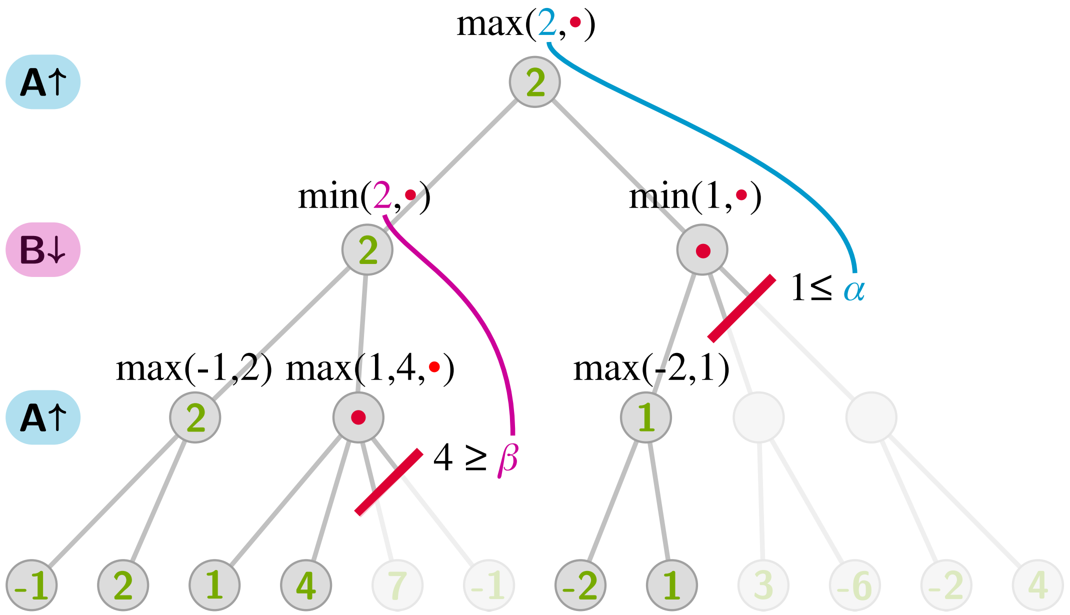
Bemerkungen

- Dynamische Programmierung (nächste Vorlesung) kann *BestDifference* deutlich effizienter lösen.
- Alpha-Beta Suche hat normalerweise einen Parameter `depth` für die Suchtiefe (siehe oben).
- Alpha-Beta Suche kann eine erhebliche Einschränkung des Suchraums bewirken.
- Die Einsparung hängt stark von der Reihenfolge der Zugmöglichkeiten ab.
- Wenn früh ein günstiger Zug gefunden wird, kann effektiver beschnitten werden.
- Es können z. B. vorher sehr gut bewertete Spielzüge an anderer Stelle vorrangig untersucht werden (Killer-Heuristik). Diese Übertragung von Spielzügen ist nur bei manchen Spielen anwendbar.

Ein weiteres Beispiel zur Alpha-Beta Suche



Ein weiteres Beispiel zur Alpha-Beta Suche



Branch-and-Bound

Branch-and-Bound

- Der ***Branch-and-Bound*** Ansatz generiert wie *Backtracking* Teillösungen in einer Baumstruktur, um eine optimale Lösung in einem Optimierungsproblem zu finden.
- Bei der folgenden Darstellung gehen wir von einem Maximierungsproblem aus. Minimierungsprobleme können mit entsprechenden Anpassungen analog behandelt werden.
- Zusätzlich zum *Backtracking*: stoppt die Suche bei Teillösungen, die zu keiner optimalen Lösung führen können.
- Dazu muss man wissen, wie gut sich eine Teillösung bestenfalls entwickeln kann.
- Wenn dies nicht besser ist, als die beste schon bekannte Lösung, braucht man die Teillösung nicht weiterzuverfolgen.

Die *Bounds* beim *Branch-and-Bound* (bei Maximierung)

- Essenziell: Methode, die zu einer gegebenen Teillösungen einen ***Upper Bound*** bestimmt.
- Fortsetzungen dieser Teillösung können keinen größeren Wert erzielen.
  Der *Upper Bound* ist also eine obere Schranke für die Güte.
- Wurde eine Lösung mit Wert c_0 gefunden, brauchen alle Teillösungen mit *Upper Bound* $\leq c_0$ nicht weiterverfolgt zu werden. Sie können nicht zu besseren Lösungen führen.
- Der Wert einer gefundenen Lösung stellt somit einen ***Lower Bound*** dar.
  Der gesuchte Maximalwert ist mindestens so groß.
- Es werden nur Teillösungen weiterverfolgt, deren *upper bound* über dem aktuellen *lower bound* liegt.

Branch-and-Bound mit Initiallösung

- Der *Bound* greift also erst, nachdem die erste Lösung gefunden wurde.
- Bei manchen Problem kann man schnell eine (meist suboptimale) Lösung generieren, z. B. mit einem Greedy-Algorithmus.
- Der Wert dieser Initiallösung kann als *Lower Bound* für *Branch-and-Bound* dienen.
- Teillösungen, deren *Upper Bound* $\leq$ dem *Lower Bound* ist, werden verworfen.
- Bemerkung: Der *Branch-and-Bound* Ablauf fängt mit der leeren Teillösung an. Die Initiallösung wird nur für den Startwert des *Lower Bound* verwendet.

Branch-and-Bound für Maximierung im Pseudocode

```
Bestimme eine Initiallösung
Setze Lower Bound auf den Wert dieser Lösung (oder −Inf)
Rekursion
    Falls Teillösung eine Lösung darstellt:
        Falls Lösung den bisher höchsten Wert hat: speichern
        Lösungswert ist neuer Lower Bound
    Generiere Kandidaten für nächsten Schritt in aktueller Teillösung
    Für jeden Kandidaten:
        Führe Schritt aus
        Berechne Upper Bound der erweiterten Teillösung
        Falls dieser über dem Lower Bound liegt
            Gehe in die Rekursion für den nächsten Schritt
        Mache Schritt rückgängig
```

Elemente des *Branch-and-Bound* Ansatzes

Folgende Elemente werden beim *Branch-and-Bound* zusätzlich zu den Elementen des *Backtracking* benötigt:

1. ***Branching:*** Verzweigung im Baum der Teillösungen, als Teil davon
   - Reihenfolge der Entscheidungen beim *branching*
2. ***Bounding:*** Bestimmen von oberen Schranken für die aktuelle Teillösung: Wie gut können resultierende Lösungen bestenfalls sein.
3. ***Lower Bound:*** Wird mit `-Inf` oder dem Wert einer Initiallösung initialisiert, und während des Ablaufs aktualisiert, wenn eine Lösungen mit höherem Wert gefunden wurde.

- Das *branching* ist auch Teil des *Backtracking*, aber hier kommt dem Punkt eine größere Bedeutung zu (Wichtigkeit der Auswahlreihenfolge für *Bounds*).

Beispiel: *Branch-and-Bound* für das 0/1-Rucksackproblem

- Baum der Teillösungen: in Knoten in der k-ten Ebene wird entschieden, ob der k-te Gegenstand ausgewählt wird.

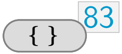
- Der *Upper Bound* einer Teillösung wird als Greedy Lösung des teilbaren Rucksackproblems für die verbleibenden Gegenstände bestimmt. (**2**)

  - Wurden in der Teillösung schon Entscheidungen für die ersten k Gegenstände getroffen, so wird des teilbare Rucksackproblem für die Gegenstände ab $k+1$ betrachtet. Dabei wird als Kapazität, die Restkapazität der Teillösung gesetzt.

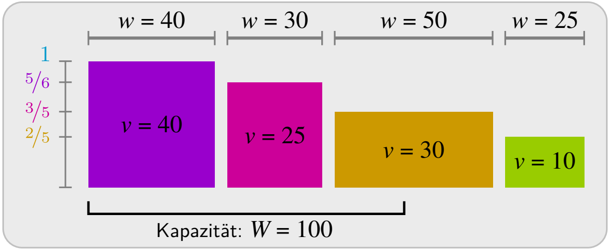
- Für den Greedy-Algorithmus, müssen die Gegenstände also im *Branch-and-Bound* Verfahren absteigend nach ihrem relativen Wert sortiert werden. (**1**)

- Als Initiallösung für den *Lower Bound* nehmen wir die Greedy-Lösung, wobei der geteilte Gegenstand weggelassen wird. (**3**)

lower bound:
65

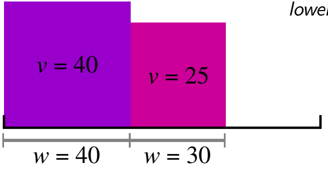


0/1 Rucksack Problemstellung



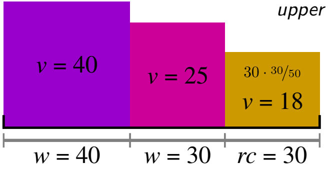
Untere Schranke durch Greedy 0/1 Lösung:

lower bound = **65**

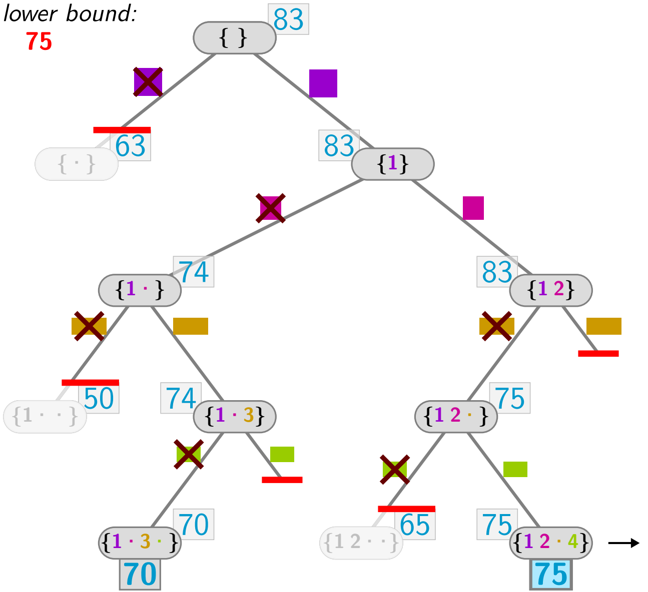


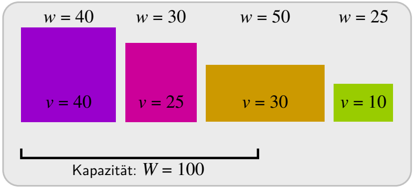
Obere Schranke durch Greedy *fractional* Lösung:

upper bound = **83**



lower bound: **75**





→ Die letzte Lösung, die den *lower bound* erhöht hat, ist eine optimale.

Fragen bei den Elementen des *Branch-and-Bound*

- ***Branching:*** Welche Entscheidung wird zur Verzweigung im Baum der Teillösungen verwendet?
  - Typische Entscheidungen sind "Objekt X wird ausgewählt oder nicht".
- **Auswahlreihenfolge:** In welcher Reihenfolge werden diese Entscheidungen getroffen?
  - Diese sollte so gewählt werden, dass die Wahrscheinlichkeit höher ist, früh gute Lösungen zu finden. (Sortierung beim 0/1 Rucksack)
- ***Bounding:*** Wie werden die Schranken für die Lösungswerte (*Bounds*) bestimmt?
  - *Bounds* kann man häufig dadurch erhalten, dass man Lösungen unter vereinfachten Bedingungen betrachtet. (0/1-Rucksack: Die obere Schranke wird unter Missachtung der Unteilbarkeit per Greedy Lösung des teilbaren Rucksackproblems bestimmt.)
- **Initiallösung:** Wie kann schnell (meist Greedy) ein korrekte, wenn gleich suboptimale Lösung generiert werden?

Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public class Item {
  public double value;
  public double weight;

  Item(double value, double weight) {
    this.value = value;
    this.weight = weight;
  }
}

// Comparator for sorting items suitable for the Greedy algorithm
public class ItemCompareRelativeValue implements Comparator<Item> {
  @Override public int compare(Item item1, Item item2)
  {
    return Double.compare(item1.value/item1.weight,
                          item2.value/item2.weight);
  }
}
```

Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public class Knapsack {
  private LinkedList<Item> inventory;
  protected double capacity;
  protected double residualCapacity;

  public Knapsack(double capacity) {
    this.inventory = new LinkedList<>();
    this.capacity = capacity;
    this.residualCapacity = capacity;
  }

  // copy constructor (omitted here)
  public Knapsack(Knapsack that) {...}

  public double value() {
    double knapsackValue = 0.0;
    for (Item item : inventory) {
      knapsackValue += item.value;
    }
    return knapsackValue;
  }
```

Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public class Knapsack {
  private LinkedList<Item> inventory;
  protected double capacity;
  protected double residualCapacity;

  public Knapsack(double capacity) {
    this.inventory = new LinkedList<>();
    this.capacity = capacity;
    this.residualCapacity = capacity;
  }

  // copy constructor (omitted here)
  public Knapsack(Knapsack that) {...}

  public double value() {
    double knapsackValue = 0.0;
    for (Item item : inventory) {
      knapsackValue += item.value;
    }
    return knapsackValue;
  }
```

```java
  public double weight() {
    double knapsackWeight = 0.0;
    for (Item item : inventory) {
      knapsackWeight += item.weight;
    }
    return knapsackWeight;
  }

  public void addItem(Item item) {
    inventory.push(item);
    residualCapacity -= item.weight;
  }

  public void removeItem() {
    Item item = inventory.pop();
    residualCapacity += item.weight;
  }
}
```

[Hier wurde eine `LinkedList` und kein `Stack` verwendet, da sich so der (nicht gezeigte) Copy Constructor besser implementieren lässt.]

Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public class BranchAndBoundKnapsack {
  private Item[] items;
  private double capacity;
  Knapsack knapsack;
  Knapsack optKnapsack;

  public BranchAndBoundKnapsack(Item[] items, double capacity) {
    this.items = items;  // Side effect: changing input object!
    Arrays.sort(items, Collections.reverseOrder(new ItemCompareRelativeValue()));
    this.capacity = capacity;
  }

  public Knapsack initialSolution() {
    Knapsack initialKnapsack = new Knapsack(capacity);
    for (Item item : items) {
      if (item.weight < initialKnapsack.residualCapacity) {
        initialKnapsack.addItem(item);
      }
    }
    return initialKnapsack;
  }
```

Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public double upperBound(Knapsack knapsack, int startIndex) {
  double residualCapacity = knapsack.residualCapacity;
  double addedValue = 0;
  for (int k = startIndex; k < items.length; k++) {
    double weight = items[k].weight;
    if (weight <= residualCapacity) {
      addedValue += items[k].value;
      residualCapacity -= weight;
    } else {
      addedValue += items[k].value * residualCapacity / weight;
      break;
    }
  }
  return knapsack.value() + addedValue;
}

public double branchAndBound() {
  Knapsack initKnapsack = initialSolution();
  optKnapsack = initKnapsack;
  knapsack = new Knapsack(capacity);
  return pack(initKnapsack.value(), 0);
}
```

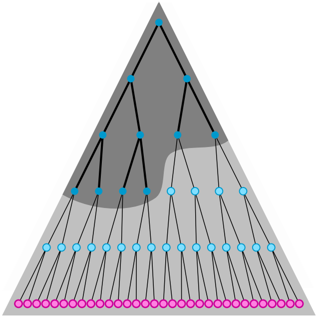
Branch-and-Bound Implementation für den 0/1 Rucksack

```java
public double pack(double lowerBound, int level) {
  if (level == items.length) {
    if (knapsack.value() > optKnapsack.value()) {
      optKnapsack = new Knapsack(knapsack);           // copy constructor!!
    }
    return knapsack.value();
  }
  if (upperBound(knapsack, level + 1) > lowerBound) {  // exclude item
    double value = pack(lowerBound, level + 1);
    if (value > lowerBound) lowerBound = value;
  }
  Item newItem = items[level];                          // include item
  if (newItem.weight <= knapsack.residualCapacity) {   // if it fits
    knapsack.addItem(newItem);
    if (upperBound(knapsack, level + 1) > lowerBound) {
      double value = pack(lowerBound, level + 1);
      if (value > lowerBound) lowerBound = value;
    }
    knapsack.removeItem();
  }
  return lowerBound;
}
```

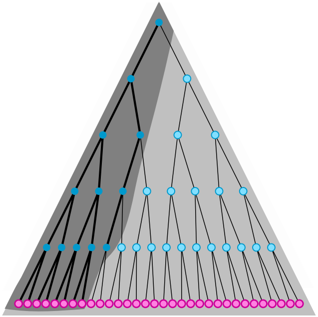
Branch-and-Bound and beyond

- Entscheidend für die Effektivität der *Bounds* ist es, dass frühzeitig möglichst gute Lösungen gefunden werden. Erst dann können die *Bounds* greifen.
- Der *Upper Bound* einer Teillösung gibt nur eine Beschränkung nach oben.
- Bei sinnvollen *Bounds* gibt es meist auch einen anderen Zusammenhang:
- Je höher der *Bound* einer Teillösung ist, desto besser ist die Chance darunter eine gute Lösung zu finden.
- Dies motiviert die Strategie von der starren Suchreihenfolge der Tiefensuche Abstand zu nehmen, und immer diejenige Teillösung weiterzuverfolgen, die unter allen bekannten den größen *Bound* besitzt.
- Das ist ein Ansatz, der bei den Heuristischen Algorithmen (Vorlesung #11) systematisiert wird und z. B. zu dem A*-Algorithmus führt.

Breitensuche

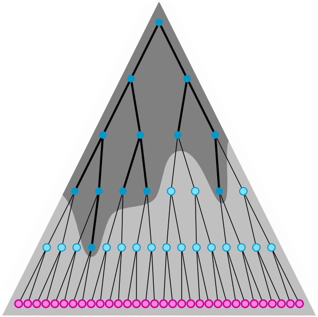


Tiefensuche



Heuristische Suche

(informierte Reihenfolge)



Implikation der Suchreihenfolge für Implementierung

- Bei der Tiefensuche bewegt man sich immer nur zwischen Eltern und Kindknoten, entlang der Baumkanten.
- Sobald eine Teillösung entdeckt wurde, wird sie direkt 'besucht'.
- Daher reicht es ein Objekt für die aktuelle Teillösung zu speichern. Der Übergang geht mit "Schritt ausführen" und "Schritt zurücknehmen".
- Bei der Breitensuche und der dynamischen Suche werden die Teillösungen auch quer zur Baumstruktur (nicht entlang der Kanten) durchlaufen.
- Man muss also die entdeckten Teillösungen speichern. So können sie später besucht werden, wenn sie nach der Suchreihenfolge dran sind.
- Zur Speicherung der Teillösungen bei einer Breitensuche verwenden wir eine `Queue`.

Implementation einer Klasse zur Speicherung der Teillösungen

```java
public class PartialSolution {
  public Knapsack knapsack;
  public int level;

  PartialSolution(Knapsack knapsack, int level) {
    this.knapsack = knapsack;
    this.level = level;
  }
}
```

Branch-and-Bound Implementation mit eine Queue (bfs)

```java
public Knapsack pack() {
  Knapsack initKnapsack = initialSolution();
  double maxValue = initKnapsack.value();
  Knapsack optKnapsack = initKnapsack;

  Queue<PartialSolution> queue = new LinkedList<>();
  queue.add(new PartialSolution(new Knapsack(capacity), 0));
  while (!queue.isEmpty()) {
    PartialSolution psol = queue.poll();
    if (psol.knapsack.value() > maxValue) {
      maxValue = psol.knapsack.value();
      optKnapsack = psol.knapsack;
    }
    if (psol.level < items.length) {
      if (bound(psol.knapsack, psol.level+1) > maxValue)
        queue.add(new PartialSolution(psol.knapsack, psol.level+1));
      Item newItem = items[psol.level];
      if (newItem.weight <= psol.knapsack.residualCapacity) {
        Knapsack knapsack = new Knapsack(psol.knapsack);
        knapsack.addItem(newItem);
        if (bound(knapsack, psol.level+1) > maxValue)
          queue.add(new PartialSolution(knapsack, psol.level+1));
      }
    }
  }
  return optKnapsack;
}
```

Branch-and-Bound Implementation mit einem Stack (dfs)

```java
public Knapsack pack() {
  Knapsack initKnapsack = initialSolution();
  double maxValue = initKnapsack.value();
  Knapsack optKnapsack = initKnapsack;

  Stack<PartialSolution> stack = new Stack<>();
  stack.push(new PartialSolution(new Knapsack(capacity), 0));
  while (!stack.isEmpty()) {
    PartialSolution psol = stack.pop();
    if (psol.knapsack.value() > maxValue) {
      maxValue = psol.knapsack.value();
      optKnapsack = psol.knapsack;
    }
    if (psol.level < items.length) {
      if (bound(psol.knapsack, psol.level+1) > maxValue)
        stack.push(new PartialSolution(psol.knapsack, psol.level+1));
      Item newItem = items[psol.level];
      if (newItem.weight <= psol.knapsack.residualCapacity) {
        Knapsack knapsack = new Knapsack(psol.knapsack);
        knapsack.addItem(newItem);
        if (bound(knapsack, psol.level+1) > maxValue)
          stack.push(new PartialSolution(knapsack, psol.level+1));
      }
    }
  }
  return optKnapsack;
}
```

Dynamische Suchreihenfolge nach *Bound*

- Darauf aufbauend, kann eine *Branch-and-Bound* Variante implementiert werden, bei der jeweils diejenige Teillösung bearbeitet wird, die den größten *Upper Bound* hat.
- Nach den Vorüberlegungen ist das die aussichtsreichste Teillösung. Daher sollte diese Variante mit den wenigsten Schritten zum Ziel führen.
- In der Implementation wird im wesentlichen die `Queue` durch eine `PriorityQueue` ersetzt.
- Dafür muss die Klasse der Teillösungen den `bound` als Attribut haben und `Comparable` gemäß dieses Bounds implementieren.
- In dieser Variante geschieht das Beschneiden des Suchbaumes indirekt. Teillösungen mit schlechtem *Bound* werden nicht aus der PQ geholt.
- Außerdem ist die zuerst gefundene Lösung immer eine optimale.
  Wenn der Bound halbwegs sinnvoll ist – mehr dazu in Vorlesung #11!

Implementation der Klasse Teillösung mit *Bound*

```java
public class PSolBound implements Comparable<PSolBound> {
  public Knapsack knapsack;
  public int level;
  private double bound;

  public PSolBound(Knapsack knapsack, Item[] items, int level) {
    this.knapsack = knapsack;
    this.bound = bound(knapsack, items, level);
    this.level = level;
  }

  public double bound(Knapsack knapsack, Item[] items, int startIndex) {
     // ... as before as method of BranchAndBoundKnapsack
  }

  @Override
  public int compareTo(PSolBound that) {
    return - Double.compare(this.bound, that.bound);
  }
}
```

Branch-and-Bound Implementation mit einer PriorityQueue

```java
public Knapsack pack() {
  PriorityQueue<PSolBound> queue = new PriorityQueue<>();
  queue.add(new PSolBound(new Knapsack(capacity), items, 0));
  while (!queue.isEmpty()) {
    PSolBound psol = queue.poll();
    if (psol.level == items.length) {
      return psol.knapsack;
    } else {
      queue.add(new PSolBound(psol.knapsack, items, psol.level+1));
      Item newItem = items[psol.level];
      if (newItem.weight <= psol.knapsack.residualCapacity) {
        Knapsack knapsack = new Knapsack(psol.knapsack);
        knapsack.addItem(newItem);
        queue.add(new PSolBound(knapsack, items, psol.level+1));
      }
    }
  }
  return null;
}
```

Bemerkungen zu den Implementationen

- Die Implementation mit `Queue` ist nur Gründen der Vollständikeit und für den Übergang zur `PriorityQueue` gegeben.
  - Die Breitensuche ergibt in den wenigsten Fällen eine sinnvolle Suchreihenfolge für *Branch-and-Bound*, da dann erst sehr spät die ersten vollständigen Lösungen gefunden werden und der *Bound* greifen kann.
- Die Implementation mit `Stack` ist in der Suchreihenfolge der rekursiven Lösung sehr ähnlich.
- Die rekursive Variante ist meist schneller.
- Allerdings ist die Größe des Rekursionsstacks beschränkt. Daher muss für größere Probleme die `Stack` Variante genutzt werden.
- Die Implementation mit `PriorityQueue` kann zu der Klasse der heuristischen Algorithmen gezählt werden, die in einer späteren Vorlesung besprochen werden.

Literatur I

Generell:

- Schöning U. *Algorithmik (Spektrum Lehrbuch).* Spektrum Akademischer Verlag; 2001. ISBN: 978-3827410924
- Skiena S. *The Algorithm Design Manual.* Springer; Auflage: 2nd ed. 2008. ISBN: 978-1848000698
- Ottmann T & Widmayer P. *Algorithmen und Datenstrukturen.* Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042

Anderes Vorlesungsmaterial:

- Mutzel P. *Datenstrukturen, Algorithmen und Programmierung 2*, Technische Universität Dortmund, SS 2009, `https://ls11-www.cs.tu-dortmund.de/people/beume/dap2-09/folien`

Index