Handlungsplanung und Allgemeines Spiel

„Spielen allgemeiner Spiele“

Peter Kissmann

Themen Allgemeines Spiel

- Einführung
- Game Desciption Language (GDL)
- Spielen allgemeiner Spiele
- Heuristiken im allgemeinen Spiel und Verbesserungen
- Lösen allgemeiner Spiele
- Instanziierung
- Ausblick: Unvollständige Information und Zufall

Spielen durch Suche

- Wie guten Zug finden?
  - Suchalgorithmen
  - recht einfach in Einpersonenspielen
  - auch in Zweipersonenspielen mit abwechselnden Zügen
  - sonst zunächst schwieriger
- Problem Suchalgorithmen:
  - Zustandsraumexplosion

Zustandsraumexplosion

- Solitär: 375 110 246
- Vier Gewinnt: 4,5 x 10^{12}
- (n^2-1)-Puzzle: n^2!/2
  - 8-Puzzle: 181 440
  - 15-Puzzle: 10^{13}
  - 24-Puzzle: 7,8 x 10^{24}
  - 35-Puzzle: 1,8 x 10^{41}

Was tun?

- Vollständige Suche schwierig
  - mit speziellen Datenstrukturen evtl. möglich → spätere Vorlesung
- Alternativen:
  - Suchraum beschneiden
    - Klassiker: Alpha-Beta Pruning (Zweipersonenspiel)
    - Schwieriger im allgemeinen Spiel: Wie Evaluationsfunktion bestimmen?
  - Zufällige Pfade absuchen
    - dabei möglichst gut führen lassen
    - Vorteil gegenüber Handlungsplanung: jeder Pfad endlich

Ablauf

- Suche mit Prolog
- Minimax-basiert
  - Minimax
  - Alpha-Beta
  - (Soft-)Max^n
- Simulationsbasiert
  - Monte-Carlo
  - UCT

Suche mit Prolog

- GDL logik-basierte Eingabe
  - sehr nah an Prolog
  - Übersetzung recht einfach
- Nutzung von Prologs Inferenzmechanismus, um mit Variablen klarzukommen

Grundlegende Funktionen

- findeLegals(role)
  - `setof(does(role, X), legal(role, X), Y).`
  - Menge aller gültigen Züge für Spieler role in Y gespeichert
  - aufsteigend sortiert
  - Duplikatsfrei
- simuliere(moves)
  - `assert(move`$_1$`), …, assert(move`$_n$`).`
  - `setof(true(X), next(X), Y).`
  - `retract(move`$_1$`), …, retract(move`$_n$`).`
  - Züge zur Wissensbasis hinzugefügt
  - Menge aller Fluents wahr im Nachfolgezustand in Y gespeichert
  - Zug aus Wissensbasis entfernt

Grundlegende Funktionen

- setzeZustand(fluents)
  - `abolish(true/1).`
  - `assert(true(fluent`$_1$`)), …, assert(true(fluent`$_m$`)).`
  - Fluents des alten Zustands werden entfernt
  - Übergebene Fluents in Wissensbasis eingefügt.

Grundlegende Funktionen

- prüfeTerminierung
  - `terminal.`
  - Resultat:
    - true, wenn erfüllbar, also Terminalzustand erreicht
    - false, sonst
- findeBewertung(role)
  - `goal(role, X).`
  - Bewertung für Spieler role in X

Aufbau Minimax-basierte Suche

- Zweipersonenspiele
  - Minimax
  - Negamax
  - Alpha-Beta Pruning
  - Probleme von Alpha-Beta Pruning
- Mehrpersonenspiele
  - Max^n
  - Alpha-Beta Pruning
  - Probleme von Alpha-Beta Pruning

Minimax

- Typisches Suchverfahren bei Zweipersonen-Nullsummenspielen
  - enspr. in GGP: Summe der Bewertungen beider Spieler immer 100
  - genügt, nur Bewertung eines Spielers zu betrachten
- Spieler 1 versucht, Bewertung von Spieler 1 zu maximieren

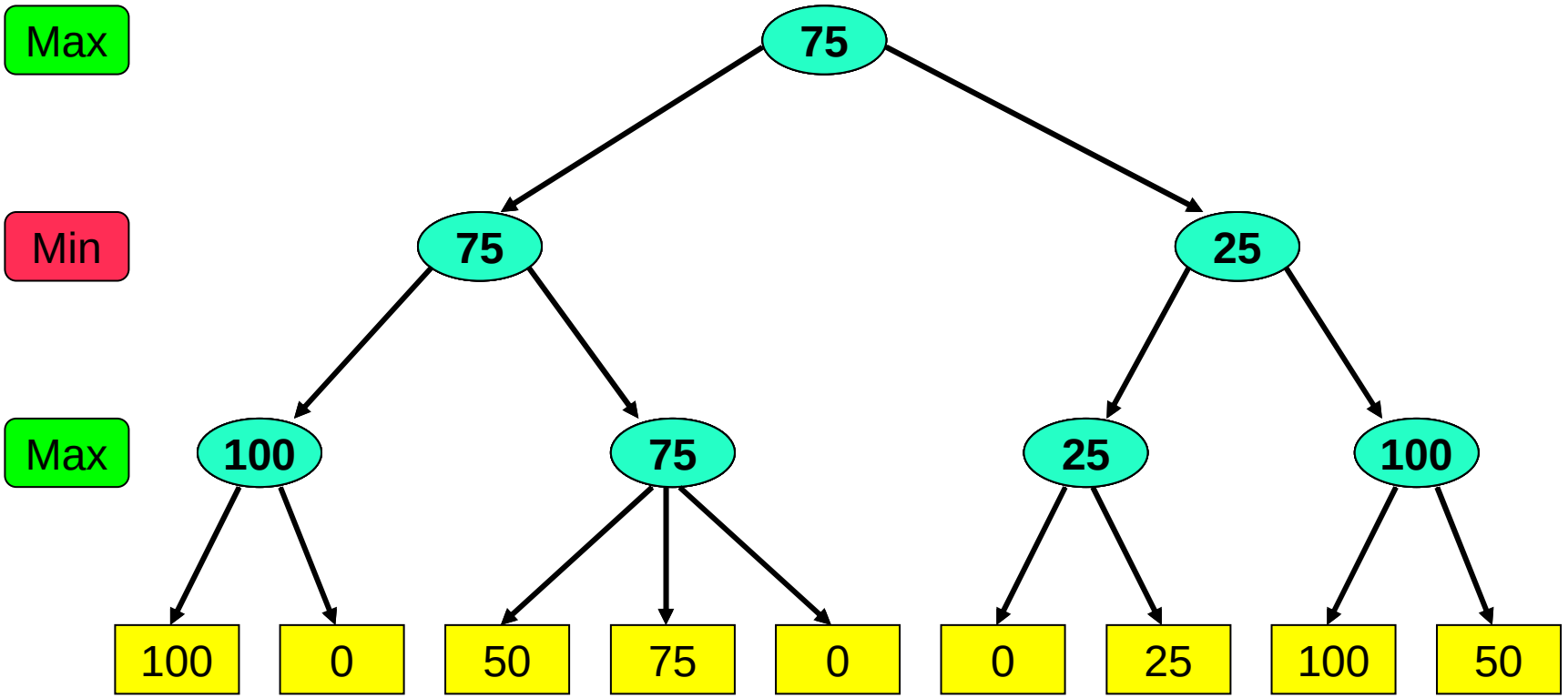
  → Max-Spieler
- Spieler 2 versucht, Bewertung von Spieler 1 zu minimieren

  → Min-Spieler

Minimax [Minimax(zustand)]

```
falls (prüfeTerminierung(zustand))
    gib findeBewertung(spieler1) zurück
falls (aktiv(zustand) = spieler1)
    bewertung(zustand) ← -∞
sonst
    bewertung(zustand) ← ∞
züge ← findeLegals(aktiv(zustand))
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    falls aktiv(zustand) = spieler1
        bewertung(zustand) ← max{bewertung(zustand), Minimax(nachfolger)}
    sonst
        bewertung(zustand) ← min{bewertung(zustand), Minimax(nachfolger)}
gib bewertung(zustand) zurück
```

Minimax



Minimax

- Probleme
  - gesamter Suchraum muss durchsucht werden
  - kann nicht abgebrochen werden

Negamax

- Äquivalent zu Minimax
- Unterschied: Spieler 2 wertet Spieler 1 Bewertung negativ
  - alternativ: Spieler 2 maximiert seine Bewertung
- Damit sind alle Knoten Max-Knoten
  - Maximierung der negierten Bewertung entspricht Minimierung der echten Bewertung

Negamax [Negamax(zustand)]

```
falls (prüfeTerminierung(zustand))
    falls (aktiv(zustand) = spieler1)
        gib findeBewertung(spieler1) zurück
    sonst
        gib -findeBewertung(spieler1) zurück
bewertung(zustand) ← -∞
züge ← findeLegals(aktiv(zustand))
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    bewertung(zustand) ← max{bewertung(zustand), -Negamax(nachfolger)}
gib bewertung(zustand) zurück
```

Alpha-Beta Pruning

- Nutzt Minimax als Basis
  - Formulierung über Negamax auch möglich
- Kann Teile des Suchraumes abschneiden
  - bleibt aber korrekt
- Jeder Knoten hat (α, β)-Fenster:
  - α: Mindestwert, den Spieler 1 erreichen kann
  - β: Spieler 2 kann Bewertung hierauf begrenzen (= Maximalwert für Spieler 1)
- initial:
  - normalerweise: (-∞, ∞)
  - hier: (0, 100)
    - weniger als 0 und mehr als 100 Punkte bei GDL unmöglich

Alpha-Beta Pruning

- α-Schnitte
  - Nachfolger von Min-Knoten mit Bewertung ≤ α
- β-Schnitte
  - Nachfolger von Max-Knoten mit Bewertung ≥ β

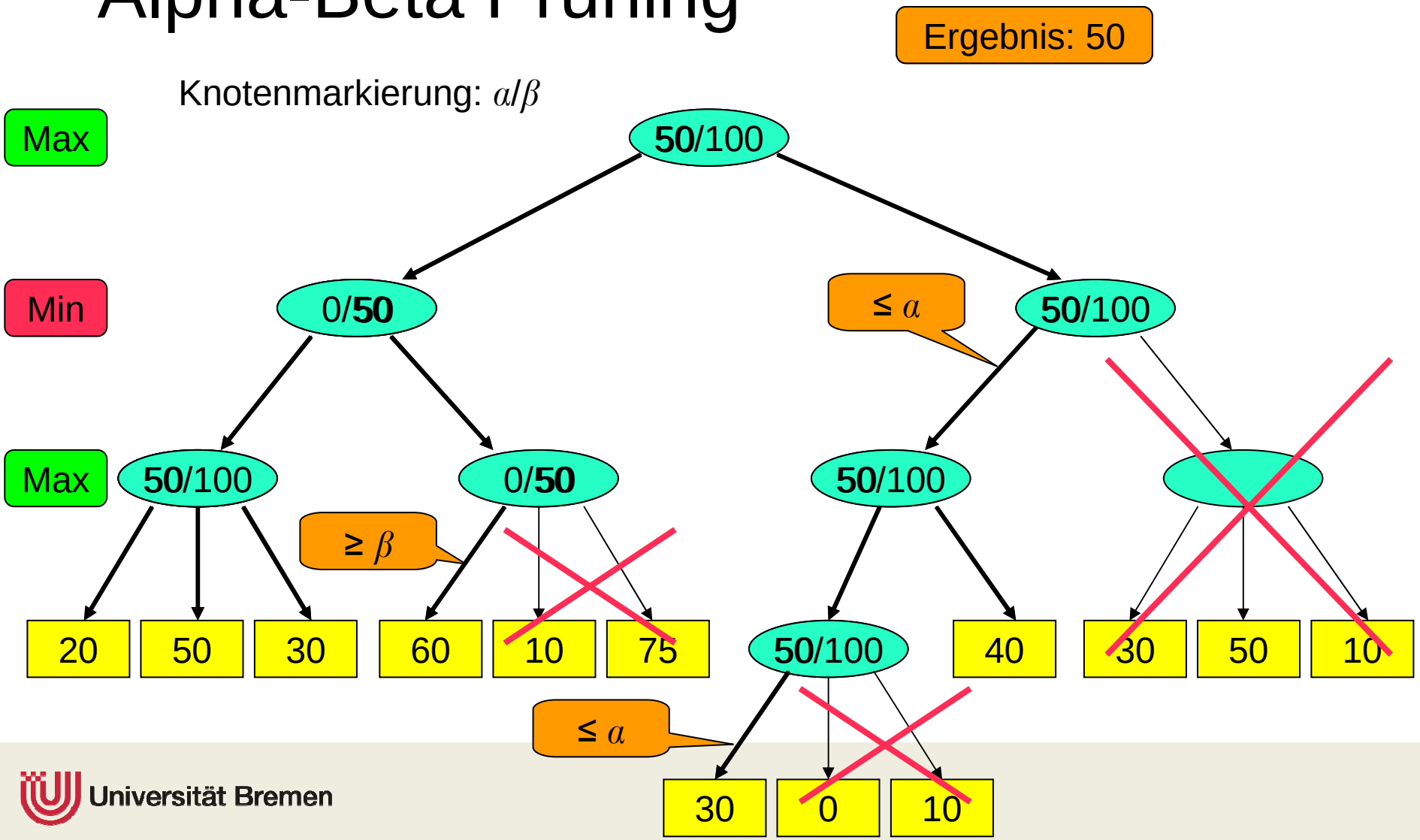
Alpha-Beta Pruning [Max(zustand, α, β)]

```
falls (prüfeTerminierung(zustand))
    gib findeBewertung(spieler1) zurück
züge ← findeLegals(spieler1)    // spieler1 aktiv in Max-Knoten!
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    bewertung ← Min(nachfolger, α, β)
    falls (bewertung ≥ β) gib β zurück    // β-Schnitt
    falls (bewertung > α) α ← bewertung
gib α zurück
```

Alpha-Beta Pruning [Min(zustand, α, β)]

falls (prüfeTerminierung(zustand))
    gib findeBewertung(spieler1) zurück
züge ← findeLegals(spieler2) // spieler2 aktiv in Min-Knoten!
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    bewertung ← Max(nachfolger, α, β)
    falls (bewertung ≤ α) gib α zurück // α-Schnitt
    falls (bewertung < β) β ← bewertung
gib β zurück

Alpha-Beta Pruning



Alpha-Beta Pruning [Negamax(zustand, α, β)]

falls (prüfeTerminierung(zustand))
    falls (aktiv(zustand) = spieler1)
        gib findeBewertung(spieler1) zurück
    sonst
        gib -findeBewertung(spieler1) zurück
züge ← findeLegals(aktiv(zustand))
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    bewertung ← -Negamax(nachfolger, -β, -α)
    falls (bewertung ≥ β) gib β zurück
    falls (bewertung > α) α ← bewertung
gib α zurück

Alpha-Beta Pruning

- Probleme:
  - oft immer noch zu viele Zustände betrachtet
- Lösung:
  - früher abbrechen (Tiefenschranke o.ä.)
  - Evaluationsfunktion zur Bewertung von nicht-terminalen Blättern
- Aber:
  - Wie Evaluationsfunktion bestimmen?

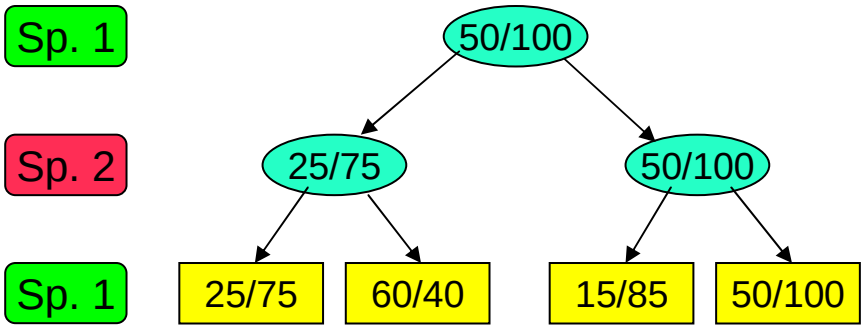
    → in nächster Vorlesung

Alpha-Beta Pruning

- Problem:
  - Nicht unterbrechbar
- Lösung:
  - Iterative Deepening
    - schnell erste Bewertungen
    - schrittweise tiefer suchen und Bewertungen präzisieren
    - uniform: Tiefenschranke iterativ erhöhen
    - nicht-uniform [Schiffel & Thielscher, 2007]:
      - maximale Tiefenschranke für besten Zug aus letzter Iteration
      - andere Züge (nach Bewertung) früher abschneiden
      - minimale Tiefenschranke für schlechteste Züge

Alpha-Beta Pruning

- Problem:
  - bei Nicht-Nullsummenspielen kein Pruning möglich, wenn beide Spieler eigenen Gewinn maximieren wollen



- Lösung:
  - pessimistische Sichtweise [Korf, 1989]:
    - Gegenspieler versucht, unseren Gewinn zu minimieren, unabhängig vom eigenen
- Aber:
  - kann zu suboptimalem Spiel führen

Alpha-Beta Pruning

- Problem:
  - Behandlung von simultanen Zügen
- 1. Lösung:
  - pessimistische Sichtweise [Schiffel & Thielscher, 2007]:
    - Wir führen unseren Zug aus, dann der Gegenspieler seinen
- Aber:
  - kann zu beliebig schlechtem Spiel führen
  - Beispiel: Schere-Stein-Papier
    - wenn Gegenspieler unseren Zug kennt, gewinnt er immer
- 2. Lösung:
  - randomisierte Gegenspieler [Kuhlmann, Dresner & Stone, 2006]:
    - Annahme: Gegenspieler wählt zufällig
    - basierend darauf: Unser Zug so, dass erwarteter Gewinn maximal

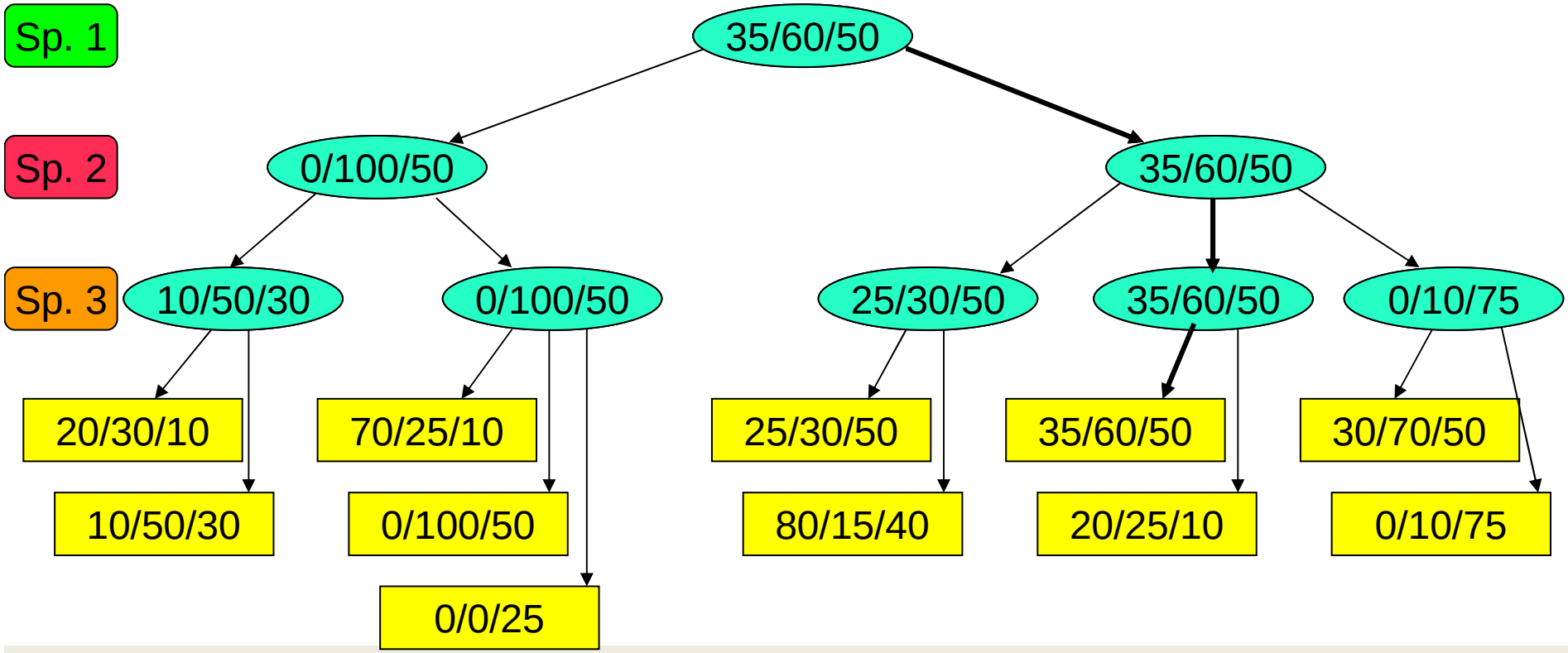
Max^n [Luckhardt & Irani, 1986]

- ähnlich Minimax / Negamax für n Spieler
  - jeder Spieler versucht, eigenen Gewinn zu maximieren
  - funktioniert nur bei nicht-simultanen Zügen
  - Bewertungen von Blättern zur Wurzel hochpropagiert
  - Verwendung von Iterative Deepening und Evaluationsfunktionen möglich

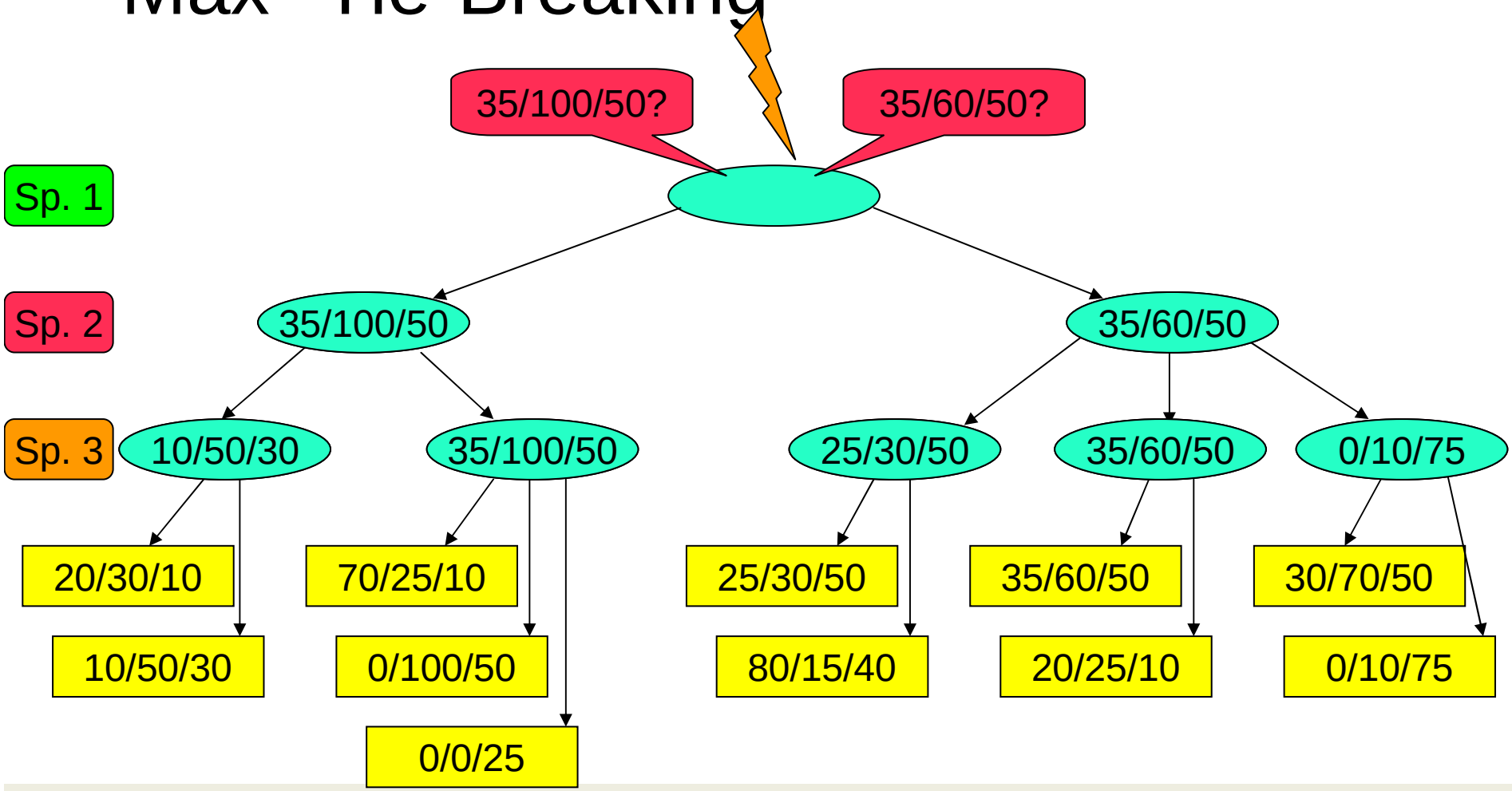
Max^n [Max-n(zustand)]

falls (prüfeTerminierung(zustand))
    gib [findeBewertung($spieler_1$), …, findeBewertung($spieler_n$)] zurück
bewertung(zustand) ← [-∞, …, -∞]
züge ← findeLegals(aktiv(zustand))
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    tmpBewertung ← Max-n(nachfolger)
    falls (tmpBewertung[aktiv(zustand)] > bewertung(zustand)[aktiv(zustand)]
        bewertung(zustand) ← tmpBewertung
gib bewertung(zustand) zurück

Max^n


Sp. 1
Sp. 2
Sp. 3
35/60/50
0/100/50
35/60/50
10/50/30
0/100/50
25/30/50
35/60/50
0/10/75
20/30/10
70/25/10
25/30/50
35/60/50
30/70/50
10/50/30
0/100/50
80/15/40
20/25/10
0/10/75
0/0/25

Max^n Tie-Breaking



Max^n

- Problem:
  - Tie-Breaking
- Lösung:
  - festgelegt auf ersten Nachfolger
    - damit bei Alpha-Beta besseres Pruning
- Aber:
  - warum sollte Spieler diesen Zug wählen?
  - suggeriert Sicherheit, die es gar nicht gibt

Soft-Max^n [Sturtevant & Bowling, 2006]

- statt Tie-Breaking:
  - propagiere alle möglichen Lösungen nach oben
    - Max^n-Mengen (Mengen von n-Tupeln)
    - möglich = nicht dominiert
  - evtl. immer noch eindeutig
  - keine Irreführung möglich
- Aber:
  - viel langsamer
  - was tun, wenn Bewertung nicht eindeutig?

Soft-Max^n: Dominanz

- Max^n-Menge s_1 dominiert s_2 stark bzgl. Spieler i gdw.

$$\forall v_1 \in s_1 \forall v_2 \in s_2 . v_1[i] > v_2[i]$$

- Max^n-Menge s_1 dominiert s_2 schwach bzgl. Spieler i gdw.

$$\forall v_1 \in s_1 \forall v_2 \in s_2 . v_1[i] \geq v_2[i] \text{ und}$$

$$\exists v_1 \in s_1 \exists v_2 \in s_2 . v_1[i] > v_2[i]$$

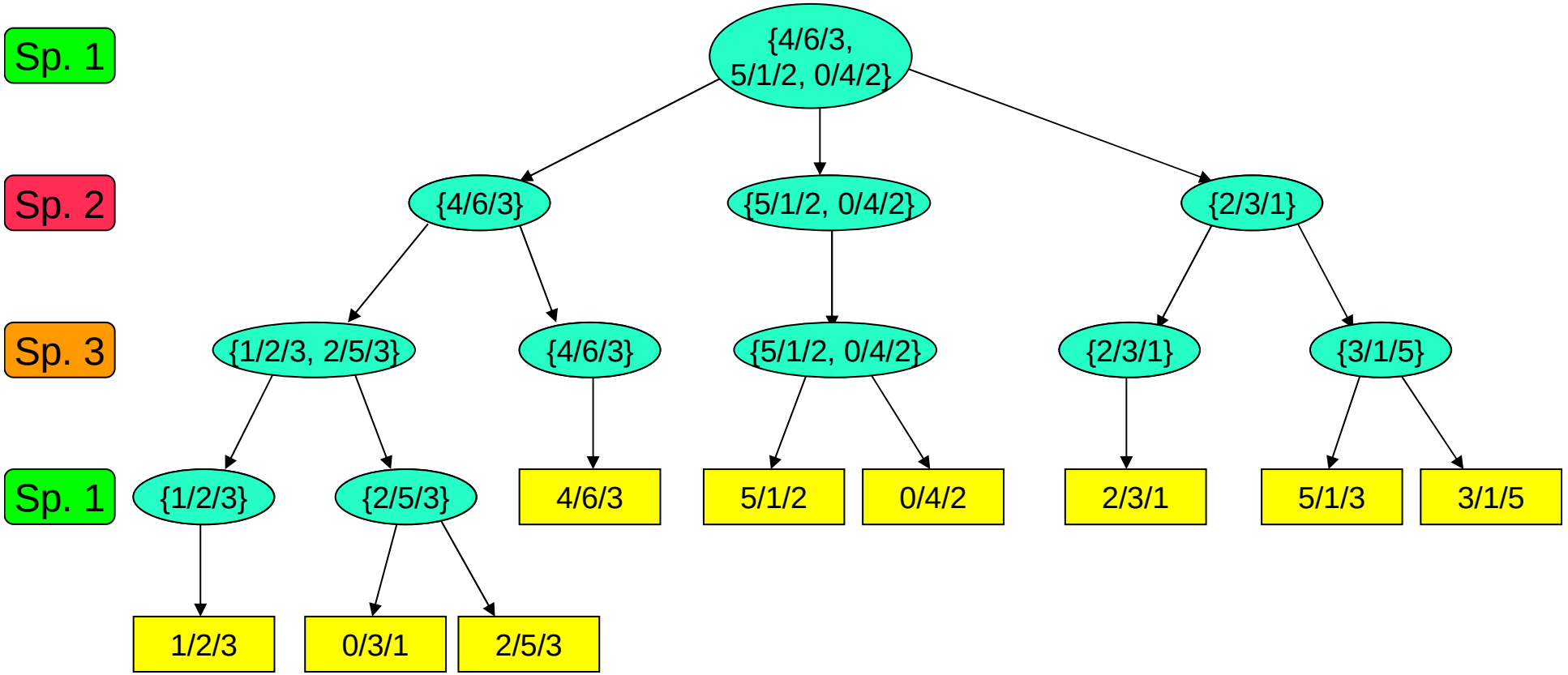
Soft-Max^n: Dominanz

- Vier Max^n-Mengen:
  1. {(3, 4, 3)}
  2. {(2, 1, 7), (4, 2, 4)}
  3. {(4, 1, 5), (3, 2, 5)}
  4. {(10, 0, 0), (0, 10, 0), (0, 0, 10)}
- Spieler1: 3. dominiert 1. schwach
- Spieler 2: 1. dominiert 2. und 3. stark
- Spieler 3: 2. und 3. dominieren 1. stark
- falls Gewinne in [0, 10]:
  - 4. kann keine Menge stark dominieren
  - 4. kann von keiner Menge stark dominiert werden

Soft-Max^n

- An Blatt, Max^n-Menge echte Bewertung (oder Evaluationsfunktion)
- An innerem Knoten (Spieler j am Zug), Max^n-Menge ist
  - Vereinigung aller Mengen der Kinder
  - die nicht stark dominiert bzgl. Spieler j sind
- An Wurzel, aktiver Spieler kann beliebige Entscheidungsregel anwenden, um beste Menge aus nicht-dominierten Züge zu wählen

Soft-Max^n


Sp. 1
Sp. 2
Sp. 3
Sp. 1
{4/6/3, 5/1/2, 0/4/2}
{4/6/3}
{5/1/2, 0/4/2}
{2/3/1}
{1/2/3, 2/5/3}
{4/6/3}
{5/1/2, 0/4/2}
{2/3/1}
{3/1/5}
{1/2/3}
{2/5/3}
4/6/3
5/1/2
0/4/2
2/3/1
5/1/3
3/1/5
1/2/3
0/3/1
2/5/3

Mehrpersonen Alpha-Beta Pruning

- nutzt Max^n als Basis
- Pruning nur möglich, wenn
  - Obere Schranke für Summe der Gewinne aller Spieler
  - Untere Schranke für Gewinn jedes Spielers
- ähnliche Bedingung wie bei Zweipersonenspielen (konstante Summe)
- [Korf, 1989]

Mehrpersonen Alpha-Beta Pruning

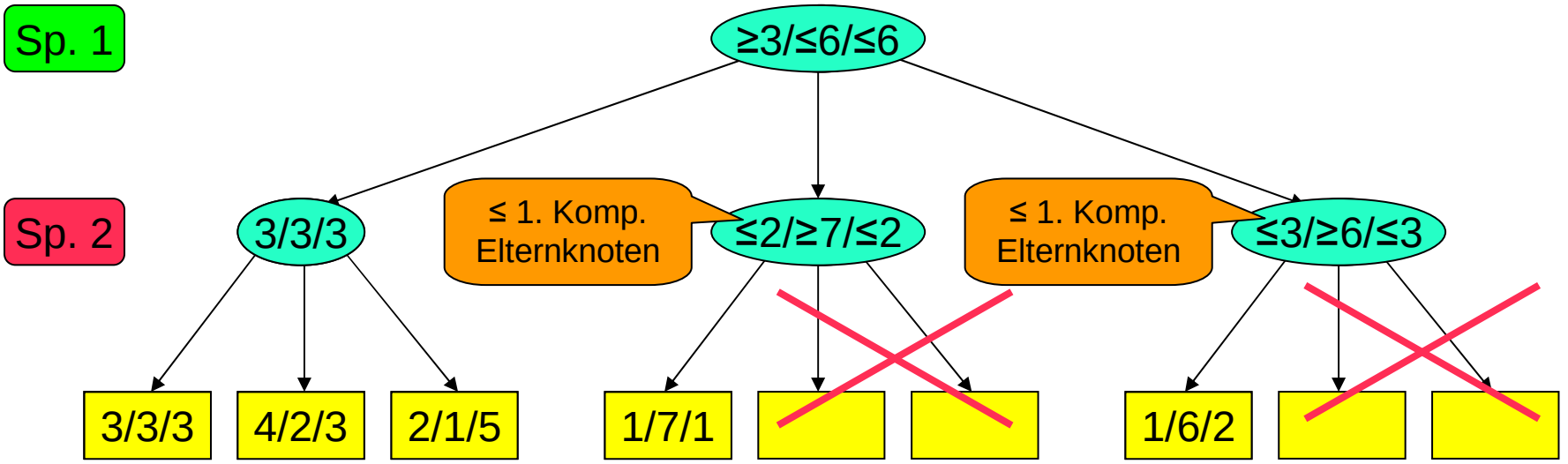
- sofortiger Schnitt
  - Spieler i am Zug
  - i-te Komponente eines der Kinder = oberer Schranke für Summe
  - Rest kann nicht besser sein → kann abgeschnitten werden

Mehrpersonen Alpha-Beta Pruning

- flacher Schnitt

Obere Schranke (Summe): 9
(hier sogar exakt 9)

Untere Schranke (Einzelwert): 0

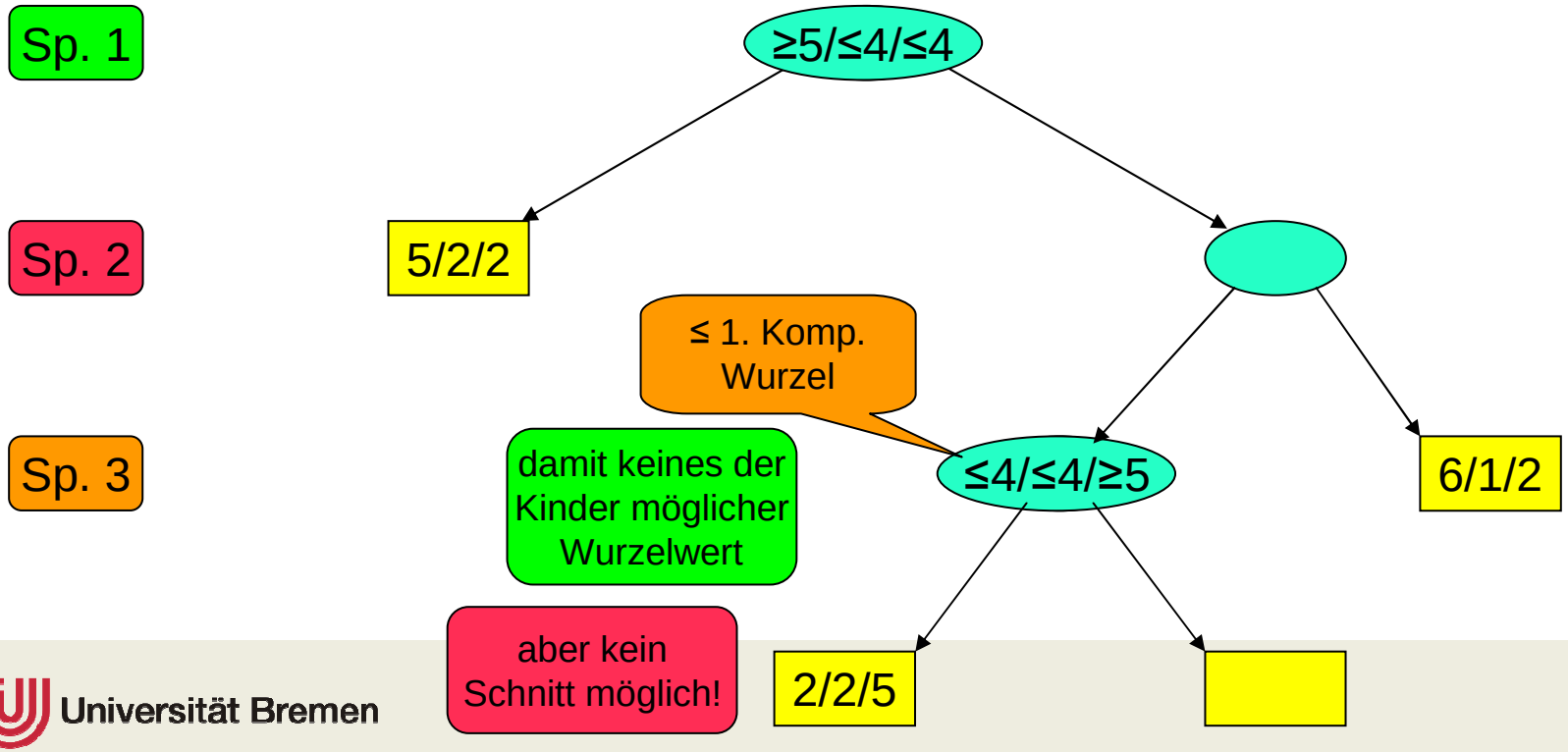


Mehrpersonen Alpha-Beta Pruning

- tiefer Schnitt

Obere Schranke (Summe): 9
(hier sogar exakt 9)

Untere Schranke (Einzelwert): 0

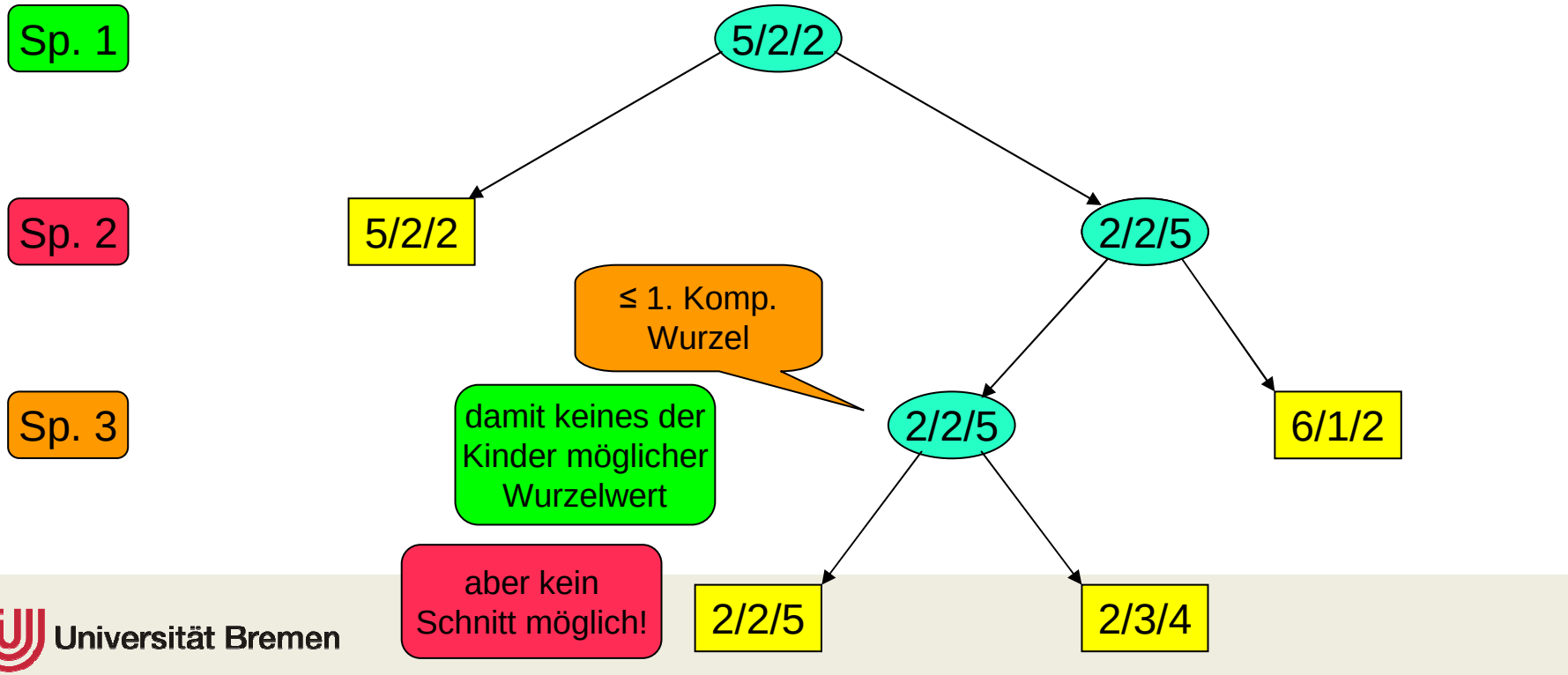


Mehrpersonen Alpha-Beta Pruning

- tiefer Schnitt

Obere Schranke (Summe): 9
(hier sogar exakt 9)

Untere Schranke (Einzelwert): 0

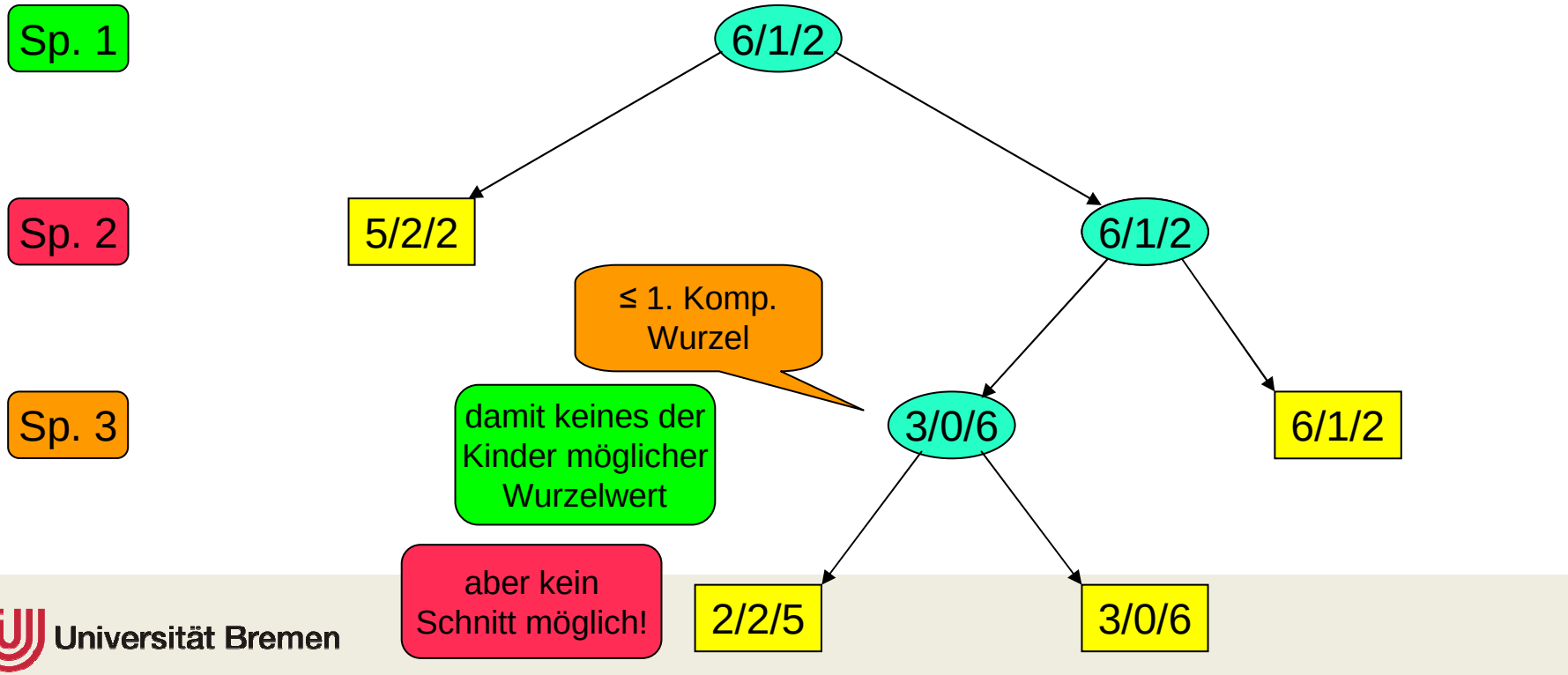


Mehrpersonen Alpha-Beta Pruning

- tiefer Schnitt

Obere Schranke (Summe): 9
(hier sogar exakt 9)

Untere Schranke (Einzelwert): 0



Mehrpersonen Alpha-Beta Pruning

- Problem:
  - keine tiefen Beschneidungen bei Mehrpersonenspielen
  - damit Alpha-Beta nur bedingt geeignet
- Lösung:
  - pessimistische Sichtweise [Kuhlmann, Dresner & Stone, 2006]:
    - alle nicht mit uns kooperierenden Spieler gegen uns
    - versuchen, unseren Gewinn zu minimieren, unabhängig vom eigenen
    - entspricht also Zweipersonenspiel (Alpha-Beta leichter möglich)
- Aber:
  - kann zu suboptimalem Spiel führen

Aufbau Simulationsbasierte Suche

- Monte-Carlo
- Monte-Carlo + Erinnerung
- UCT
- Besonderheiten bei Einpersonenspielen
- Erweiterungen für Mehrpersonenspiele

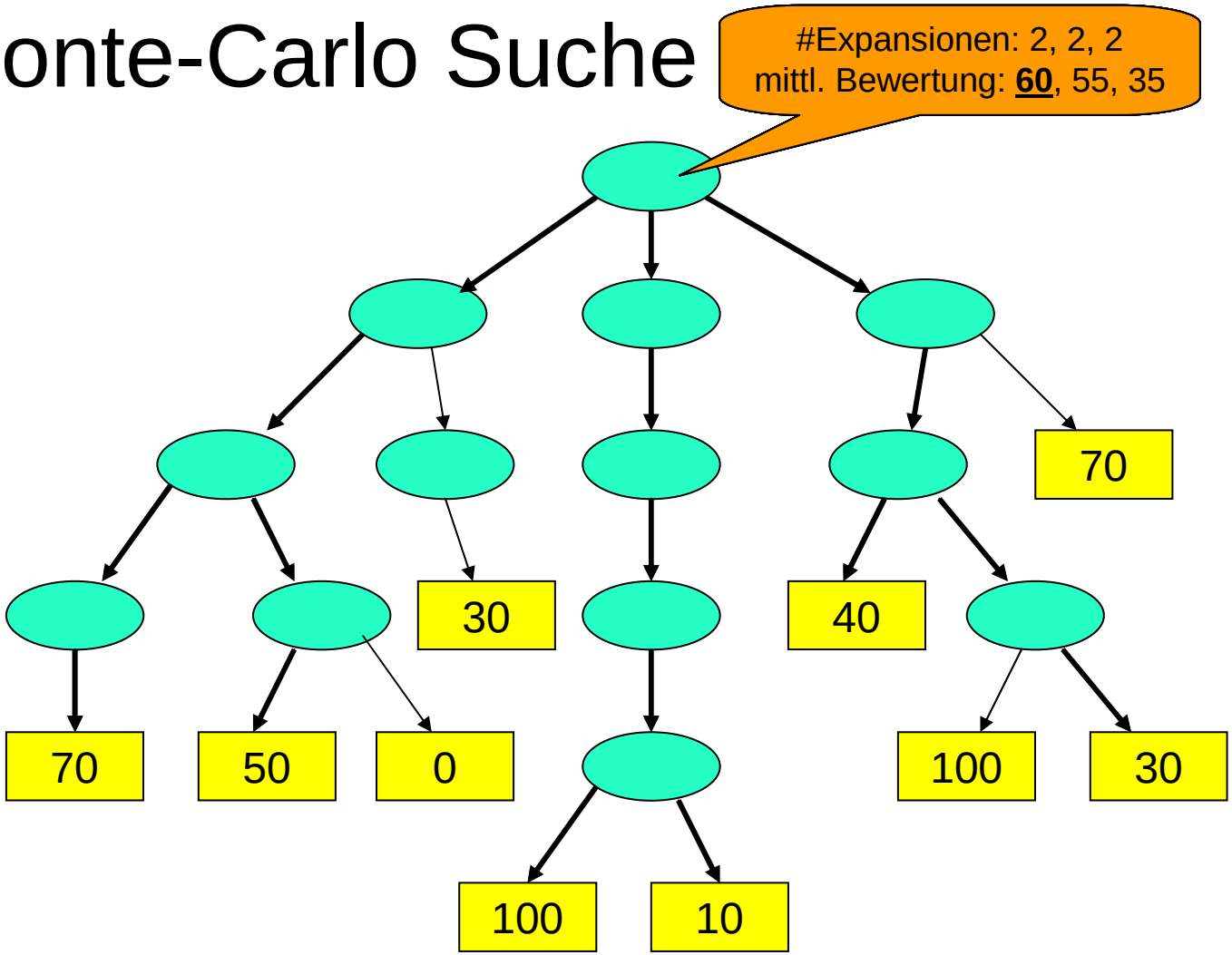
Monte-Carlo Suche

- Betrachtet Menge von zufälligen Pfaden vom Initialzustand zu einem Terminalzustand
- Setzt mittlere Bewertung für die Spieler für jeden gültigen Zug
- Keine weitere Steuerung, kein zusätzliches Wissen erarbeitet
- Am Ende: Wählt Zug mit bester Bewertung

Monte-Carlo Suche

- Endzeit = Jetzt + Playclock - 0,5 (etwas Sicherheit, um Zug rechtzeitig zu übertragen)
- speichere Initialzustand init
- solange (Jetzt != Endzeit)
  - solange (!prüfeTerminierung)
    - züge ← [findeLegals($spieler_1$), …, findeLegals($spieler_n$)]
    - für (i = 1 … n) wähle zug_i zufällig aus züge[i]
    - nachfolger ← simuliere(zug_1, …, zug_n)
    - setzeZustand(nachfolger)
  - aktualisiere mittlere Bewertung für eigenen Spieler bei erstem Zug entsprechend findeBewertung(selbst)
  - setzeZustand(init)
- gib Zug mit bestem mittleren Gewinn für eigenen Spieler aus

Monte-Carlo Suche



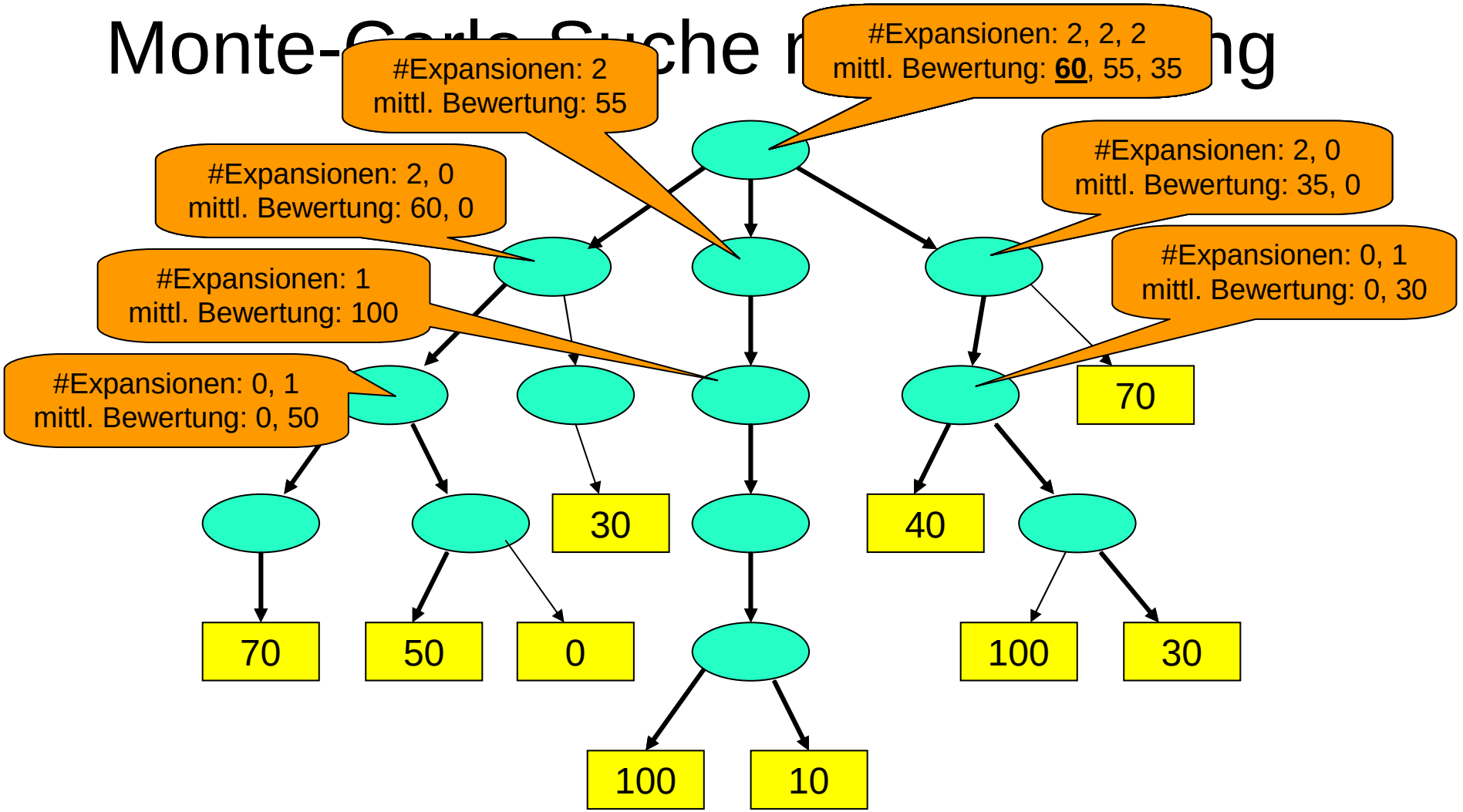
Monte-Carlo Suche

- Vorteile:
  - sehr einfach zu implementieren
  - besser als Random
  - sehr geringer Speicherbedarf
- Nachteile:
  - ständige Expansion der selben Zustände
    - verhältnismäßig langsam
  - ungesteuerte Suche
  - Ergebnisse potenziell schlecht
  - Informationen verloren bei Schritt zu nächstem Zustand

Monte-Carlo Suche mit Erinnerung

- erstes + letztes Problem behebbar durch Nutzung von Baum
  - dient Kapselung der Erinnerung
  - statt nach jedem Monte-Carlo Lauf alles zu vergessen, wird erster Knoten an Baum angehängt
  - nur ein Knoten wegen Speicherbedarf
- Suche innerhalb des Baumes wie bisher rein zufällig
  - Knotenexpansion schneller, wenn Nachfolger schon im Baum gespeichert
  - Aktualisierungsschritt für alle Knoten des Baumes
    - beim Fortschreiten des Spieles ist bestehende Information nutzbar

Monte-Carlo-Suche mit Bewertung



Monte-Carlo Suche mit Erinnerung

- Vorteile:
  - Speichern zugehöriger Nachfolger → weniger (Prolog-)Expansionen
  - Informationen der Nachfolger für nächsten Schritt brauchbar
- Nachteile:
  - höherer Speicherbedarf
  - ungesteuerte Suche
  - Ergebnisse potenziell schlecht

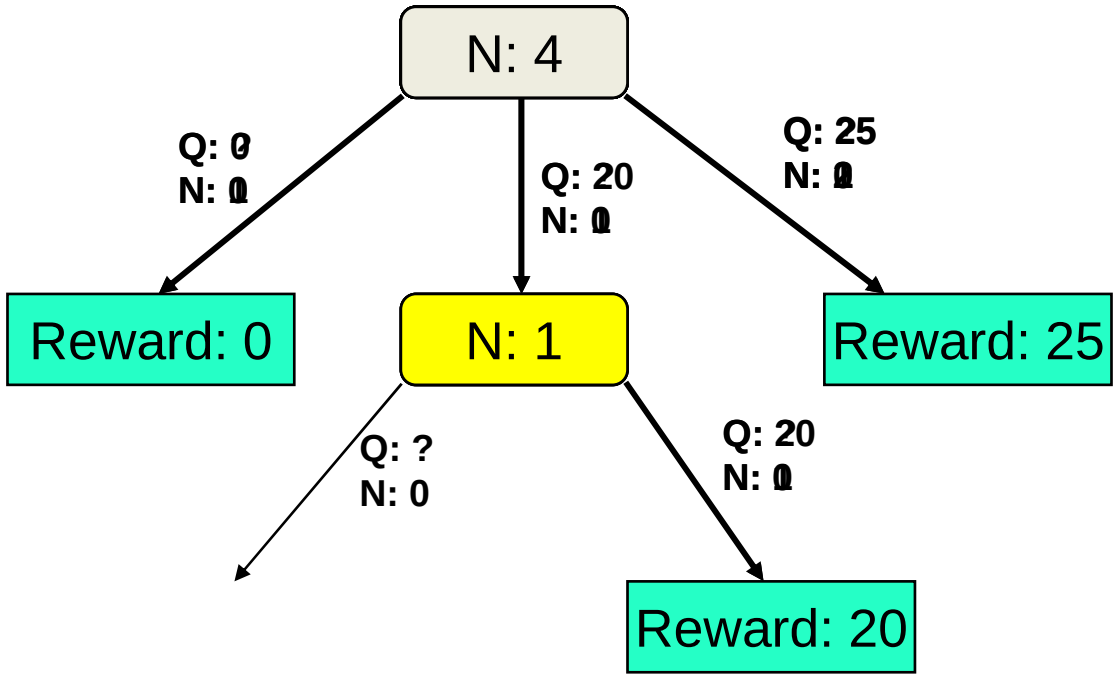
UCT [Kocsis & Szepesvári, 2006]

- "Upper Confidence Bounds applied to Trees"
- Mit Zusatzinformationen, Suche in Baum einfach steuerbar
  - In Baum
    - Wenn ≥ 1 unexpandierter Zug, wähle einen davon zufällig
    - Wähle Nachfolger, der UCT-Wert maximiert

$$Q(s,m) + C\sqrt{\frac{\ln(N(s))}{N(s,m)}}$$

    - Q(s, m): mittlere Bewertung von Zug m in Zustand s
    - C: Konstante zur Steuerung der Suche
    - N(s): Anzahl Besuche von Zustand s
    - N(s, m): Anzahl Besuche von Zug m in Zustand s
  - Wenn Blatt erreicht
    - Führe normale Monte-Carlo Suche durch
    - Propagiere Terminalbewertung durch Baum zur Wurzel hoch

UCT



$$Q(s,m) + C\sqrt{\frac{\ln(N(s))}{N(s,m)}}$$

für C = 10:
Zug 1: 10
Zug 2: 30
Zug 3: 35

für C = 100:
Zug 1: 108
Zug 2: 138
Zug 3: 130

für C = 1,000:
Zug 1: 1047
Zug 2: 1097
Zug 3: 1073

UCT

- Vorteile:
  - Speichern zugehöriger Nachfolger → weniger (Prolog-)Expansionen
  - Suche im Baum gesteuert durch bestehende Information
  - Informationen der Nachfolger für nächsten Schritt brauchbar
  - Ergebnisse tendenziell brauchbar
    - Bei internationaler Meisterschaft:
      - 2005 und 2006: Sieger mit Minimax-Suche
      - seit 2007: Sieger mit UCT
- Nachteile:
  - höherer Speicherbedarf
  - ungesteuert in Monte-Carlo Suchen

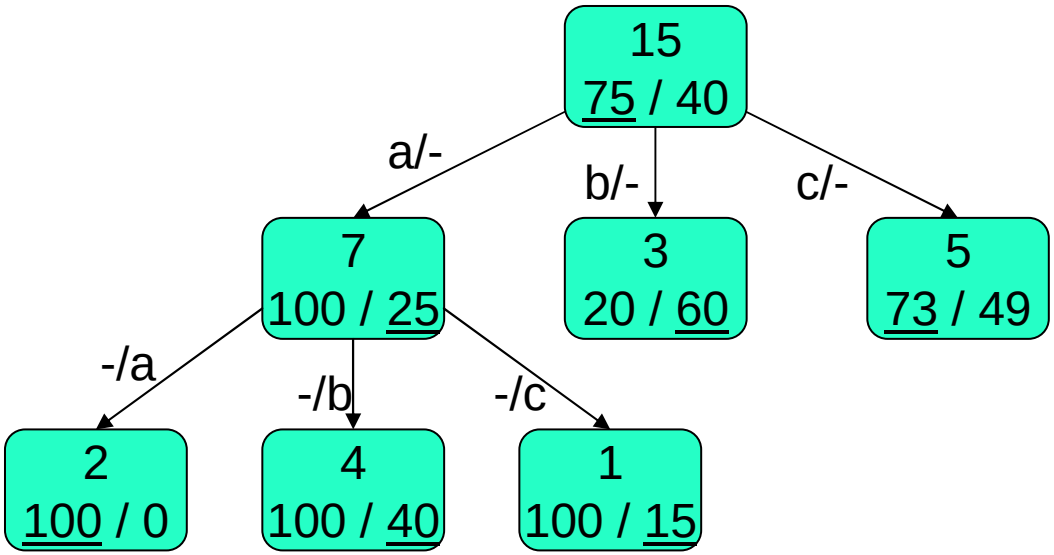
UCT für Einpersonenspiele

- speichere aktuellen (Gesamt-)Pfad temporär
- wenn Bewertung bisher beste
  - speichere Pfad global
- Speichern bester statt durchschnittlicher Bewertung
- wenn Bewertung maximal (= 100)
  - speichere Pfad global
  - stoppe UCT
  - gib ersten Zug zurück
- da kein Gegenspieler, Verfolgen maximalen Pfades sicherer Gewinn
- z.B. [Finnsson & Björnsson, 2010]

UCT für Mehrpersonenspiele

- einfach bei abwechselnden Zügen
  - Ein UCT Baum
  - Knoten-Information:
    - Anzahl Besuche
    - mittlere Bewertungen für alle Spieler
  - Knoten entspricht Zustand
    - in jedem Zustand ≤ 1 Spieler aktiv
    - Wähle Nachfolger, der für aktiven Spieler UCT-Wert maximiert

UCT für Mehrpersonenspiele



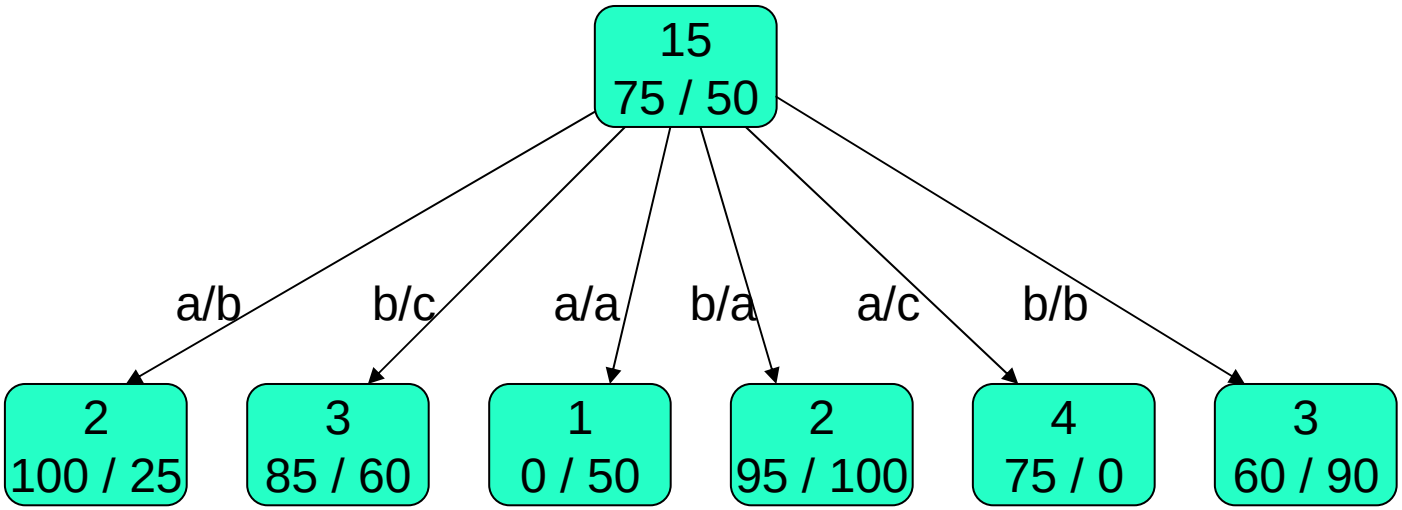
aktiver Spieler

passiver Spieler

UCT für Mehrpersonenspiele

- schwieriger bei simultanen Zügen
  - aktiven Spieler zufällig bestimmen
    - Rest wie im Fall abwechselnder Züge
    - schlecht, wenn alle Spieler unabhängig voneinander spielen, etwa Runners oder Racer
    - hier Faktorisieren von Spielen relevant (kommt später!)
  - oder für jeden Spieler
    - alle Nachfolger bei Wahl eines Zuges bestimmen
    - mittlere Bewertung und Anzahl Besuche bestimmen
    - UCT-Wert ermitteln
    - Zug mit maximalem UCT-Wert wählen
    - Problem: Laufzeit

UCT für Mehrpersonenspiele



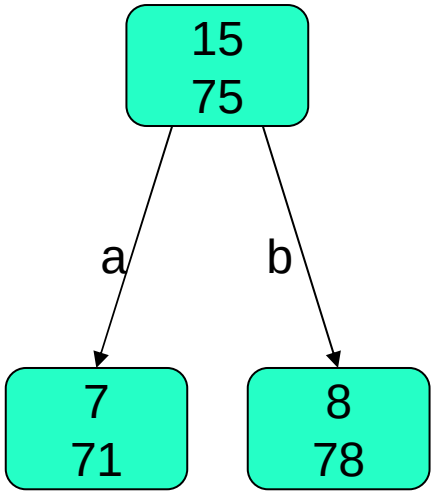
Spieler 1:
Zug a: 7 Besuche, 71 mittl. Bewertung
Zug b: 8 Besuche, 78 mittl. Bewertung

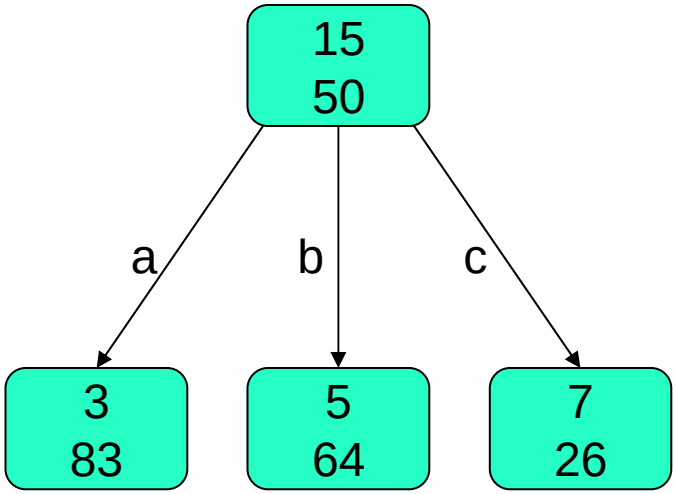
Spieler 2:
Zug a: 3 Besuche, 83 mittl. Bewertung
Zug b: 5 Besuche, 64 mittl. Bewertung
Zug c: 7 Besuche, 26 mittl. Bewertung

UCT für Mehrpersonenspiele

- Alternative: mehrere UCT Bäume [Finnsson & Björnsson, 2008]
  - ein Baum pro Spieler
  - Knoteninformation:
    - mittlere Bewertung entsprechenden Spielers
    - Anzahl Besuche
  - zusammengeführte Information direkt gespeichert (schneller)
  - Aber:
    - n Bäume speichern → speicheraufwändiger
    - tatsächlicher Nachfolger auf Basis des Gegnerzugs berechnet (Prolog)

UCT für Mehrpersonenspiele





Quellen (Minimax-basiert)

- S. Schiffel & M. Thielscher: *Fluxplayer: A Successful General Game Player*, AAAI, pp. 1191-1196, 2007
- G. Kuhlmann, K. Dresner & P. Stone: *Automatic Heuristic Construction in a Complete General Game Player*, AAAI, pp. 1457-1462, 2006
- C.A. Luckhardt & K.B. Irani: *An Algorithmic Solution of N-Person Games*, AAAI, pp. 158-162, 1986
- N. Sturtevant & M. Bowling: *Robust Game Play Against Unknown Opponents*, AAMAS, pp. 713-719, 2006
- R.E. Korf: *Generalized Game Trees*, IJCAI, pp. 328-333, 1989

Quellen (Simulations-basiert)

- L. Kocsis & C. Szepesvári: *Bandit Based Monte-Carlo Planning*, ECML, pp. 282-293, 2006
- H. Finnsson & Y. Björnsson: *Learning Simulation Control in General Game-Playing Agents*, AAAI, pp. 954-959, 2010
- H. Finnsson & Y. Björnsson: *Simulation-Based Approach to General Game Playing*, AAAI, pp. 259-264, 2008