Handlungsplanung und <u>Allgemeines Spiel</u>

„Verbesserungen für UCT und Alpha-Beta“

Peter Kissmann

Themen Allgemeines Spiel

- Einführung
- Game Desciption Language (GDL)
- Spielen allgemeiner Spiele
- Evaluationsfunktionen im allgemeinen Spiel
- Verbesserungen für UCT und Alpha-Beta
- Lösen allgemeiner Spiele
- Instanziierung
- Ausblick: Unvollständige Information und Zufall

Aufbau der Vorlesung

- Verbesserungen für UCT [Finnsson & Björnsson, 2010]
  - Move-Average Sampling Technique (MAST)
  - Tree-Only MAST (TO-MAST)
  - Predicate-Average Sampling Technique (PAST)
  - Features-to-Action Sampling Technique (FAST)
  - Rapid Action Value Estimation (RAVE)
- Verbesserungen für Alpha-Beta
  - Zuganordnung
  - Aspirationssuche
  - Null-Fenster Suche
  - Principal-Variation Suche
  - Transpositionstabellen

Probleme bei UCT

- UCT: geleitet anhand von UCT-Werten
- aber: Führung nur innerhalb des UCT Baumes
- beliebig schlechte Führung, solange wenige Expansionen

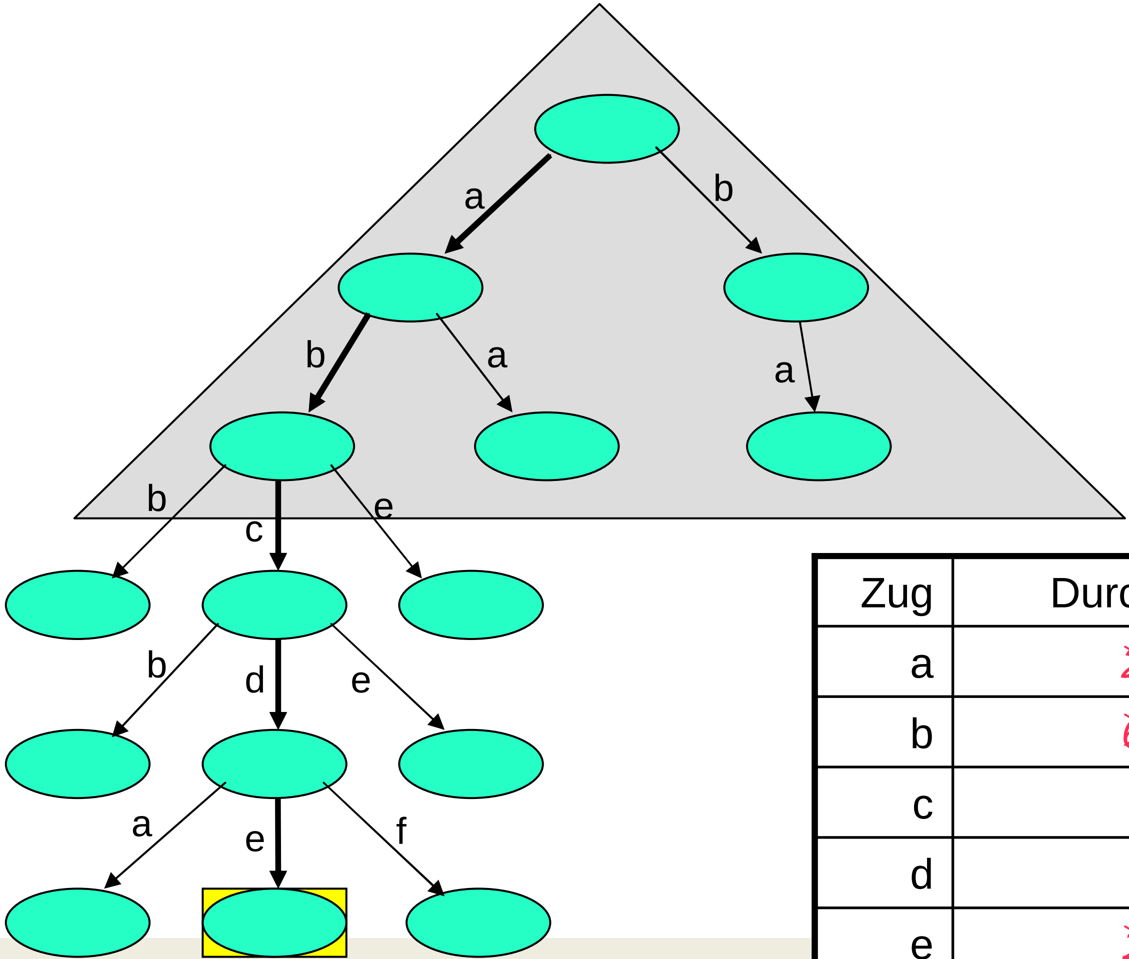
Move-Average Sampling Technique

- Idee:
  - bei jedem UCT-Durchlauf, Wissen über alle gewählten Züge verfeinern
  - Wissen nutzen, um Suchen außerhalb des Baumes zu steuern

Move-Average Sampling Technique

- Verwalte mittleren Gewinn für jeden Zug
  - unabhängig von Zustand
- Nach UCT-Durchlauf: aktualisiere mittleren Gewinn aller gewählten Züge
  - Züge, die häufig (unabhängig von Zustand) gut, bekommen höheren Wert
  - Hoffnung: Züge mit hohen Werten wahrscheinlicher gut, wenn verfügbar
    - z.B. Platzieren eines Markers in einer Ecke in Reversi
    - z.B. Schlagen eines gegnerischen Steins vor der eigenen Grundlinie in Breakthrough

Move-Average Sampling Technique



Zug	Durchschnitt	#Besuche
a	~~25~~ **41,67**	~~2~~ **3**
b	~~60~~ **63,75**	~~3~~ **4**
c	~~15~~ **35**	~~2~~ **3**
d	~~85~~ **83**	~~4~~ **5**
e	~~10~~ **20,83**	~~5~~ **6**
f	70	1

Move-Average Sampling Technique

- Während Monte-Carlo Durchlauf (außerhalb UCT Baum)
  - wähle Zug gemäß Gibbs Verteilung

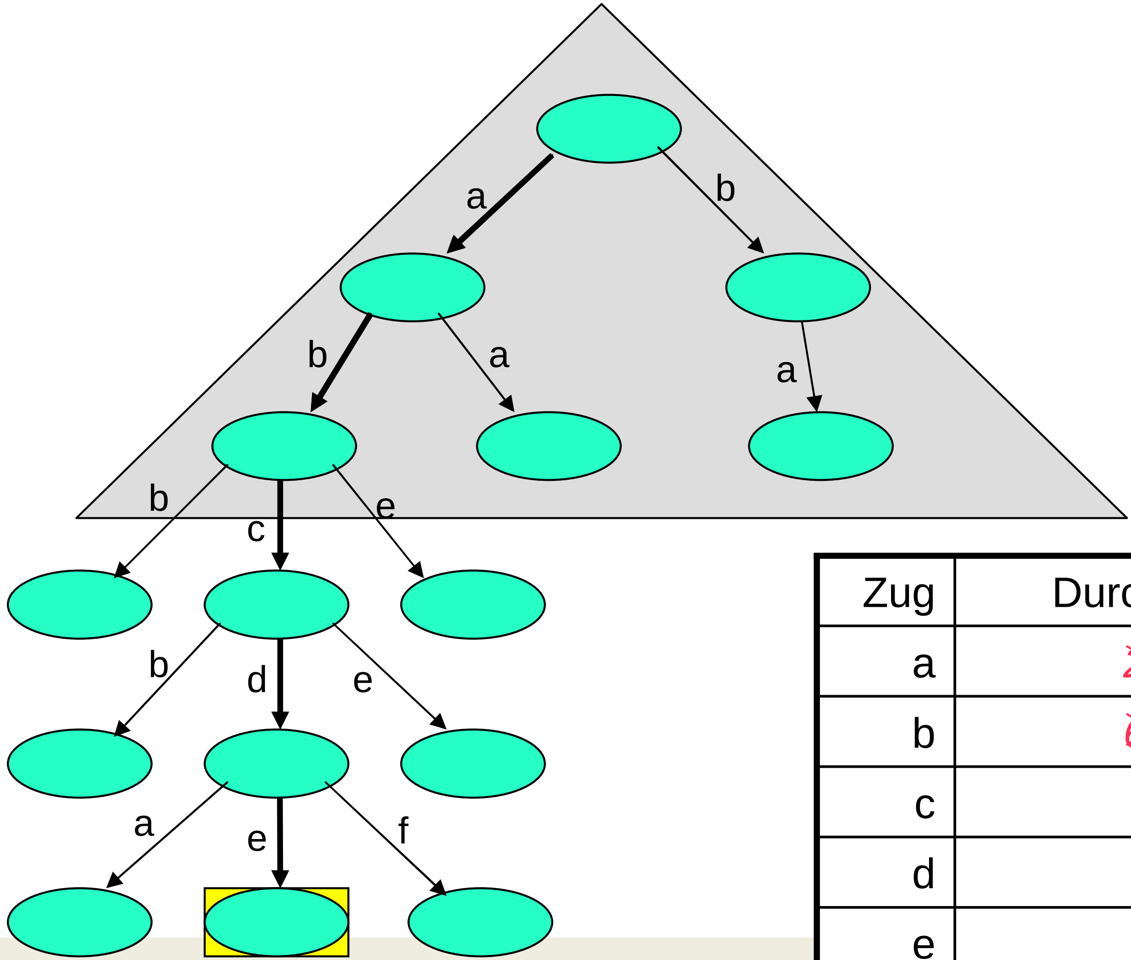
$$P(m) = \frac{e^{Q_h(m)/\tau}}{\sum_{b=1}^{n} e^{Q_h(b)/\tau}}$$

  - mit
    - m: betrachteter Zug
    - $Q_h(m)$: mittlerer Gewinn von m
    - τ: Konstante zur Steuerung (großer Wert: näher an uniformer Verteilung)

Tree-Only MAST

- (anfängliche) Monte-Carlo Durchläufe rein zufällig
- können mittleren Gewinn von Aktionen negativ beeinflussen
- Idee: nutze nur Ergebnisse aus UCT-Baum
- genauer: führe UCT (mit MAST) durch, wie bisher
- aber: aktualisiere nur Züge, die in UCT Baum gewählt
  - (ignoriere Monte-Carlo Durchlauf)
- während Monte-Carlo Durchlauf: Zugwahl gemäß Verteilung nach MAST

Move-Average Sampling Technique



Zug	Durchschnitt	#Besuche
a	~~25~~ **41,67**	~~2~~ **3**
b	~~60~~ **63,75**	~~3~~ **4**
c	15	2
d	85	4
e	10	5
f	70	1

Predicate-Average Sampling Technique

- ähnlich MAST, aber
  - nutzt gewisse Zustandsinformation während Mittelwertberechnung
- Zustand = Menge von Prädikaten (= Fluents), die gerade wahr sind
- nach UCT-Durchlauf:
  - Zug m in Zustand S gewählt
  - für alle p‘ wahr in S, aktualisiere Mittelwert Q_p(p‘, m)
- während Monte-Carlo Durchlauf:
  - wähle Zug wie in MAST, aber
  - Q_h(m) ersetzt durch Q_p(p‘, m) mit p‘ Fluent in aktuellem Zustand, für das Q_p für Zug m maximal
  - damit keine zu hohe Varianz: Nutze mittleren Wert des gesamten Spiels, bis bestimmte Zahl an Samples erreicht

Predicate-Average Sampling Technique

- MAST: Konzentration auf generell gute Züge
- PAST: Nutzen von Möglichkeit, dass Zug nur in gewissem Kontext gut

Features-to-Action Sampling Technique

- erster Ansatz, um Evaluationsfunktionen von Alpha-Beta mit UCT zu kombinieren
- statt nur der Züge oder Züge mit einzelnen Prädikaten: Züge mit identifizierten Features genutzt

Features-to-Action Sampling Technique

- hier genutzte Features:
  - Figurtypen
  - Spielbretter
- identifiziert durch template matching (= Syntaxvergleich)
- Figurtyp wichtigeres Feature, aber nur, wenn es unterschiedliche Werte annehmen kann
- sonst: Spielbrettpositionen

Features-to-Action Sampling Technique

- Temporal-Difference Learning, um relative Wichtigkeit gefundener Features zu lernen
- Ergebnis: gewichtete Kombination erkannter Features
  - Spiele mit unterschiedlichen Figurtypen:
    - Feature entspricht Anzahl Figuren gegebenen Typs
  - Spiele mit Spielbrett-basierten Features:
    - Features binär
    - bedeuten, ob Spielfeldposition durch Figur von Spieler i belegt ist oder nicht

Features-to-Action Sampling Technique

- Nutzung von Evaluationsfunktion während Monte-Carlo Durchlaufs möglich, aber zeitaufwändig:
  - für jeden Zustand
    - jeden Zug ausführen und Evaluationsfunktion auswerten
    - Zug mit bester Bewertung wählen
- Stattdessen: Einbetten in Q(m)-Framework, das auch andere Verfahren nutzen

Features-to-Action Sampling Technique

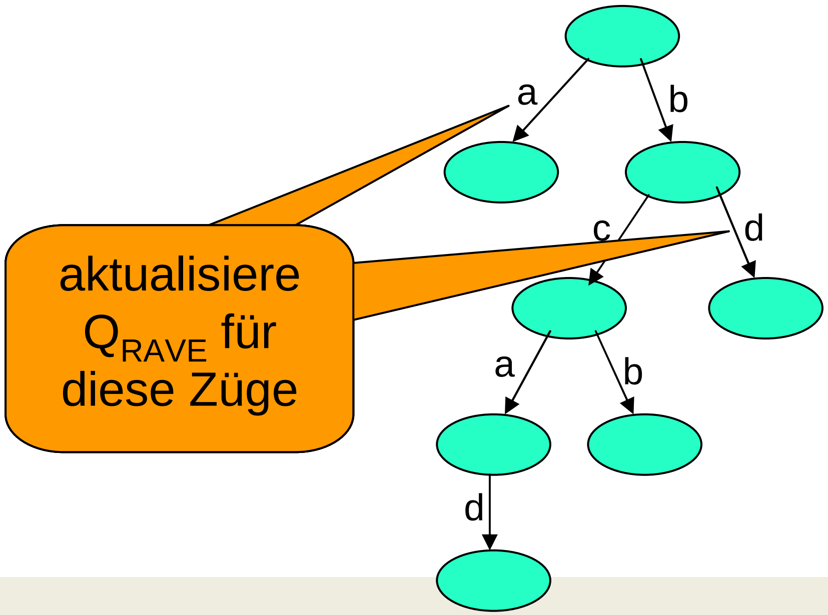
- Einbettung unterschiedlich, abhängig von erkannten Features und Zügen
  - für Figurentypen:
    - $Q(m) = \begin{cases} -\left(2 \cdot \theta_{Pce(to)} + \theta_{Pce(from)}\right) & \text{, falls eine Figur schlägt} \\ -100 & \text{, sonst} \end{cases}$
  - mit θ_{Pce}(from) und θ_{Pce}(to) gelernte Werte der Figuren auf Feldern from und to
  - Bedeutung:
    - Schlagen wichtiger, wenn möglich
    - Schlagen höherwertiger Figur wichtiger als niedrigwertiger Figur
  - für Spielbrettpositionen:
    - $Q(m) = c \cdot \theta_{p,to}$
  - mit
    - $\theta_{p,\,to}$: Gewicht für das Feature, dass Spieler p Figur auf Feld to hat
    - c: positive Konstante
- mit Q(m) kann P(m) wie üblich berechnet werden und Züge entsprechend gewählt werden

Rapid Action Value Estimation

- Entwickelt in erfolgreichem UCT Go-Spieler [Gelly & Silver, 2007]
  - dort bekannt als all-moves-as-first Heuristik
- beschleunigt Lernprozess innerhalb des UCT Baumes
- nutzt später gewählte Züge, um mehr Samples für identische, nicht gewählte Züge zu generieren

Rapid Action Value Estimation

- innerhalb des UCT Baumes
  - aktualisiere Q(s, m), wenn Zug m in Zustand s gewählt (wie bisher)
  - aktualisiere Q_{RAVE}(s, m‘) für in Zustand s nicht gewählten Zug m‘, wenn m‘ später in UCT Baum gewählt



Rapid Action Value Estimation

- bewirkt Verzerrung von üblichen Mittelwerten
  - gut am Anfang, wenn wenige Samples vorhanden
  - wegen Verzerrung aber nur bei hoher Varianz von Q(s, m) wählen
  - später, Q(s, m) zuverlässiger
  - dann, $Q_{RAVE}(s, m)$ besser ignorieren

Rapid Action Value Estimation

- dazu:
  - RAVE Werte zusätzlich speichern
  - nutze gewichte Kombination: $\beta(s) * Q_{RAVE}(s, m) + (1 - \beta(s)) * Q(s, m)$ in UCT Auswahl
  - mit
    - $\beta(s) = \sqrt{\frac{k}{3N(s)+k}}$
    - k: Äquivalenzparameter (steuert, nach wie vielen Samples beide Schätzungen gleich gewichtet werden)
    - N(s): Anzahl Besuche von Zustand s

Kombination von Verfahren

- MAST, TO-MAST, PAST, FAST für Steuerung von Monte-Carlo Durchläufen
- RAVE für frühe Steuerung in UCT-Baum
- Kombination von Verfahren damit möglich

Belegungen der Konstanten

- gemäß [Finnsson & Björnsson, 2010]
- UCT: C = 40
- MAST, TO-MAST, PAST: τ = 10
- FAST: τ = 1
- PAST: #Besuche, bis Aktionsmittelwert zurückgegeben: 3
- RAVE: k = 1000

Vergleich der Verfahren

- gemäß [Finnsson & Björnsson, 2010]
- Vergleich mit reinem UCT

Game	MAST win %	TO-MAST win %	PAST win %	RAVE win %	FAST win %
Breakthrough	90.00 (± 3.40)	85.33 (± 4.01)	85.00 (± 4.05)	63.33 (± 5.46)	81.67 (± 4.39)
Checkers	56.00 (± 5.37)	82.17 (± 4.15)	57.50 (± 5.36)	82.00 (± 4.08)	50.33 (± 5.36)
Othello	60.83 (± 5.46)	50.17 (± 5.56)	67.50 (± 5.24)	70.17 (± 5.11)	70.83 (± 5.10)
Skirmish	41.33 (± 5.18)	48.00 (± 5.29)	42.33 (± 5.16)	46.33 (± 5.30)	96.33 (± 1.86)

Schlagen passte zu keinem Template

FAST für diese Art Spiel erzeugt

- Vergleich mit MAST

Game	TO-MAST win %	PAST win %	RAVE win %	FAST win %
Breakthrough	52.33 (± 5.66)	45.67 (± 5.65)	20.33 (± 4.56)	39.67 (± 5.55)
Checkers	82.00 (± 4.18)	55.83 (± 5.35)	78.17 (± 4.36)	46.17 (± 5.33)
Othello	40.67 (± 5.47)	49.17 (± 5.60)	58.17 (± 5.49)	56.83 (± 5.55)
Skirmish	56.00 (± 5.31)	43.33 (± 5.26)	59.83 (± 5.15)	97.00 (± 1.70)

MAST für diese Art Spiel erzeugt

PAST: aufwändige Berechnungen weniger Expansionen pro Sekunde

Vergleich der Verfahren

- gemäß [Finnsson & Björnsson, 2010]
- reines UCT gegen RAVE+MAST (RM) bzw. RAVE+FAST (RF)

Game	RM win %	RF win %
Breakthrough	89.00 (± 4.35)	76.50 (± 5.89)
Checkers	84.50 (± 4.78)	77.00 (± 5.37)
Othello	79.75 (± 5.52)	81.00 (± 5.32)
Skirmish	45.00 (± 6.55)	96.00 (± 2.34)

- MAST gegen RAVE+MAST (RM) bzw. RAVE+FAST (RF)

Game	RM win %	RF win %
Breakthrough	50.50 (± 6.95)	38.50 (± 6.76)
Checkers	83.50 (± 4.87)	74.00 (± 5.81)
Othello	73.75 (± 6.01)	66.00 (± 6.43)
Skirmish	53.00 (± 6.47)	97.00 (± 2.04)

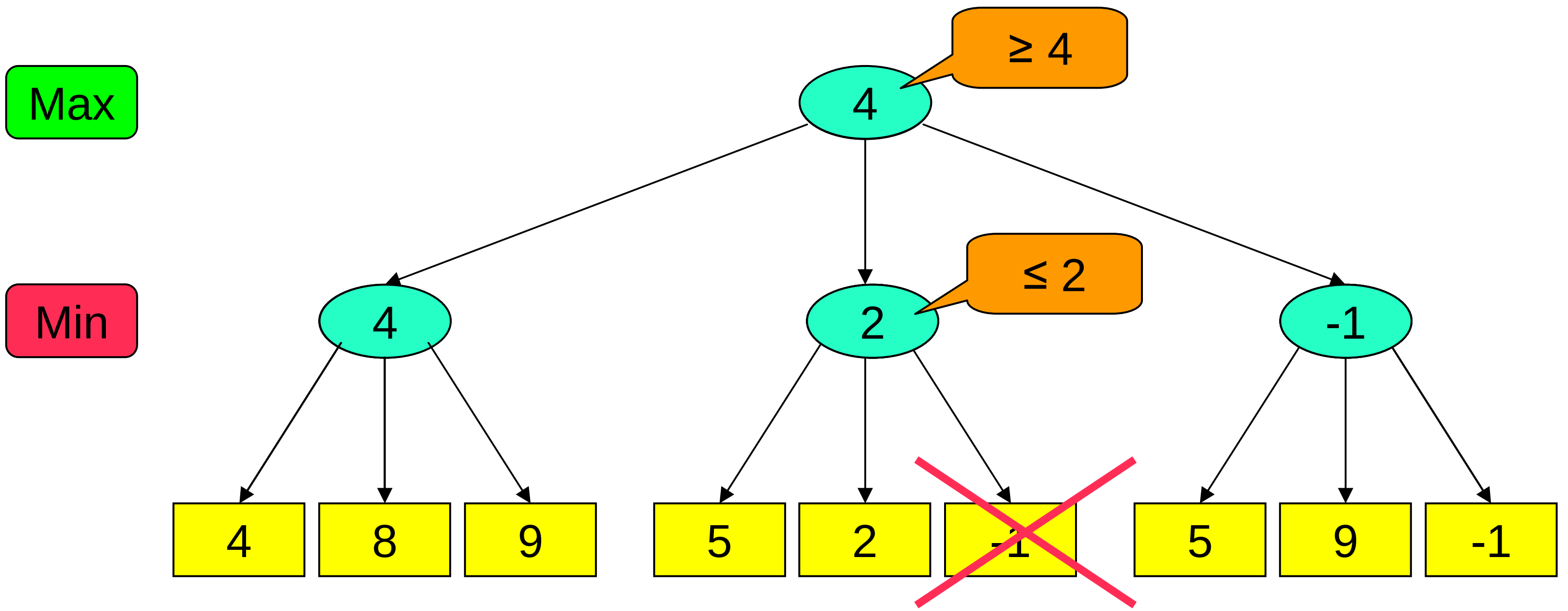
Verbesserungen für Alpha-Beta

- Alpha-Beta liefert (ohne Evaluationsfunktion) optimales Ergebnis
- expandiert weniger Zustände als Minimax
- aber:
  - viel mehr Expansionen nötig als bei UCT
  - kann nicht unterbrochen werden
- daher: Verbesserungen durch
  - Relaxierung der Optimalität (etwa Evaluationsfunktionen)
  - weniger Expansionen (hier)

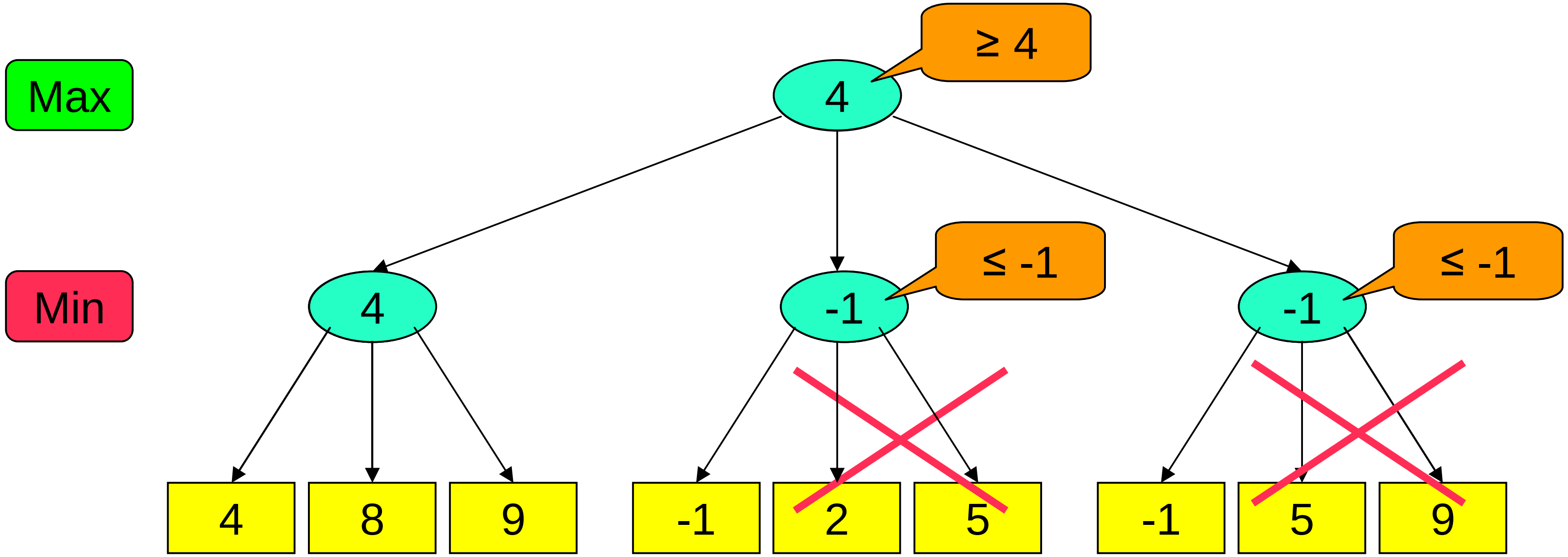
Zuganordnung

- Bisher Zugordnung
  - oft lexikalisch
  - auch entsprechend der “Erfüllungsreihenfolge” in Prolog
- besser: Züge so ordnen, dass Beschneidung möglichst früh

Zuganordnung



Zuganordnung



Zuganordnung

- Schwierig: Wie gute Zuganordnung ermitteln?
  - durch Lernverfahren
  - bei Iterative Deepening Alpha-Beta etwa auf Basis von Wissen aus vorherigen Iterationen
  - Killer-Heuristik

Killer-Heuristik

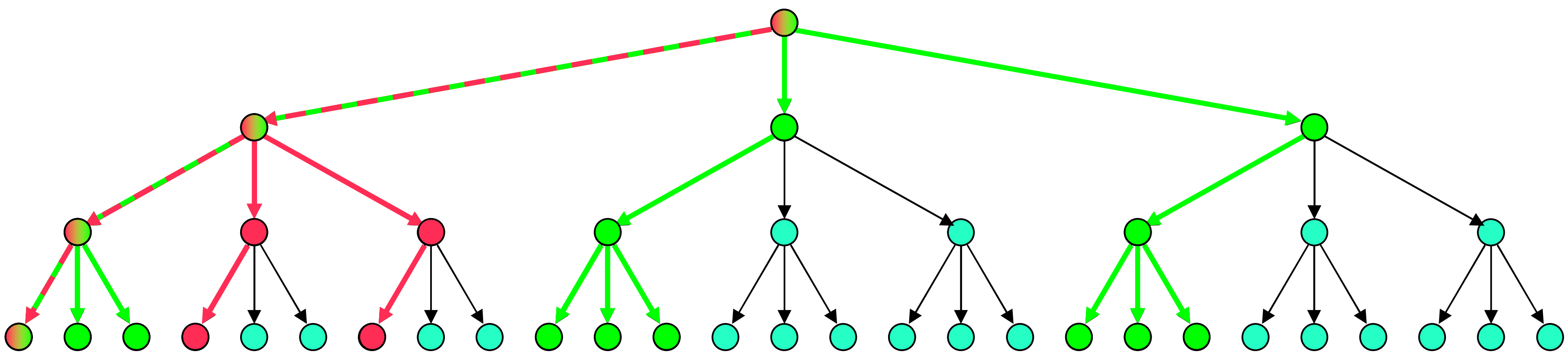
- Nutze Zug, der in der selben (BFS)-Schicht eine Beschneidung bewirkte, zuerst
  - oft 2 (oder mehr) Killer-Züge für jede Schicht gespeichert
  - diese werden (sofern anwendbar) zuerst analysiert
  - wenn nicht-Killer-Zug beschneidet
    - speichere diesen als neuen Killer Zug
    - entferne einen der alten Killer-Züge

Etwas Theorie gefällig?

- [Knuth & Moore, 1975]
- Kritischer Baum muss von jeder Variante unabhängig von Terminalwerten untersucht werden
- Bei optimaler Zugordnung müssen nur kritische Knoten untersucht werden
- Knoten beschreibbar durch Indizes der Züge, die zu ihm führen
  - $n = (m_1, m_2, \ldots, m_l)$
- Knoten $n = (m_1, m_2, \ldots, m_l)$ kritisch gdw.
  - $m_i = 1$ für i ungerade oder
  - $m_i = 1$ für i gerade

Kritischer Baum

- Knoten mit m_i = 1 für i ungerade
- Knoten mit m_i = 1 für i gerade
- Mögliche Interpretation:
  - Knoten mit m_i = 1 für i ungerade für Spielerstrategie
  - Knoten mit m_i = 1 für i gerade für Gegenspielerstrategie



Kritischer Baum

- Beispiel: b-ärer Baum (= fester Branching-Faktor von b):
  - Anzahl Knoten mit $m_i = 1$ für i ungerade in Tiefe d: $b^{\lceil d/2 \rceil}$
  - Anzahl Knoten mit $m_i = 1$ für i gerade in Tiefe d: $b^{\lfloor d/2 \rfloor}$
  - Gesamtanzahl kritische Knoten in Tiefe d: $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$

    (-1, da Knoten (1, 1, …, 1) doppelt gezählt)
- damit: bei optimaler Zugordnung etwa doppelte Suchtiefe in gleicher Zeit möglich

Aspirationssuche

- in Alpha-Beta: Suchfenster $(\alpha, \beta) = (-\infty, \infty)$ [bzw. (-1, 101) in GGP]
- Idee: wähle kleineres Fenster, um Wahrscheinlichkeit für Beschneidung zu erhöhen
- Beispiel: $(\alpha, \beta) = (v_0 - \varepsilon, v_0 + \varepsilon)$
  - v_0: (statistischer) Schätzwert der Wurzel
- funktioniert gut mit Iterative Deepening
  - Ergebnis letzter Iteration guter Schätzwert für Wurzel

Aspirationssuche

- Problem:
  - Wenn korrekte Lösung außerhalb Fenster, nicht auffindbar
- Lösung:
  - Starte Alpha-Beta mit Fenster (v_0 - ε, v_0 + ε)
  - Wenn Lösung gefunden, gib sie zurück
  - Sonst
    - erweitere Fenster und starte Suche von vorne
- Alternative:
  - Wenn keine Lösung, starte Suche neu mit Fenster
    - $(-\infty, v_1 + 1)$, falls Rückgabewert $v_1 < v_0 - \varepsilon$
    - $(v_1 - 1, \infty)$, falls Rückgabewert $v_1 > v_0 + \varepsilon$

Nullfenster-Suche

- Extremform der Aspirationssuche
- Fenster wird so klein gewählt, dass keine Lösung gefunden werden kann (v_0, v_0+1)
- damit mehr Schnitte als sonst
- Ergebnis eine Durchlaufs:
  - *fail-high*, wenn optimale Lösung größer als v_0
  - *fail-low*, wenn optimale Lösung kleiner als v_0
- Starte Suche erneut für neues v_0

Nullfenster-Suche

- Im Allgemeinen Spiel Lösung in [0, 100]
- Mit Nullfenster-Suchen binäre Suche nach optimaler Lösung möglich
- also log(100) Suchen nötig (etwa 6 Stück), um optimalen Wert auf eine Zahl einzuschränken

Principal-Variation Suche (auch Negascout)

- Vermischung von klassischem Alpha-Beta mit Nullfenster-Suche
- Starte Alpha-Beta mit normalem Fenster
- Annahme: Zugordnung optimal
  - dann sollte erster Rückgabewert in einem Knoten optimal sein
- Beweise, dass Wert optimal
  - Starte Nullfenster-Suchen für restliche Züge
  - Wenn (in Max-Knoten) alle Ergebnisse *fail-low*, Wert optimal
  - Sonst fahre an diesem Knoten mit normalem Fenster fort

Principal-Variation Suche

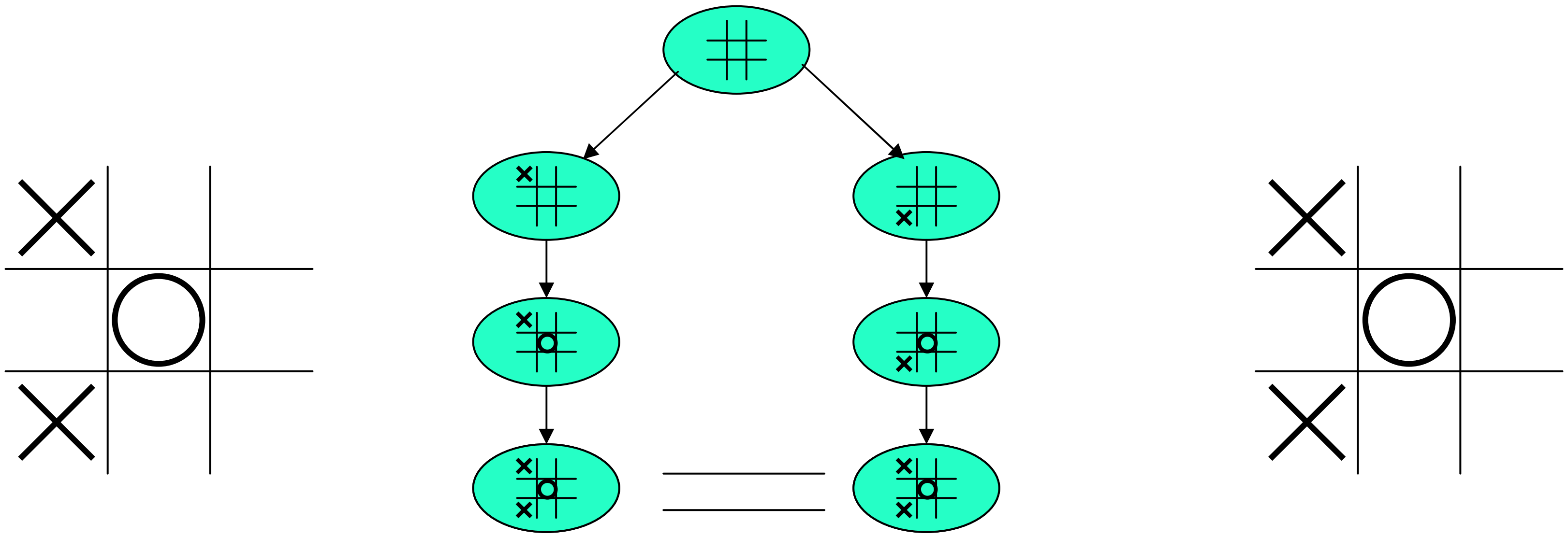
```
falls (prüfeTerminierung(zustand))
    falls (aktiv(zustand) = spieler1)
        gib findeBewertung(spieler1) zurück
    sonst
        gib -findeBewertung(spieler1) zurück
bound2 ← β
züge ← findeLegals(aktiv(zustand))
für jeden zug ∈ züge
    nachfolger ← simuliere(zug)
    bewertung ← -Negascout(nachfolger, -bound2, -α)
    falls α < bewertung < β & nicht erster Zug
        bewertung ← -Negascout(nachfolger, -β, -α)   // vollständige Suche
    falls (bewertung ≥ β) gib bewertung zurück
    falls (bewertung > α) α ← bewertung
    bound2 ← α + 1   // Null-Fenster gesetzt
gib α zurück
```

Principal-Variation Suche

- Hoffnung: sehr viele Schnitte, möglichst wenige Neustarts der Suche mit vollem Fenster in den Knoten
  - erfüllt, wenn Zugordnung gut
- dann sehr starke Beschneidung

Transpositionstabellen

- Alpha-Beta verarbeitet SpielBAUM
- aber: Spiele häufig eher Graph



Transpositionstabellen

- Erkennen von Duplikaten hilft, Expansionsanzahl zu verringern
- Transpositionstabelle speichert Zustand und gefundenen Wert
  - evtl. auch zu wählenden Zug
  - falls tiefenbeschränkt, auch Tiefe und ob gefundener Wert exakt oder untere oder obere Schranke
- Wenn Zustand erreicht, der schon in Transpositionstabelle
  - Informationen daraus übernehmen

Transpositionstabellen

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

Transpositionstabellen

- Suche nach Zustand in Tranpositionstabelle über tt_suche
- Speichern von Zustand in Tabelle über tt_speichere
- Zugriff typischerweise nicht direkt über Zustand, sondern Hashwert
  - Vergleich von Zahlen viel schneller als Vergleich kompletter Zustände
  - wenn Hashfunktion “perfekt”, wird jeder Zustand auf eindeutigen Hashwert abgebildet
  - sonst evtl. Kollisionen
    - verschiedene Verfahren zur Kollisionsbehandlung

Quellen

- H. Finnsson & Y. Björnsson: *Learning Simulation Control in General Game-Playing Agents*, AAAI, pp. 954-959, 2010
- S. Gelly & D. Silver: *Combining Online and Offline-Knowledge in UCT*, ICML (227), pp. 273-280, 2007
- D.E. Knuth & R.W. Moore: *An Analysis of Alpha-Beta Pruning*, Artificial Intelligence 6 (4), pp. 293-326, 1975