Handlungsplanung und Allgemeines Spiel

„Lösen allgemeiner Spiele durch symbolische Suche“

Peter Kissmann

Themen Allgemeines Spiel

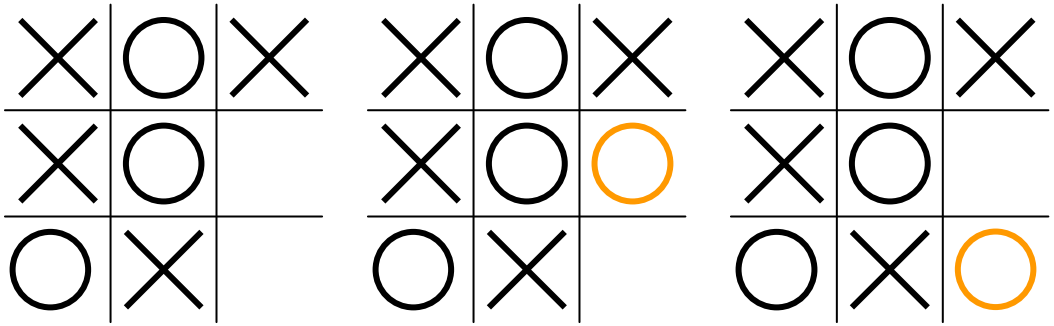
- Einführung
- Game Desciption Language (GDL)
- Spielen allgemeiner Spiele
- Evaluationsfunktionen im allgemeinen Spiel
- Verbesserungen für UCT und Alpha-Beta
- Allgemeine Verbesserungen
- Lösen allgemeiner Spiele durch symbolische Suche
- Instanziierung
- Ausblick: Unvollständige Information und Zufall

Motivation Symbolische Suche

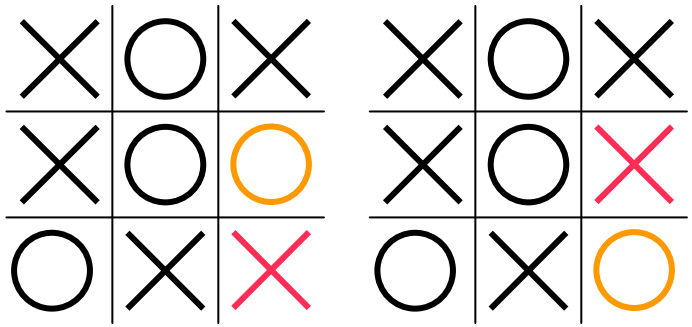
- bisherige Verfahren explizit
  - jeder Zustand einzeln gespeichert
  - bei b Bits pro Zustand und n Zuständen insgesamt b·n Bits nötig
- Idee:
  - Gemeinsamkeiten von mehreren Zuständen ausnutzen
  - statt Einzelzustände Zustandsmengen verarbeiten
- Schwierigkeit:
  - Bisherige Algorithmen nur für Einzelzustände
  - Umdenken für Zustandsmengen nötig

Motivation Symbolische Suche

- Beispiel Tic-Tac-Toe:
  - 5 Zustände:

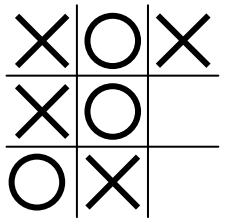


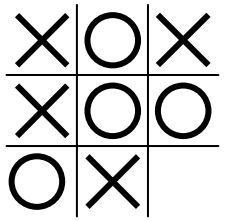
- Spaltenweise Kodierung:
  - xxoooxxbb
  - xxoooxxob
  - xxoooxxbo
  - xxoooxxox
  - xxoooxxxo



Motivation symbolische Suche

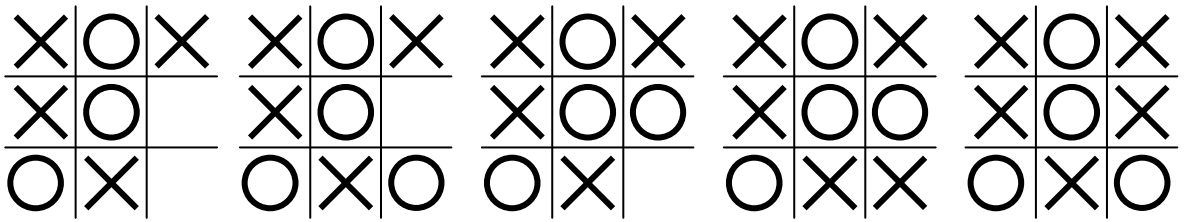
- Kodierung kann als Disjunktion von Konjunktionen betrachtet werden:
  - ((cell 1 1 x) und (cell 1 2 x) und (cell 1 3 o) und (cell 2 1 o) und (cell 2 2 o) und (cell 2 3 x) und (cell 3 1 x) und (cell 3 2 b) und (cell 3 3 b)) oder
  - ((cell 1 1 x) und (cell 1 2 x) und (cell 1 3 o) und (cell 2 1 o) und (cell 2 2 o) und (cell 2 3 x) und (cell 3 1 x) und (cell 3 2 o) und (cell 3 3 b)) oder
  - …





Motivation Symbolische Suche

- Diese gemäß Aussagenlogik zusammenfassbar
  - hier insbesondere: Distributivgesetz
    - $((a \wedge b) \vee (a \wedge c)) \equiv (a \wedge (b \vee c))$
    - $((a \vee b) \wedge (a \vee c)) \equiv (a \vee (b \wedge c))$
- Ergebnis:
  - 5 Zustände beschreibbar als
    - (cell 1 1 x) und (cell 1 2 x) und (cell 1 3 o) und
      (cell 2 1 o) und (cell 2 2 o) und (cell 2 3 x) und
      (cell 3 1 x) und
      (((cell 3 2 b) und ((cell 3 3 b) oder (cell 3 3 o)) oder
      ((cell 3 2 o) und ((cell 3 3 b) oder (cell 3 3 x))) oder
      ((cell 3 2 x) und (cell 3 3 0)))



Motivation Symbolische Suche

- Damit:
  - nicht mehr jeder Zustand durch identische Anzahl Bits dargestellt
  - mehrere Zustände in einer Formel zusammengefasst
  - typischerweise speichereffizienter
- Dieses bewerkstelligen insbesondere Binäre Entscheidungsdiagramme (BDDs)
  - bilden Grundlage für unsere symbolische Suche

Aufbau

- BDDs
- Symbolische Suche
- Finden aller erreichbaren Zustände
- Lösen von Einpersonenspielen
- Lösen von Zweipersonenspielen

Binäre Entscheidungsdiagramme (BDDs)

- Gerichteter azyklischer Graph mit
  - einem Wurzelknoten
  - einer Menge (binärer) innerer Knoten
  - zwei ausgezeichneten Blättern
    - 0-Senke
    - 1-Senke

BDDs

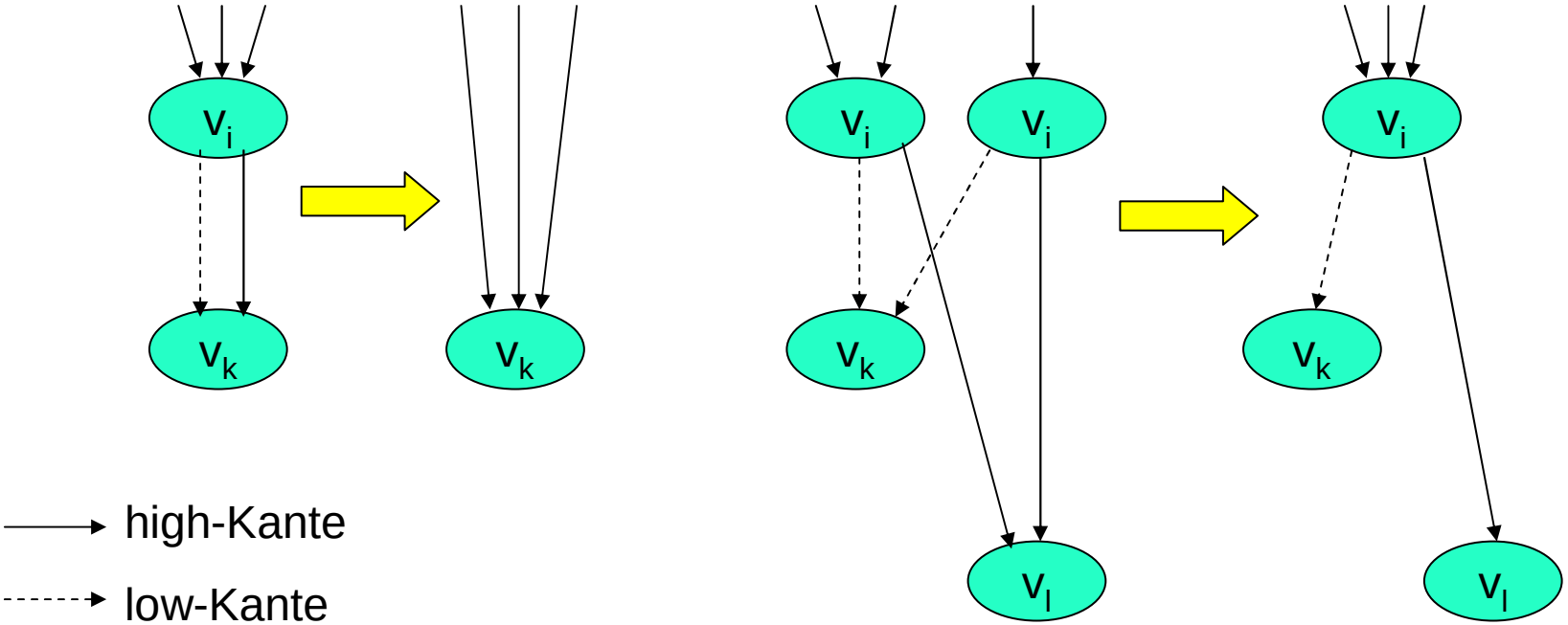
- Innere Knoten:
  - repräsentieren binäre Variable
  - zwei ausgehende Kanten:
    - high-Kante = Wert repräsentierter Variable ist *wahr*
    - low-Kante = Wert repräsentierter Variable ist *falsch*
- Senken:
  - jeder Pfad von Wurzel zu 1-Senke entspricht gültiger Belegung von Variablen
- BDDs gut geeignet, Boolesche Formeln zu repräsentieren

Geordnete BDDs (OBDDs)

- [Bryant, 1986]
- Nutzung fester Variablenordnung
  - zuvor, Variablen in beliebiger Reihenfolge
  - konnten auf einem Pfad auch mehrfach auftauchen

Reduzierte OBDDs (ROBDDs)

- mit fester Variablenordnung, 2 Reduktionsregeln anwendbar

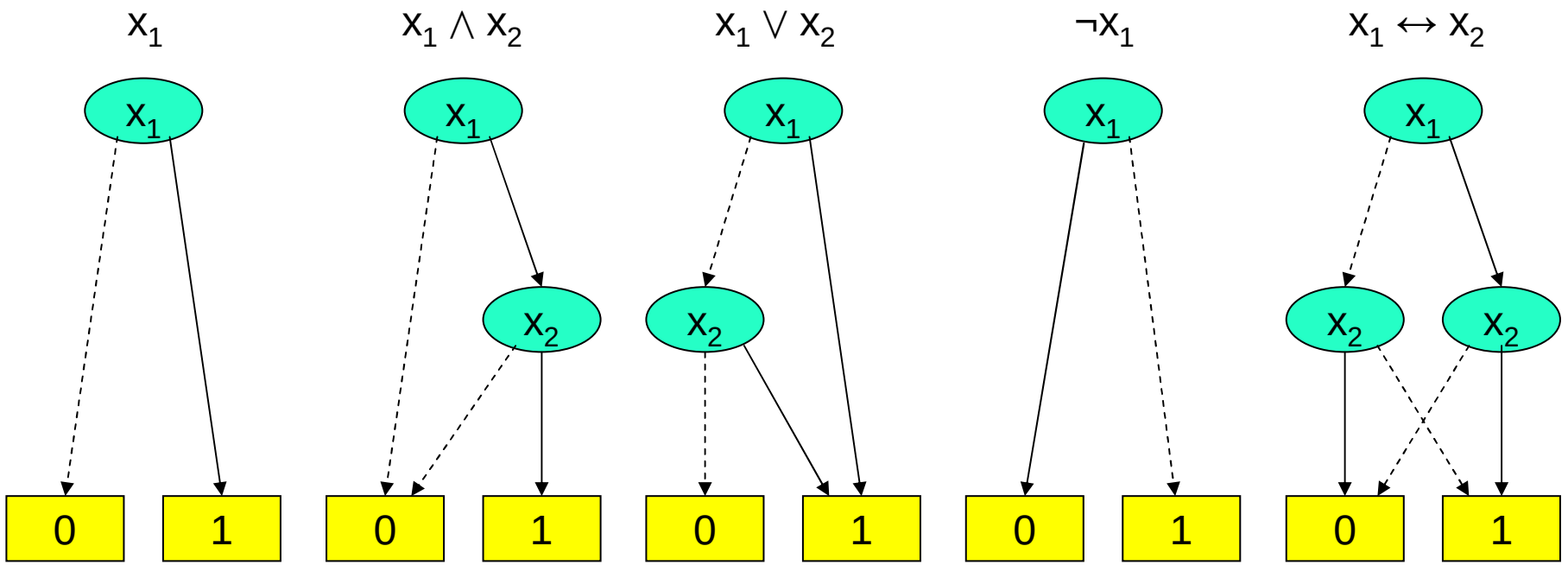


ROBDDs

- durch feste Variablenordnung und Reduzierung
  - eindeutige Darstellung für Boolesche Formeln
  - jede Belegung von Variablen nur einmal enthalten (keine Duplikate)
- heutzutage Standardform der BDDs
  - wenn von BDDs gesprochen, zumeist ROBDDs gemeint (auch hier!)

ROBDDs

- Standard-ROBDDs für einfache Boolesche Formeln:



ROBDDs

- Zustand = Menge von (mehrwertigen) Variablen
- mehrwertige Variablen lassen sich als binäre Variablen darstellen
  - für eine n-wertige Variable benötigen wir ⌈log(n)⌉ binäre Variablen
    - für 7-wertige:
      - Wert 0 entspricht 000
      - Wert 1 entspricht 001
      - Wert 2 entspricht 010
      - etc.
  - damit Zustand auch als Konjunktion binärer Variablen

ROBDDs

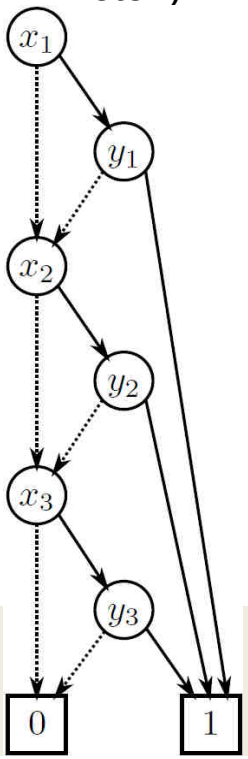
- Zustandsmenge = Menge von Zuständen
  - entspricht Disjunktion verschiedener Zustände
  - also Disjunktion von Konjunktionen von binären Variablen

Variablenordnung

- gute Variablenordnung kann entscheidend sein
  - bei schlechter Variablenordnung keine (oder nur geringe) Ersparnis gegenüber expliziter Zustandsspeicherung
  - bei guter Variablenordnung teils exponentiell weniger BDD Knoten als Bits in expliziter Darstellung

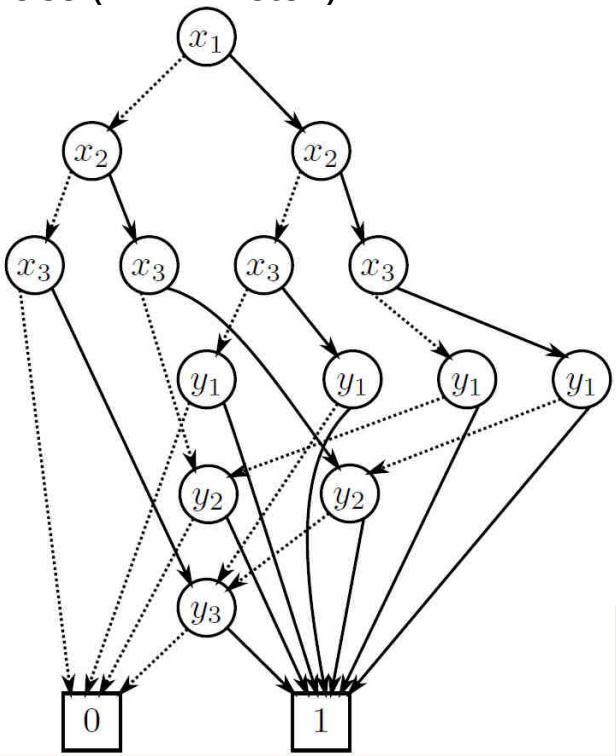
Variablenordnung

- Beispiel: Disjunkte Quadratische Form (DQF)
  - $DQF_n(x_1, \ldots, x_n, y_1, \ldots, y_n) := (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$
  - Variablenordnung $\pi = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$:
    - Lineare Größe (2n + 2 Knoten)



Variablenordnung

- Beispiel: Disjunkte Quadratische Form (DQF)
  - $DQF_n(x_1, \ldots, x_n, y_1, \ldots, y_n) := (x_1 \wedge y_1) \vee (x_2 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$
  - Variablenordnung $\pi = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$:
    - Exponentielle Größe (2^{n+1} Knoten)



Variablenordnung

- es gibt auch Beispiele, in denen Variablenordnung keinen signifikanten Unterschied macht
  - für Integermultiplizierung hat jedes BDD exponentiell viele Knoten
  - Permutationsspiele ((n^2-1)-Puzzle, Blocksworld) schlecht für BDDs (haben exponentielle Größe)
  - für symmetrische Funktionen hat jedes BDD höchstens $O(n^2)$ Knoten

Symbolische Suche

- Wir können einzelne Zustände als BDDs darstellen
- Wir können Zustandsmengen als BDDs darstellen
  - indem wir Zustände als boolesche Formeln interpretieren
- Wie können wir jetzt mit BDDs suchen?

Symbolische Suche

- Wir benötigen zwei Variablensätze
  - einen Variablensatz (S) für Variablen aktueller Zustände
  - einen Variablensatz (S') für Variablen nachfolgender Zustände

Transitionsrelation

- Zusätzlich nötig: Transitionsrelation als BDD
  - in Handlungsplanung: Aktion bestehend aus
    - Vorbedingung
    - Effekten
  - im allgemeinen Spiel: Zug bestehend aus
    - globaler Vorbedingung (legal)
    - Menge von Vorbedingung / Effekt Paaren (next)
- Vorbedingungen und Effekte jeweils Boolesche Formeln
- Vorbedingungen nutzen aktuelle Zustandsvariablen S
- Effekte nutzen Nachfolgezustandsvariablen S’
- Verknüpft über Konjunktion

Transitionsrelation

- Handlungsplanung
  - BDD für Aktion a_i = (pre, eff)
    - $trans_i(S, S') := pre_{BDD}(S) \wedge eff_{BDD}(S') \wedge frame_{BDD}(S, S')$
  - $frame_{BDD}$: modelliert Frame
    - alles, was sich nicht ändert (in Handlungsplanung muss dies erst berechnet werden - alles, was kein positiver oder negativer Effekt ist, bleibt unverändert)
    - falls v_1 und v_2 unverändert:
      - $frame_{BDD}(S, S') := (v_1(S) \leftrightarrow v_1(S')) \wedge (v_2(S) \leftrightarrow v_2(S'))$

Transitionsrelation

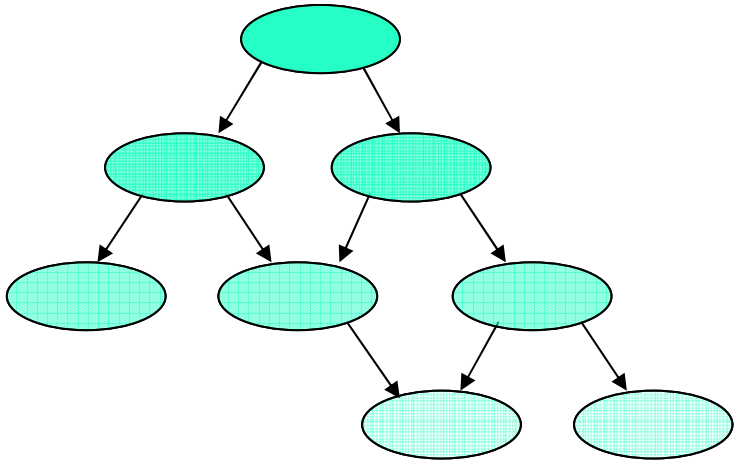
- Allgemeines Spiel
  - Frame explizit modelliert (in next Relation)
  - Zug a_i
    - folgt aus legal leg
    - kompatibel mit $next_1$, …, $next_n$
      - $next_k$:= (<= h_k b_k)
    - $trans_i(S, S') := leg_{BDD}(S) \wedge (h_{1,BDD}(S') \leftrightarrow b_{1,BDD}(S)) \wedge \ldots \wedge (h_{n,BDD}(S') \leftrightarrow b_{n,BDD}(S))$

Transitionsrelation

- Transitionsrelation Disjunktion von BDDs aller Züge
  - trans := $trans_1 \vee trans_2 \vee \ldots \vee trans_n$
- eventuell besser, Transitionsrelationen für alle Züge einzeln zu verwalten
  - gesamte Transitionsrelation kann
    - groß werden (viel Speicher benötigen)
    - lange zur Generierung benötigen

Nachfolgerberechnung

- Finden von Nachfolgern über image-Funktion
  - image(current) := ($\exists$S: trans(S, S') $\wedge$ current(S)) [S' $\rightarrow$ S]
  - current: aktuelle Zustandsmenge
  - [S' $\rightarrow$ S]: Verschieben von Nachfolgervariablen auf aktuelle Variablen
    - nötig, um Suche fortsetzen zu können



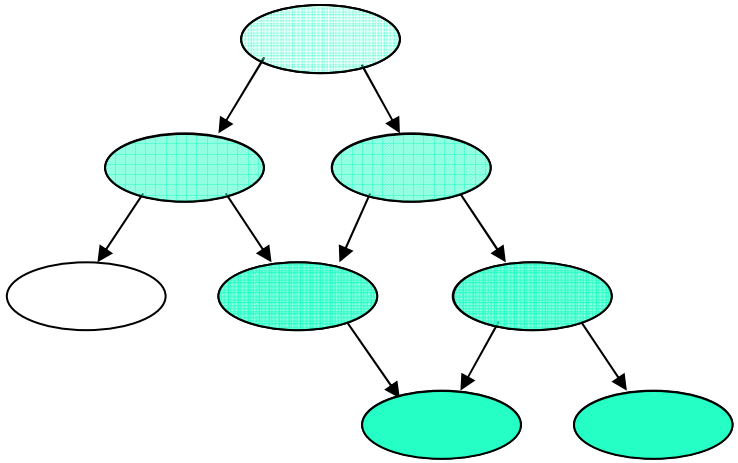
Nachfolgerberechnung

- image funktioniert auch ohne große Transitionsrelation:

$$
\begin{aligned}
image\,(current) &= (\exists S.trans\,(S,S') \wedge current\,(S))\,[S' \to S] \\
&\equiv \left(\exists S.\left(\bigvee_{a\in\mathcal{A}} trans_a\,(S,S')\right) \wedge current\,(S)\right)[S' \to S] \\
&\equiv \left(\exists S.\bigvee_{a\in\mathcal{A}} (trans_a\,(S,S') \wedge current\,(S))\right)[S' \to S] \\
&\equiv \bigvee_{a\in\mathcal{A}} (\exists S.trans_a\,(S,S') \wedge current\,(S))\,[S' \to S]
\end{aligned}
$$

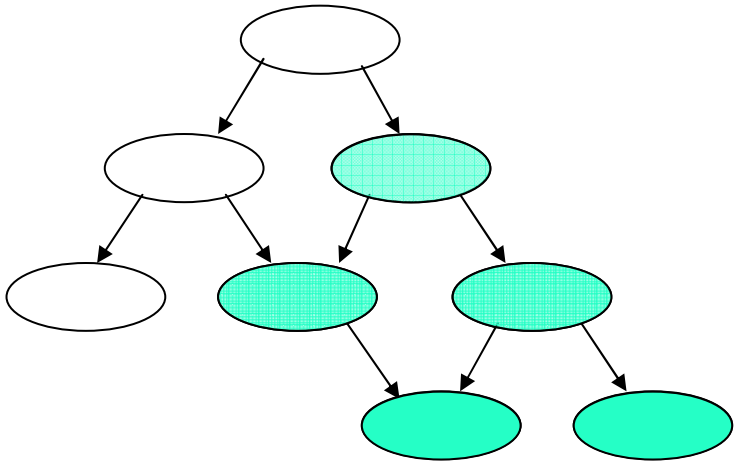
Vorgängerberechnung

- Finden aller Vorgänger sehr ähnlich (einzig Wechsel von Vorgänger- und Nachfolgervariablen)
  - pre-image(current) := ($\exists$S': trans(S, S') $\wedge$ current(S')) [S $\rightarrow$ S']



Vorgängerberechnung2

- bei Mehrpersonenspielen teilweise auch andere Form der Vorgängerberechnung nötig:
  - alle Vorgänger, von denen alle Nachfolger in übergebener Menge liegen
    - strong pre-image(current) := (∀S': trans(S, S') → current(S')) [S → S']



Vorgängerberechnung2

- In BDD-Paketen relationales Produkt (exist-and) oft sehr effizient implementiert
- damit image (und pre-image) darauf zurückführbar und auch effizient
- strong pre-image nutzt All-Quantor und Implikation statt Existenz-Quantor und Konjunktion
  - trotzdem darauf zurückführbar

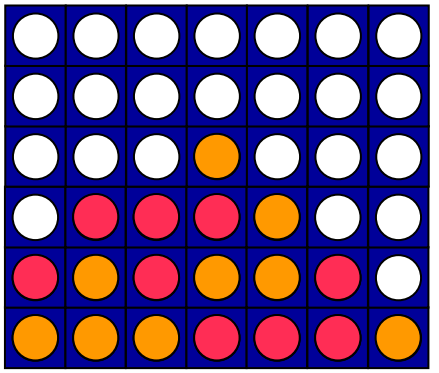
Vorgängerberechnung2

$$
\begin{aligned}
strong\ pre\text{-}image\,(current) &= (\forall S'.trans\,(S,S') \rightarrow current\,(S'))\,[S \rightarrow S'] \\
&\equiv \neg\neg\,(\forall S'.\neg trans\,(S,S') \vee current\,(S'))\,[S \rightarrow S'] \\
&\equiv \neg\,(\exists S'.\neg\,(\neg trans\,(S,S') \vee current\,(S')))\,[S \rightarrow S'] \\
&\equiv \neg\,(\exists S'.\,(trans\,(S,S') \wedge \neg current\,(S')))\,[S \rightarrow S'] \\
&\equiv \neg pre\text{-}image\,(\neg current)
\end{aligned}
$$

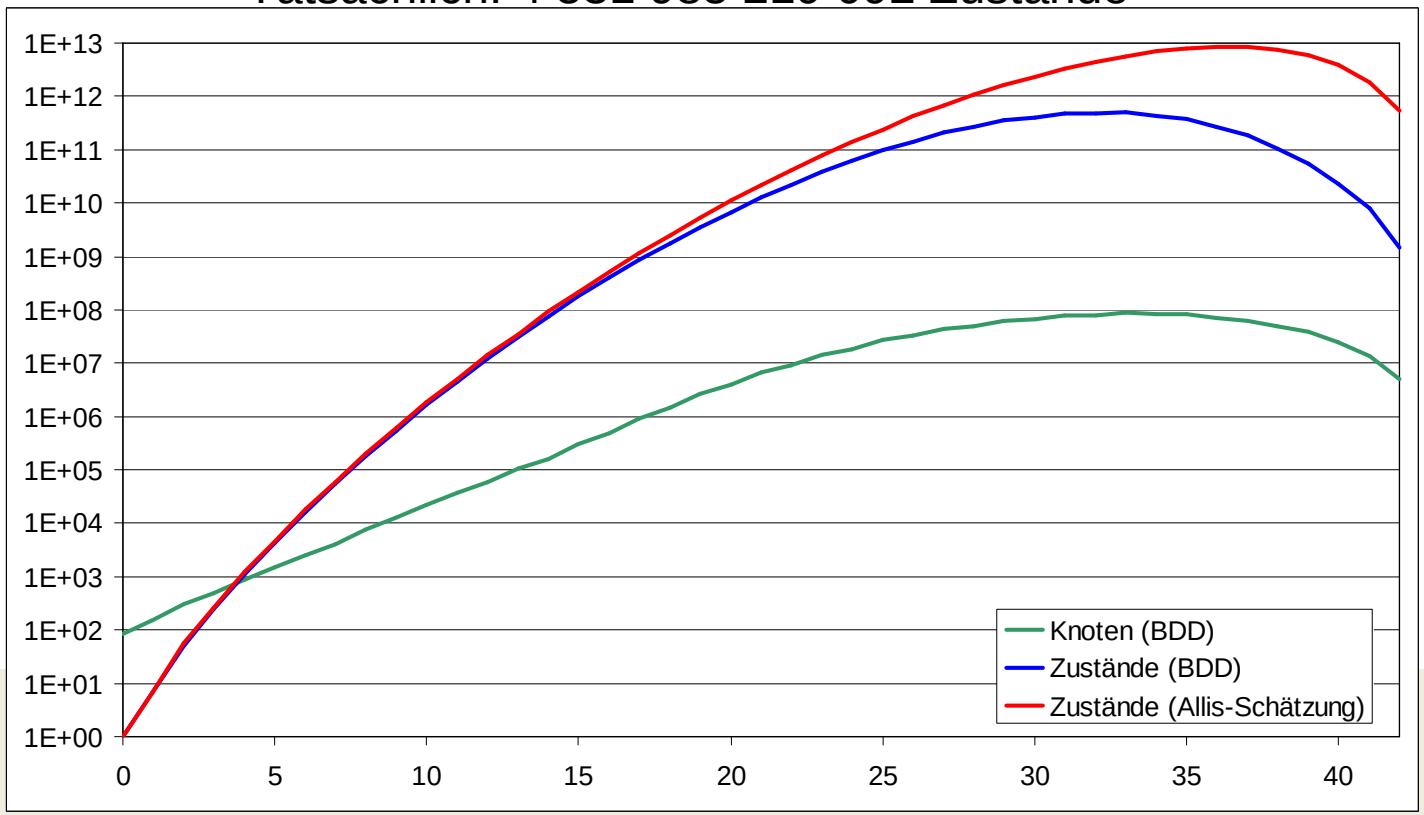
Finden aller erreichbaren Zustände

- Erreichbare Zustände typischerweise über
  - Tiefensuche (DFS) oder
  - Breitensuche (BFS)
- BDDs gut geeignet, um Breitensuche durchzuführen
  - image liefert direkt vollständiges Nachfolgelayer
  - Iteration des image, bis alle Zustände erreicht = BFS

Erreichbarkeitsanalyse Vier Gewinnt



- 1988 gelöst von Victor Allis und James D. Allen
  - Schätzung von Allis: 70 728 639 995 483 Zustände
  - Tatsächlich: 4 531 985 219 092 Zustände



Lösen von Spielen

- verschiedene Grade des Lösens (bei Zweipersonen-Spielen)
  - sehr schwach: Bewertung an Initialzustand bekannt, optimaler Spielablauf unbekannt
    - Beispiel: Hex (sicherer Gewinn für ersten Spieler)
  - schwach: finden von Strategien von Startzustand, so dass Spieler optimal spielen (= finden von Bewertung an Initialzustand + optimale Züge)
    - Beispiele: Vier Gewinnt (erster Spieler siegt), Checkers (Dame) (unentschieden)
  - stark: Strategie, um in jedem Zustand optimal zu spielen (= finden von Bewertung in allen Zuständen + optimale Züge) - Lösung auch anwendbar, wenn schon schlechter Zug durchgeführt wurde
    - Beispiele: Tic-Tac-Toe (unentschieden), Nim-Spiel (abhängig von Initialzustand)

Lösen von Einpersonenspielen

- Klassische Herangehensweise: vollständige Suche
  - Breitensuche (BFS)
  - Tiefensuche (DFS)
- Aber: damit nur alle erreichbaren Zustände, nicht
  - optimale Lösung für jeden Zustand
  - optimaler Zug in jedem Zustand
- Lösung: Mehrere Suchen (zu unterschiedlichen Zielen)

Lösen von Einpersonenspielen

- Aber: Woher bei Start wissen, zu welchem Ziel wir gehen?
- Lösung: Rückwärtssuche (Start an Terminalzuständen mit gewünschten Gewinnen)
- Hier symbolische Suche vorteilhaft
  - arbeitet auf Mengen, Start an Terminalzuständen ist oft schon mehrelementige Menge

Lösen von Einpersonenspielen

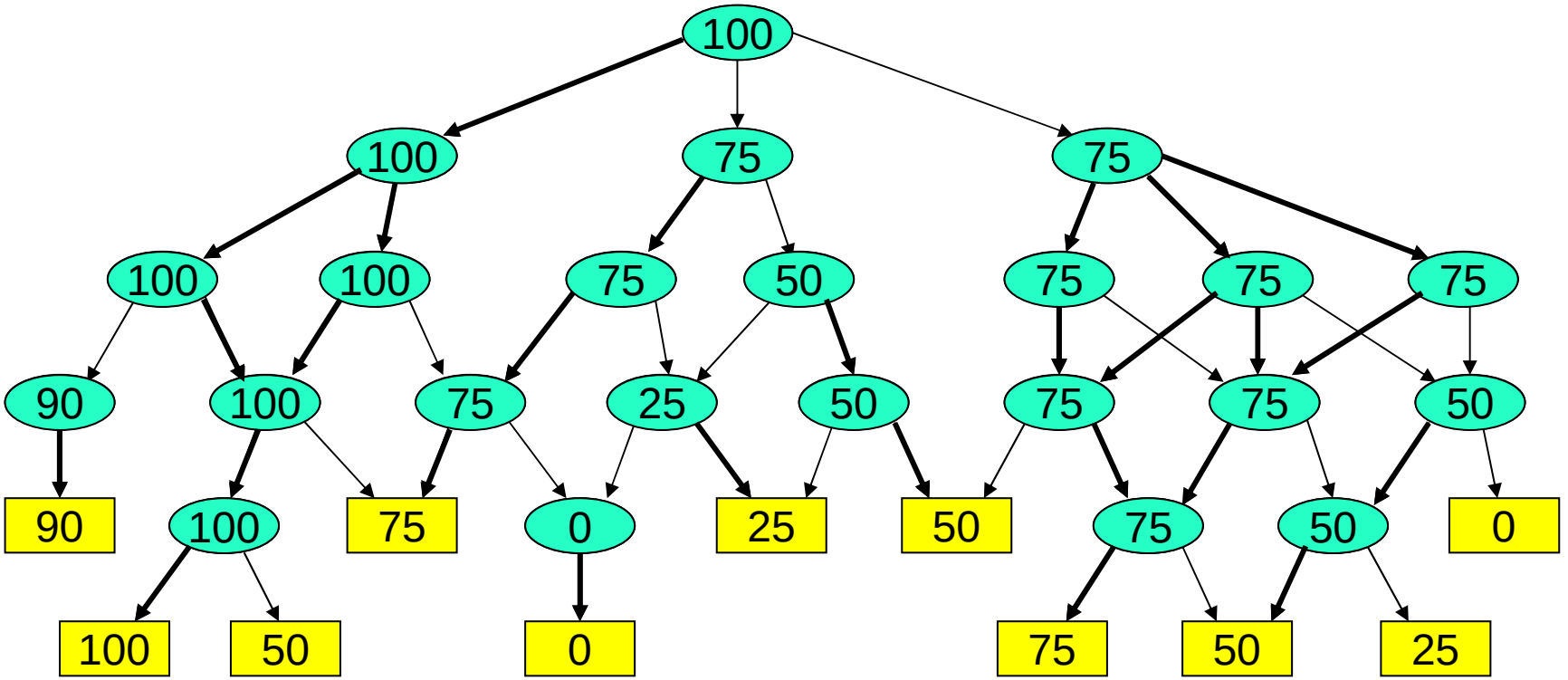
- Wichtig: Erst alle Zustände bestimmen, in denen 100 Punkte möglich
- Dann, alle Zustände, in denen 99 Punkte möglich - aber Zustände, in denen 100 Punkte möglich davon entfernen
- Am Ende, für jeden Zustand maximal möglicher Gewinn bekannt
- Optimaler Zug:
  - in aktuellem Zustand x Punkte
  - wähle Zug, so dass in Nachfolgezustand ebenfalls x Punkte
- [K&E, 2007]

Symbolic Single Player

- für alle 100 ≥ gewinn ≥ 0
  - neu ← terminal ∧ $goal_{0,gewinn}$
  - solange neu ≠ ⊥
    - $lösung_{gewinn}$ ← $lösung_{gewinn}$ ∨ neu
    - neu ← pre-image(neu)
    - für alle 100 ≥ höhererGewinn ≥ gewinn
      - neu ← neu ∧ ¬$lösung_{höhererGewinn}$

Zielbeschreibung mit Gewinn gewinn für Spieler 0

Lösen von Einpersonenspielen



Ergebnis Solitär

insgesamt:
375 110 246 Zustände

Steine übrig	Gewinn	# BDD Knoten	# Zustände
1 (in Mitte)	100	1 835 093	26 856 243
1 (woanders)	99	70	4
2	90	7 321 698	134 095 586
3	80	7 022 261	79 376 060
4	70	6 803 498	83 951 479
5	60	3 589 371	25 734 167
6	50	2 309 661	14 453 178
7	40	1 266 697	6 315 974
8	30	651 352	2 578 583
9	20	338 281	1 111 851
10	10	166 229	431 138
>10	0	94 094	205 983

Lösen von Zweipersonenspielen

- Zunächst: Nullsummenspiele
  - Genauer, Spiele mit Zuständen, die für genau einen Spieler gewonnen (für anderen entsprechend verloren) oder unentschieden sind
  - Mögliche Bewertungen: (0/100), (50/50), (100/0)
- Zwei Rückwärtssuchen, startend an gewonnenen Terminalzuständen des jeweiligen Spielers
  - Finden aller Zustände, die für jeweiligen Spieler gewonnen

Lösen von Nullsummenspielen

- Während Rückwärtssuchen, nutzen von “Doppelschritten”:
  - finden von Zuständen für die gilt:
    - für jeden Zug, den Gegenspieler wählt, kann ich Zug wählen, damit ich zu gewonnenem Zustand komme
  - dann gefundene Zustände auch gewonnen
- damit, alle Zustände, die für Spieler 0 oder Spieler 1 gewonnen
- Rest: unentschieden
- [Edelkamp, 2002]

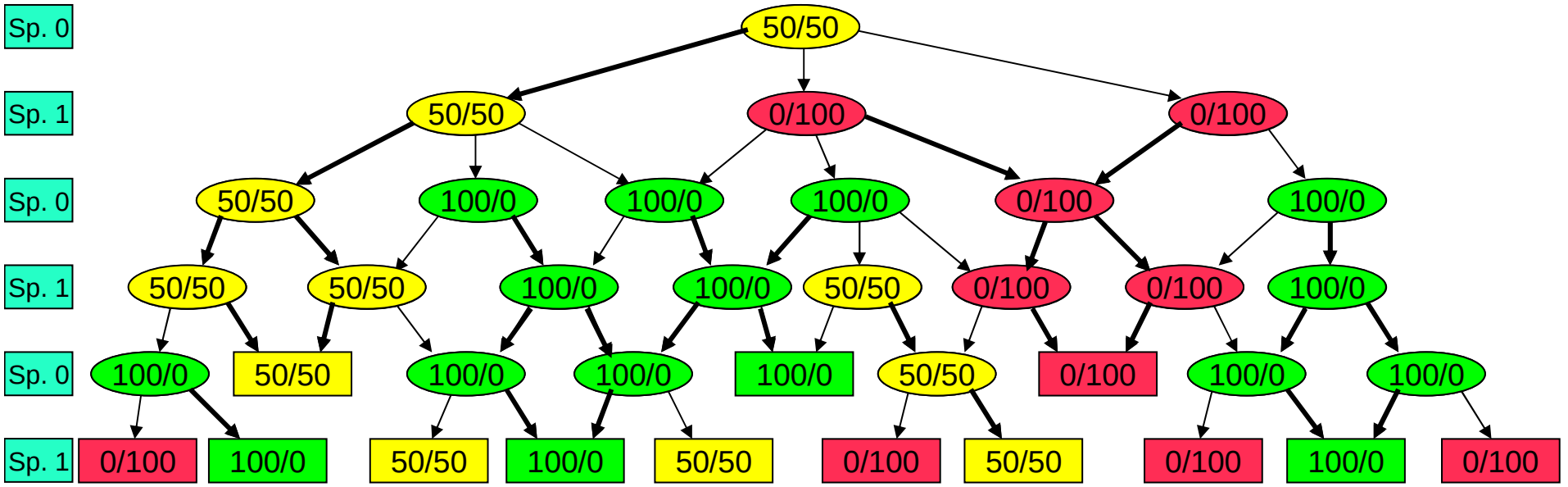
Symbolic Two-Player Zero-Sum

- für jeden spieler $\in \{0, 1\}$
  - $\text{gewonnen}_{\text{spieler}} \leftarrow \text{terminal} \wedge \text{goal}_{\text{spieler},100} \wedge \text{aktiv}_{\text{1-spieler}}$
  - $\text{verloren}_{\text{1-spieler}} \leftarrow \text{terminal} \wedge \text{goal}_{\text{spieler},100} \wedge \text{aktiv}_{\text{spieler}}$
  - $\text{neu} \leftarrow \text{gewonnen}_{\text{spieler}}$
  - solange $\text{neu} \neq \bot$
    - $\text{verloren}_{\text{1-spieler}} \leftarrow \text{verloren}_{\text{1-spieler}} \vee \text{pre-image(neu)}$
    - $\text{neu} \leftarrow \text{strong pre-image}(\text{verloren}_{\text{1-spieler}}) \wedge \neg\text{gewonnen}_{\text{spieler}}$
    - $\text{gewonnen}_{\text{spieler}} \leftarrow \text{gewonnen}_{\text{spieler}} \vee \text{neu}$

Zielbeschreibung mit Gewinn 100 für Spieler spieler

Gegenspieler ist am Zug

Symbolic Two-Player Zero-Sum



Lösen allgemeiner Zweipersonenspiele

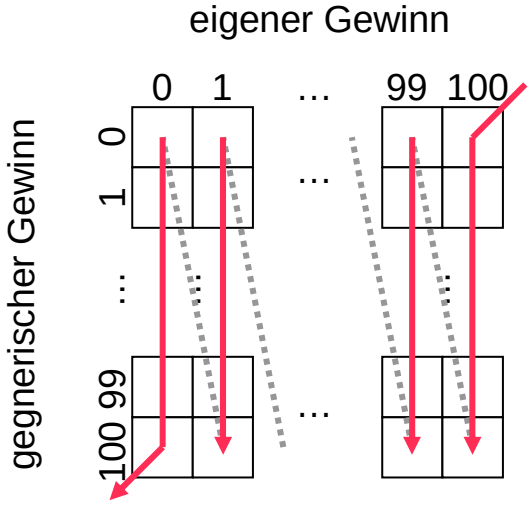
- was tun bei anderen Gewinnverteilungen?
  - Nullsummenspiele mit verschiedenen Bewertungen ((10/90), (75/25), …)
  - allgemeinere Verteilungen ((0/70), (50/25), (100/10), …)
- Möglichkeit:
  - Versuchen, auf Nullsummenspiel mit gewonnen / verloren / unentschieden zurückzurechnen
  - sicherlich möglich bei Nullsummenspielen
    - > 75 → gewonnen
    - < 25 → verloren
    - sonst → unentschieden
  - evtl. auch möglich bei allgemeinen Verteilungen
    - Gewinn Spieler 1 > Gewinn Spieler 2 → gewonnen für Spieler 1
    - Gewinn Spieler 2 > Gewinn Spieler 1 → gewonnen für Spieler 2
    - Gewinn Spieler 1 = Gewinn Spieler 2 → unentschieden

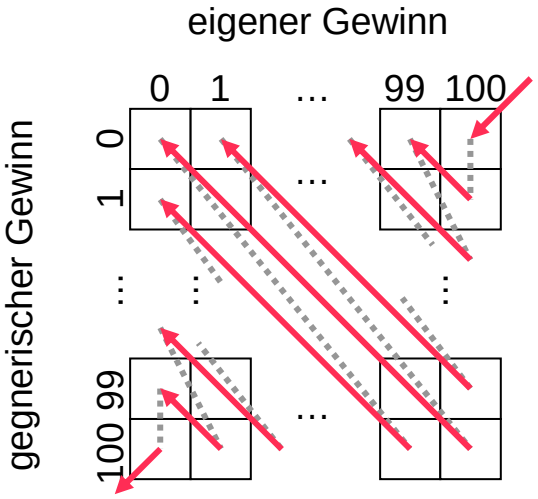
Lösen allgemeiner Zweipersonenspiele

- Aber: exakte Lösung danach unbekannt
  - (10/9) und (100/0) äquivalent
  - (10/0) besser als (99/100)
- Weitere Möglichkeit: Gegenspielermodellierung:
  - was will Gegenspieler erreichen?
    - möglichst viele Punkte?
    - Differenz zu uns möglichst groß?
  - kann von Spielsituation abhängen
    - in Turnier mit vielen Teilnehmern eher versuchen, viele Punkte zu holen, egal wie viele Gegner dabei bekommt
    - in Duell zwischen 2 Spielern eher versuchen, Differenz zu maximieren

Lösen allgemeiner Zweipersonenspiele

- Gewinne als (101x101)-Matrix
- Durchlauf entspricht gewünschtem Verhalten:
- Maximieren eigenen Gewinns (links) vs. Maximieren der Differenz (rechts)





Lösen allgemeiner Zweipersonenspiele

- Versuch, Minimax-Ansatz als symbolische Suche zu formulieren
  - (101x101)-Matrix zum Speichern von optimalen Gewinnen (gemäß Gegnermodell)
  - Starte an Blättern
    - Blätter sind Terminalzustände; damit sofort gelöst
  - alle Vorgänger, deren Nachfolger alle gelöst sind, können gelöst werden
  - Lösung gemäß Durchlaufreihung durch Matrix
  - neue gelöste Zustände in Matrix
  - am Ende, alle Zustände in Matrix
  - [E&K, 2008]

Symbolic Two-Player

- erreichbar ← BFS()
- für alle i, j ∈ {0, 100}
  - $lösung_{i,j}$ ← erreichbar ∧ terminal ∧ $goal_{0,i}$ ∧ $goal_{1,j}$
- gelöst ← $\bigvee_{0 \le i,j \le 100} lösung_{i,j}$
- ungelöst ← erreichbar ∧ ¬gelöst
- solange ungelöst ≠ ⊥
  - für jeden spieler ∈ {0, 1}
    - lösbar ← strong pre-image(gelöst) ∧ ungelöst ∧ $aktiv_{spieler}$
    - falls lösbar ≠ ⊥
      - lösung ← löseZustände(lösung, spieler, lösbar)
      - gelöst ← gelöst ∨ lösbar
      - ungelöst ← ungelöst ∧ ¬lösbar
- gib lösung zurück

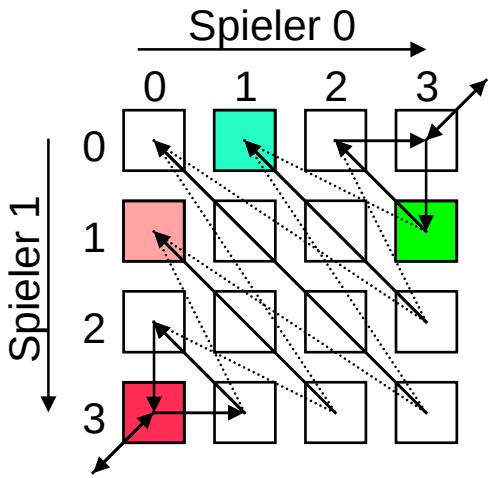
Symbolic Two-Player

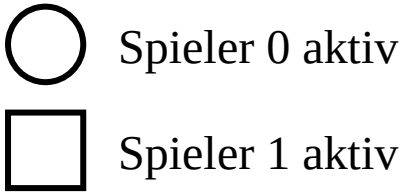
- eigentliches Lösen in löseZustände
  - Matrix durchlaufen
  - überprüfen, ob in übergebenen Zuständen Vorgänger aus aktuellem Bucket enthalten
  - wenn ja, diese Vorgänger in Bucket hinzufügen und aus lösbaren entfernen
  - am Ende: alle lösbaren Zustände in Matrix einsortiert

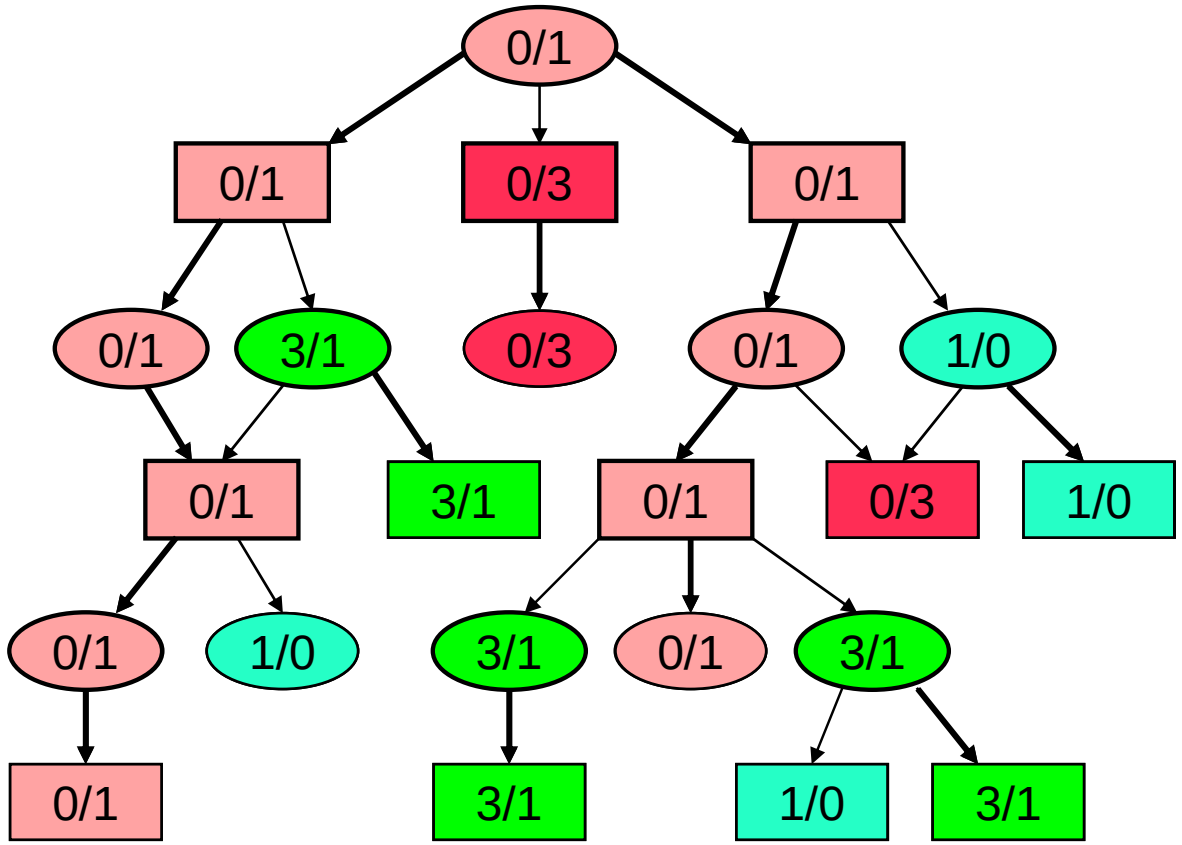
löseZustände(lösung, spieler, lösbar)

- für alle i, j $\in$ {0, 100} in entsprechender Reihenfolge (abhängig von spieler)
  - falls $lösung_{i,j} \neq \perp$
    - neuGelöst $\leftarrow$ pre-image($lösung_{i,j}$) $\wedge$ lösbar
    - $lösung_{i,j} \leftarrow lösung_{i,j} \vee$ neuGelöst
    - lösbar $\leftarrow$ lösbar $\wedge \neg$neuGelöst
    - falls lösbar = $\perp$
      - gib lösung zurück

Lösen allgemeiner Zweipersonenspiele







Lösen allgemeiner Zweipersonenspiele 2

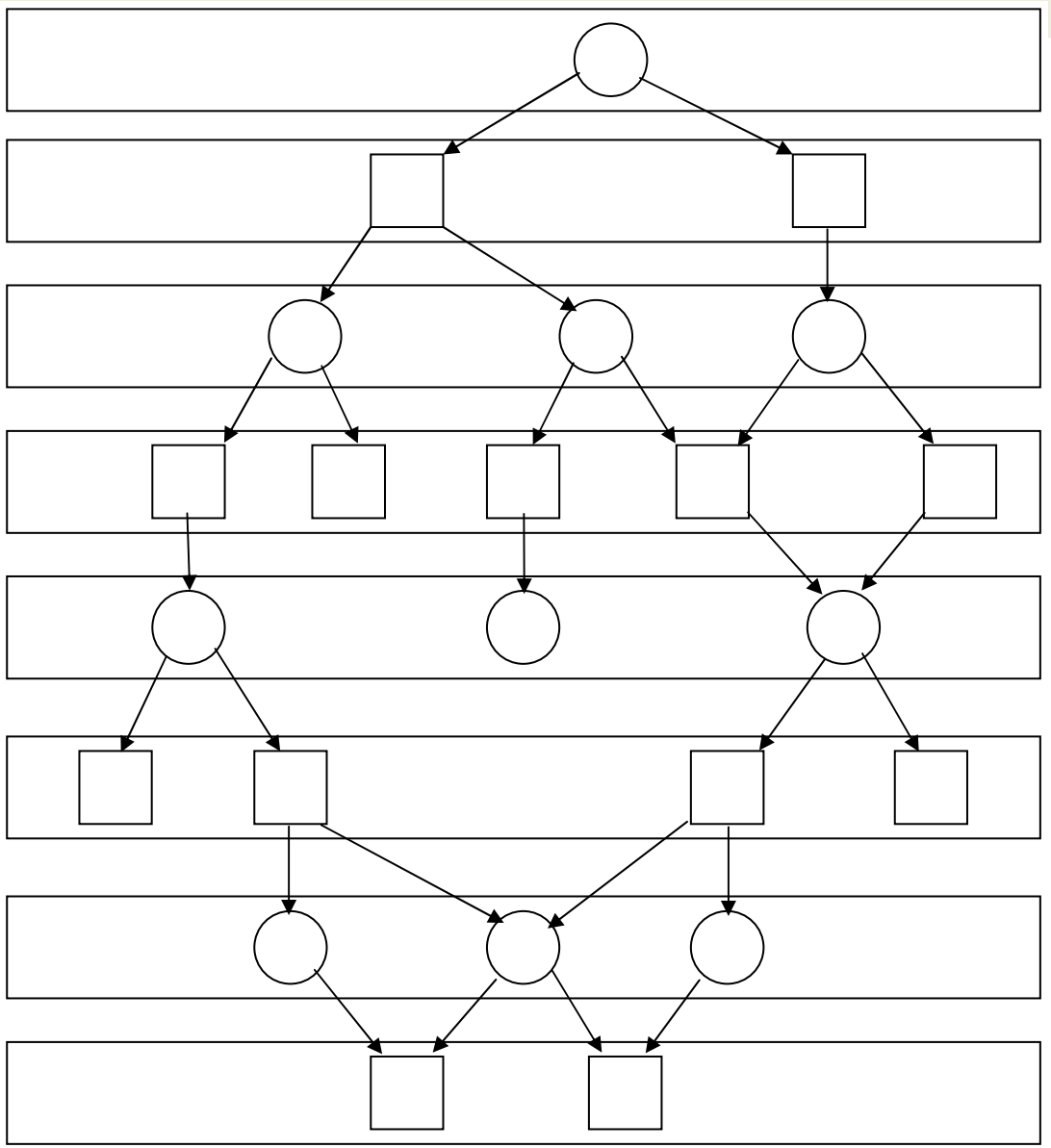
- Etwa bei Vorwärtssuche von Vier Gewinnt:
  - Speichern aller Zustände in einem BDD zeit- und speicherintensiv
  - Alternative: Tiefensuchlayer getrennt verwalten
    - schneller
    - weniger Speicherbedarf
  - Layer können auf Festplatte geschrieben werden
    - Suche jederzeit unterbrechbar
    - kann später in aktueller Schicht fortgesetzt werden

Layered BFS

- aktuell ← init
- layer ← 0
- solange aktuell ≠ ⊥
  - speichere aktuell als $\text{erreichbar}_{\text{layer}}$ auf Festplatte
  - aktuell ← image(aktuell)
  - layer ← layer + 1
- gib layer - 1 zurück

letzte gefunde Schicht leer → layer - 1 ist letzte nicht-leere Schicht

Layered BFS



Spieler 0 aktiv

Spieler 1 aktiv

Layered BFS

- Wichtig: Spiel muss nach endlich vielen Schritten terminieren (im allgemeinen Spiel ohnehin Voraussetzung)
  - hier keine Erkennung von Duplikaten (außer in Layern selbst durch BDDs)
  - Suche würde entsprechend nicht terminieren
  - Aber: Duplikatsprüfung darf hier nicht aktiviert werden (Duplikate wichtig in Rückwärtssuche)

Lösen allgemeiner Zweipersonenspiele 2

- Layer-basierter Ansatz auch für Rückwärtssuche möglich
  - einfacherer Algorithmus
  - keine strong pre-images nötig
  - Aber: Vorwärtssuche zwingend erforderlich
  - in vielen Fällen schneller
    - teils aber erheblich langsamer
  - [K&E, 2010]

Lösen allgemeiner Zweipersonenspiele 2

- Start im letzten nicht-leeren Layer
- enthält nur Terminalzustände
  - können direkt gelöst werden
  - Lösungen werden auf Festplatte gespeichert
- dann vorheriges Layer laden
  - kann Terminalzustände enthalten
    - können wieder direkt gelöst werden
  - restliche Zustände haben alle Nachfolger im nächsten Layer
  - nächstes Layer schon gelöst
  - restliche Zustände entsprechend lösbar
    - Bewertung gemäß Durchlaufordnung (Gegnermodell)

Layer-based Symbolic Two-Player

- layer ← layeredBFS()
- solange layer ≥ 0
  - aktuell ← lade erreichbar$_{layer}$ von Festplatte
  - aktuelleTerminale ← aktuell ∧ terminal
  - aktuell ← aktuell ∧ ¬terminal
  - falls (aktuelleTerminale ≠ ⊥)
    - löseTerminale(aktuelleTerminale, layer)
  - falls (aktuell ≠ ⊥)
    - löseLayer(aktuell, layer)
  - layer ← layer - 1

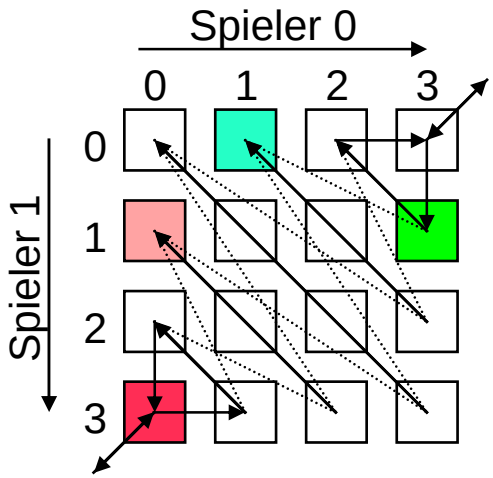
löseTerminale(aktuelleTerminale, layer)

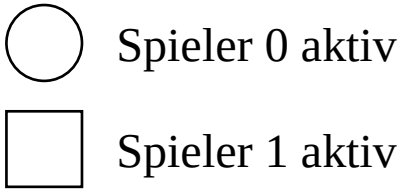
- für alle i, j ∈ {0, 100}
  - $goals_{layer,i,j} \leftarrow aktuelleTerminale \wedge goal_{0,i} \wedge goal_{1,j}$
  - speichere $goals_{layer,i,j}$ auf Festplatte
  - $aktuelleTerminale \leftarrow aktuelleTerminale \wedge \neg goals_{layer,i,j}$

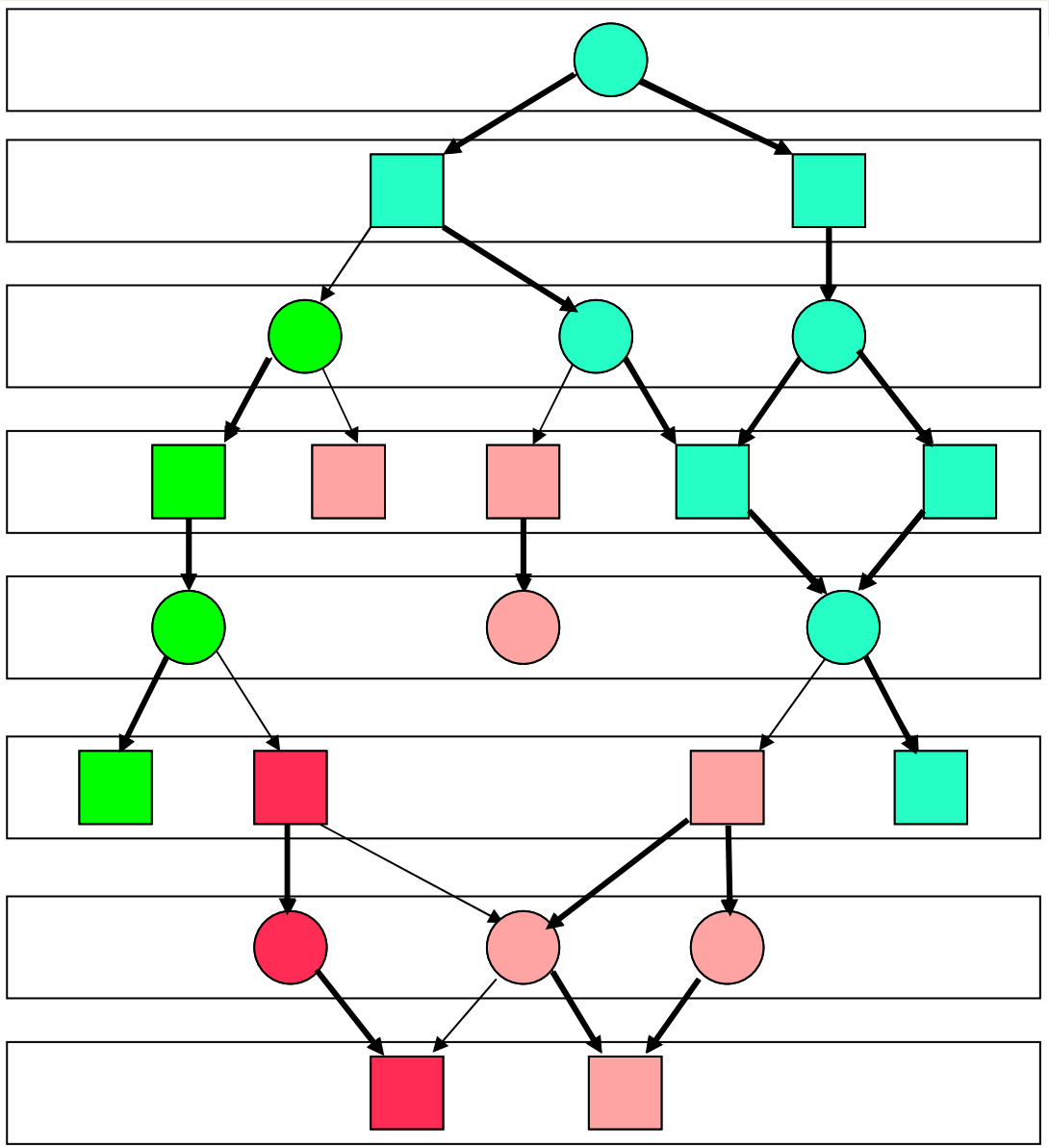
löseLayer(aktuell, layer)

- für alle i, j $\in$ {0, 100} in entsprechender Reihenfolge (abhängig von aktivem Spieler)
  - nachfolger1 $\leftarrow$ lade $goals_{layer+1,i,j}$ von Festplatte
  - nachfolger2 $\leftarrow$ lade $gewinne_{layer+1,i,j}$ von Festplatte
  - nachfolger $\leftarrow$ nachfolger1 $\vee$ nachfolger2
  - $gewinne_{layer,i,j} \leftarrow$ aktuell $\wedge$ pre-image(nachfolger)
  - speicher $gewinne_{layer,i,j}$ auf Festplatte
  - aktuell $\leftarrow$ aktuell $\wedge \neg gewinne_{layer,i,j}$

Beispiel


Spieler 0
0
1
2
3
Spieler 1
0
1
2
3


Spieler 0 aktiv
Spieler 1 aktiv



Wann funktioniert dieses Verfahren?

- wenn wir einen inkrementellen Progress Measure finden können:
  - Progress Measure: Maß für Fortschritt von Zustand zu Nachfolgezustand (oder zu Zustand nach 2 Halbzügen)
  - inkrementell: Fortschritt erhöht sich um exakt 1
- Beispiele:
  - Tic-Tac-Toe (# Marker auf dem Brett)
  - Vier Gewinnt (# eingefügte Steine)
  - Sheep and Wolf (Summe der Zeilen der Schafe)

Wann funktioniert es nicht?

- wenn es kein solches Maß gibt
  - selbst Spiele, die eine Art von Progress Measure haben, das aber nicht inkrementell ist, sind schlecht
  - denn: Resultat: Duplikate in verschiedenen Layern
  - daher, mehr Zustände in mehr Layern zu lösen
- Beispiele:
  - Nim (# genommener Streichhölzer)
  - Chomp (# gegessener Stücke)

Problematische Spiele

- Chomp
  - mit Duplikatselimination:
    - 8 Layer
    - 25 734 Zustände
  - Erhalten von Duplikaten:
    - 56 Layer
    - 162 591 Zustände
- Nim
  - mit Duplikatselimination:
    - 5 Layer
    - 129 776 Zustände
  - Erhalten von Duplikaten:
    - 63 Layer
    - 1 866 488 Zustände
- Glücklicherweise, nur wenige solcher Spiele vorhanden

Ergebnisse

- Genutzte Hardware:
  - 2.67 GHz CPU
  - 12 GB RAM
- Vergleich zum ersten Ansatz

Spiel	#Zustände	Zeit (neu)	Zeit (alt)	Ergebnis
Chomp	25 734	0:10	0:02	100/0
Clobber (4 x 5)	26 787 440	10:43	1:18:51	30/0
Connect 4 (5 x 6)	1 044 334 437	31:37	2:14:52	50/50
Cubi Cup 5	7 369 419 770	8:21:09	o.o.m.	100/0
Nim	129 776	0:10	0:01	100/0
Sheep and Wolf	783 163	0:23	0:57	0/100

Quellen

- R.E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, pp. 677-691, 1986
- S. Edelkamp & P. Kissmann: *Symbolic Exploration for General Game Playing in PDDL*, ICAPS-Workshop on Planning in Games, 2007
- S. Edelkamp: *Symbolic Exploration in Two-Player Games: Preliminary Results*. AIPS-Workshop on Model Checking, pp. 40-48, 2002
- S. Edelkamp & P. Kissmann: *Symbolic Classification of General Two-Player Games*, KI, pp. 185-192, 2008
- P. Kissmann & S. Edelkamp: *Layer-Abstraction for Symbolically Solving General Two-Player Games*, SoCS, pp. 63-70, 2010