Handlungsplanung und Allgemeines Spiel

„Instanziierung Allgemeiner Spiele“

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 gut, um Spiele zu lösen
  - Problem: Wie viele Variablen (= grundierte Fluents) gibt es?
  - muss geklärt sein, bevor erstes BDD erzeugt
- Beim Spielen, oft Prolog genutzt
  - kann recht effizient gültige Züge, Nachfolger etc. berechnen
  - erzeugt aber deutlichen Laufzeit-Overhead
  - Spieler, der instanziierte Eingabe nutzt, kann schneller operieren

Motivation

- reine Monte-Carlo Suche
- 10 Sekunden Laufzeit
- Vergleich von Expansionsanzahlen

Game	Prolog	Inst.	Faktor
Asteroids (serial)	59 364	219 575	3,70
Beatmania	28 680	3 129 300	109,11
Chomp	22 020	1 526 445	69,32
Connect Four	44 449	2 929 996	45,45
Hanoi	84 785	7 927 847	93,51
Lightsout	28 800	7 230 080	251,04
Pancakes 6	154 219	2 092 308	13,57
Peg Solitaire	19 951	1 966 075	98,55
Sheep & Wolf	20 448	882 738	43,17
Tic-Tac-Toe	65 864	5 654 553	85,85

Motivation

- gleiches Problem auch in Handlungsplanung
  - PDDL enthält auch Variablen
  - Folgern mit Nutzung von Variablen langsam
  - erfolgreiche Teilnehmer letzter Planungs-Meisterschaft nutzen Instanziierer
- Aber:
  - einfache Übersetzung von GDL zu PDDL nicht möglich
  - in PDDL, Frame nicht explizit modelliert
  - in GDL, getrennte Vorbedingungen (legals) und Effekte (next)
  - bei Mehrpersonenspielen, schwierig, kombinierte Züge in PDDL zu übersetzen

Ablauf der Vorlesung

- Instanziierungsprozess
- Berechnung der Disjunktiven Normalform
- Berechnung von Obermengen
- Instanziierung von Formeln
- Finden von Gruppen gegenseitig ausschließender Fluents
- Entfernung von Hilfsprädikaten
- Ausgabegenerierung

Instanziierungsprozess

- Eingabe: Spiel in GDL
  - bestehend aus Menge von Relationen
  - Relationen als Klauseln (Head einfaches Prädikat, Body als Boolesche Formel)
  - Formeln über
    - Konjunktionen
    - Disjunktionen
    - Negationen
    - Literale
      - Prädikate
      - negierte Prädikate
  - Prädikate können Variablen enthalten
  - Prädikate: Fluents und Hilfsprädikate
- Ziel: identisches Spiel in variablenfreiem Format
- [K&E, 2010]

Instanziierungsprozess

1. Parsen der GDL Eingabe
2. Berechnung von Disjunktiver Normalform (DNF)
3. Berechnung von (instanziierten) Obermengen erreichbarer Prädikate, Züge, Hilfsprädikate
4. Instanziierung aller Formeln
5. Finden von Gruppen gegenseitig ausschließender Prädikate
6. Entfernung von Hilfsprädikaten
7. Erzeugung der Ausgabe

Berechnung der DNF

- Disjunktive Normalform:
  - Disjunktion von Konjunktionen von Literalen
  - etwa $(X \wedge Y \wedge Z) \vee (X \wedge \neg Y \wedge \neg Z) \vee (\neg X \wedge \neg Y \wedge Z) \vee (X \wedge \neg Y \wedge Z)$
  - also:
    - Negationen nur noch vor Prädikaten
    - Konjunktionen nur über Literale
    - Disjunktion nur über Konjunktionen (und Literale)
    - keine tieferen Schachtelungen zulässig

Berechnung der DNF

- Negation nach Innen bringen
  - Anwendung von DeMorgan'schen Gesetzen:
    - $\neg(X \wedge Y) \equiv \neg X \vee \neg Y$
    - $\neg(X \vee Y) \equiv \neg X \wedge \neg Y$
  - Auflösen doppelter Negationen
    - $\neg\neg X \equiv X$

Berechnung der DNF

- Falls Konjunktion von Disjunktionen gefunden
  - Vertauschung von Konjunktion und Disjunktion durch Anwendung von Distributivgesetzen
    - $X \wedge (Y \vee Z) \equiv (X \wedge Y) \vee (X \wedge Z)$
    - $X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z)$
- Zusammenführen geschachtelter Konjunktionen
  - $X \wedge (Y \wedge Z) \equiv X \wedge Y \wedge Z$
- Zusammenführen geschachtelter Disjunktionen
  - $X \vee (Y \vee Z) \equiv X \vee Y \vee Z$

Berechnung der DNF

- Beispiel: in 8puzzle:
- ```
  (<= (legal player (move ?x ?y))
      (true (cell ?u ?y b))
      (or
          (succ ?x ?u)
          (pred ?x ?u)
      )
  )
  ```
- body:
  - true(cell) $\wedge$ (succ $\vee$ pred)
    $\equiv$ (true(cell) $\wedge$ succ) $\vee$ (true(cell) $\wedge$ pred)
  ```

Berechnung der DNF

- damit entsprechende legal-Relation durch 2 Relationen darstellbar
- ```
  (<= (legal player (move ?x ?y))
      (true (cell ?u ?y b))
      (succ ?x ?u)
  )
  ```
- ```
  (<= (legal player (move ?x ?y))
      (true (cell ?u ?y b))
      (pred ?x ?u)
  )
  ```
- Disjunktion implizit in GDL enthalten
- Ergebnis: Reihe von Relationen mit nur Konjunktionen von Literalen im Body

Berechnung der DNF

- Achtung: Vertauschen von Konjunktionen und Disjunktionen kann Formeln exponentiell vergrößern
- Aber: in GDL typischerweise (durch Nutzung von Hilfsprädikaten)
  - wenige Operanden
  - geringe Schachtelungstiefe
- damit DNF problemlos berechenbar

Berechnung von Obermengen

- um Formeln zu instanziieren, nötig, alle möglichen Instanziierungen von Prädikaten etc. zu kennen
- dafür mehrere Möglichkeiten
  - Fixpunkt-Suche mit Prolog
  - Auswertung von Abhängigkeitsgraphen
  - Nutzen von Answer Set Programming (hier nicht)

Fixpunkt-Suche mit Prolog

- in aktueller Spielbeschreibung, Negationen nur vor Prädikaten
- Erstellung neuer Spielbeschreibung (in Prolog)
  - negierte Prädikate aus Spielbeschreibung entfernt
  - ohne Terminalbedingungen
  - ohne Zielbeschreibungen

Fixpunkt-Suche mit Prolog

- Start an Initialzustand
- wiederhole
  - Berechnung aller gültigen Züge (instanziiert)
  - Speichern in Wissensbasis
  - Berechnung aller gültigen Nachfolgefluents (instanziiert)
  - Speichern in Wissensbasis
- bis #gültiger Züge und #gültiger Nachfolgefluents unverändert
- Prolog-Anfragen für alle Hilfsprädikate liefern alle möglichen Instanziierungen für diese
- aus Wissensbasis wird nichts entfernt

Fixpunkt-Suche mit Prolog

- durch Entfernung von Negationen:
  - was einmal erfüllbar war, bleibt bis zum Ende erfüllbar
- alles, was in vollständigem Spiel erreichbar, auch in vereinfachter Form erreichbar
- aber nicht alles, was in vereinfachter Form erreichbar auch in vollständigem Spiel erreichbar
  - damit, gefundene Mengen Obermengen aller real erreichbaren Prädikate und Züge
  - Vorteil gegenüber vollständigem Spiel: viel schneller

Fixpunkt-Suche mit Prolog

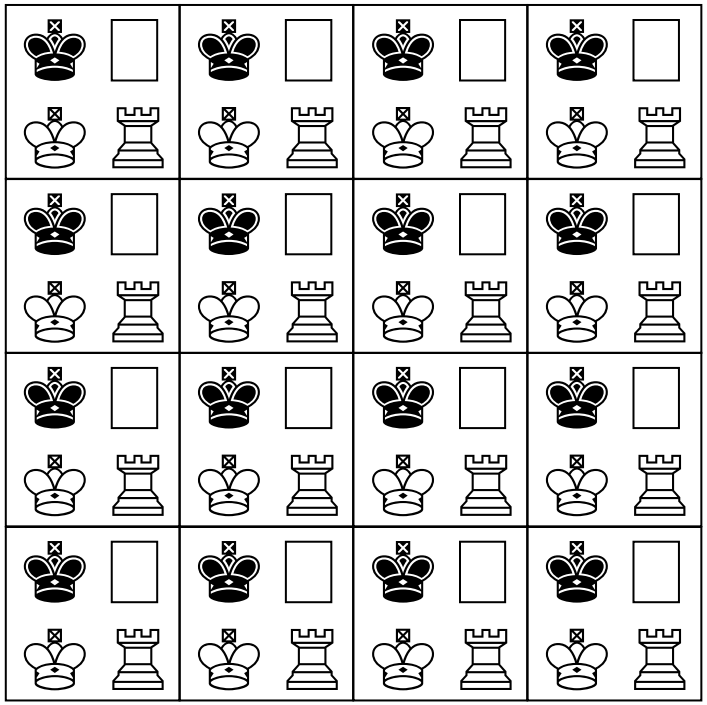
```
(<= (legal black (move bk ?u ?v ?x ?y))
    (true (control black))
    (true (cell ?u ?v bk))
    (kingmove ?u ?v ?x ?y)
    (true (cell ?x ?y b))


)
```

- liefert alle möglichen Königszüge als legal, egal ob König oder Zielfeld angegriffen

Fixpunkt-Suche mit Prolog

control(white)
control(black)



step(1)
step(2)
step(3)
step(4)
step(5)
step(6)
step(7)
step(8)
step(9)
step(10)

Fixpunkt-Suche mit Prolog

- Vorteil: Obermengen relativ klein
- übliches Problem: wegen vieler Möglichkeiten, Formeln zu erfüllen, oft noch zu langsam
  - mögliche Abhilfe: Nutzen von Abhängigkeitsgraphen

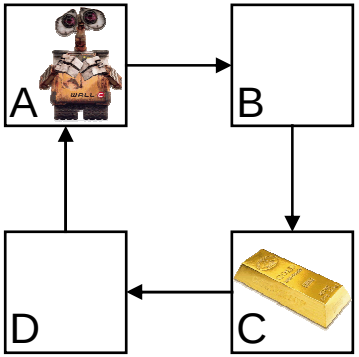
Obermengen mit Abhängigkeitsgraphen

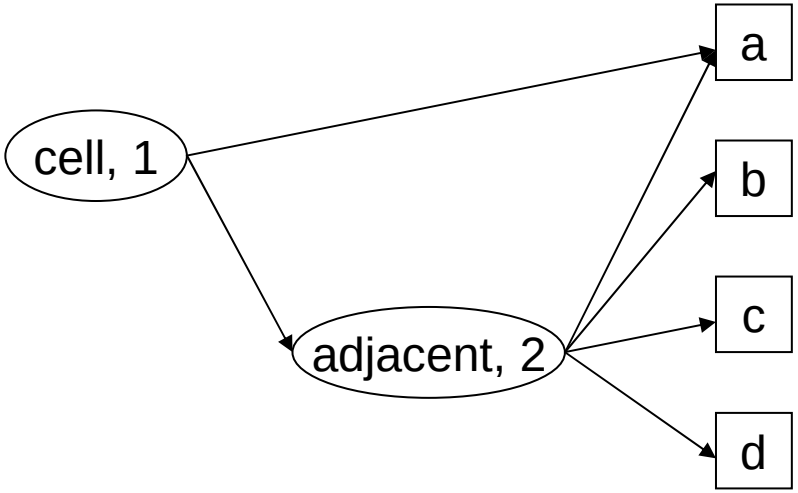
- Abhängigkeitsgraph für Formeln schon bei Evaluationsfunktionen genutzt
- für Formel mit Head h und Body b_1, b_2, …, b_n
  - falls k-ter Parameter von h Konstante c
    - füge Kante (h,k → c) zu Graph hinzu
  - falls i-ter Parameter von b_j k-tem Parameter von h entspricht
    - füge Kante (h,k → b_j,i) zu Graph hinzu

Obermengen mit Abhängigkeitsgraphen

- `(init (cell a))`
- `(<= (next (cell ?y))`
  `(does robot move)`
  `(true (cell ?x))`
  `(adjacent ?x ?y))`
- `(adjacent a b)`
- `(adjacent b c)`
- `(adjacent c d)`
- `(adjacent d a)`

- Definitionsbereich von `cell`:
- `(cell a)`
- `(cell b)`
- `(cell c)`
- `(cell d)`





Obermengen mit Abhängigkeitsgraphen

- Problem:
  - in Abhängigkeitsgraphen keine Information über Zusammenhänge einzelner Parameter eines Prädikats gespeichert
  - damit Obermengen zu ungenau

Obermengen mit Abhängigkeitsgraphen

- Beispiel:
  - ```
    (<= (legal black (move bk ?u ?v ?x ?y))
        (true (control black))
        (true (cell ?u ?v bk))
        (kingmove ?u ?v ?x ?y)
        (true (cell ?x ?y b))
        (not (attacked bk ?x ?y))
        (not (guarded ?x ?y))
    )
    ```
  - move/1: bk
  - move/2: abhängig von cell/1
  - move/3: abhängig von cell/2
  - move/4: abhängig von cell/1
  - move/5: abhängig von cell/2
  - damit auch Zug (move bk 1 1 4 3) in Obermenge enthalten
    ```

Obermengen mit Abhängigkeitsgraphen

- Abhilfe:
  - jede Formel für gegebenen (instanziierten) Head auswerten
    - wenn keine Ersetzung von Prädikaten durch instanziierte Prädikate möglich, so dass alle Variablen gleich belegt, verwerfe Head aus Obermenge
    - iteriere, bis keine Änderung in Obermenge
  - nach Instanziierung von Formeln (nächster Schritt):
    - führe Erreichbarkeitsanalyse in Negations-befreiter Beschreibung durch
    - Ergebnis: identisch zu Prolog Analyse
    - aber: erheblich schneller, da kein Prolog Aufruf nötig

Berechnung von Obermengen

- Ergebnis:
  - Nutzung von Prolog liefert (zunächst) schärfere Obermengen
    - kann aber langsam sein, wenn viele Duplikate / viele Möglichkeiten, eine Formel zu erfüllen
  - Nutzung von Abhängigkeitsgraphen liefert oft schlechte Obermengen
    - Instanziierung von Formeln dann viel langsamer (da viel mehr Prädikate präsent)
    - damit Gesamtlaufzeit teils schlechter als bei Prolog
  - nach letztem Optimierungsschritt bei Abhängigkeitsgraphen, Ergebnis aber identisch

Instanziierung von Formeln

- Prädikate (Fluents und Hilfsprädikate) und Züge in instanziierter Form bekannt
- nächster Schritt: komplette Formeln instanziieren
- naïve Idee:
  - für jedes Prädikate jede mögliche Instanziierung einsetzen
  - wenn für Formel alle Prädikate eingesetzt, überprüfen, ob möglich
  - Problem: viele teilinstanziierte Formeln, hoher Speicherbedarf, langsam

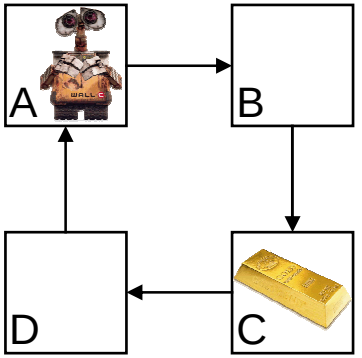
Instanziierung von Formeln

- unser Ansatz:
  - zunächst bestimmen, welche Variablennamen in Formel auftauchen
  - Matrix erzeugen
    - Spalten: unterschiedliche Variablennamen
    - Zeilen: ihre möglichen Instanziierungen (gruppiert nach zugehörigem Prädikat)
  - mögliche Variablen-Instanziierungen
    - jedes Prädikat aus Konjunktion betrachten
    - alle instanziierten Version davon finden
  - volle Instanziierung
    - mögliche Kombinationen aller Prädikate kombinieren
    - sicherstellen, dass gleichzeitig anwendbar (identische Variablen gleich belegt, kein Widerspruch zu distinct)
    - Variablen entsprechend setzen und instanziierte Formel speichern

Instanziierung von Formeln

	?y	?x
(next (cell ?y))	a b c d	
(true (cell ?x))		a b c d
(adjacent ?x ?y)	b c d a	a b c d

```
(<= (next (cell ?y))
    (does robot move)
    (true (cell ?x))
    (adjacent ?x ?y))
```




```
(<= (next (cell b))
    (does robot move)
    (true (cell a))
    (adjacent a b))

(<= (next (cell c))
    (does robot move)
    (true (cell b))
    (adjacent b c))

(<= (next (cell d))
    (does robot move)
    (true (cell c))
    (adjacent c d))

(<= (next (cell a))
    (does robot move)
    (true (cell d))
    (adjacent d a))
```

Zwischenbilanz

- jetzt:
  - alle Formeln fertig instanziiert
  - Obermenge aller erreichbaren Prädikate (Fluents, Hilfsprädikate) und Züge bekannt
- was noch fehlt:
  - Ausgabe
  - evtl. Entfernen von Hilfsprädikaten
  - Finden von Gruppen gegenseitig ausschließender Fluents

Finden von Gruppen

- je weniger Variablen für Zustandskodierung, desto besser
- sich gegenseitig ausschließende Variablen können zusammen gespeichert werden
  - n sich gegenseitig ausschließende Boolesche Variablen durch log(n) binäre Variablen darstellbar

Finden von Gruppen

- zum Finden, nutzen von Verfahren ähnlich dem Finden dynamischer Strukturen für Evaluationsfunktionen
  - Finden von Eingabe- und Ausgabeparametern
  - zur Erinnerung:
    - verschiedene Prädikate mit identischen Eingabe- aber unterschiedlichen Ausgabeparametern niemals gleichzeitig wahr
    - schließen sich also gegenseitig aus

Finden von Gruppen

- für jedes Prädikate, Hypothese, welche Parameter Eingabe- und Ausgabeparameter
- Überprüfung durch Simulation
- Am Ende: Wenn mehrere möglichkeiten, Wahl von Eingabe- und Ausgabeparametern so, dass Kodierung minimal

Entfernung von Hilfsprädikaten

- Hilfsprädikate ähnlich zu derived predicates in Handlungsplanung
- dienen vor allem Verkleinerung von (textueller) Spielbeschreibung
- kapseln ganze Formeln in einfachem Prädikat
- eigentlicher Zustand jedoch aus Fluents vollständig beschreibbar
- durch Entfernen von Hilfspärdikaten damit weitere Verkleinerung von Zuständen

Entfernung von Hilfsprädikaten

- Abhängigkeiten von Hilfsprädikaten azyklisch (gemäß GDL)
  - Body von Formel zur Berechnung von Hilfsprädikat nicht direkt oder indirekt abhängig von entsprechendem Head
- damit Head quasi Makro für Body
- jedes Auftauchen von Head durch Body ersetzbar
- da azyklisch, bei geschickter Ersetzungsreihenfolge schnell alle Hilfsprädikate ersetzt
- Reihenfolge gemäß topologischer Sortierung
- [Borowsky & Edelkamp, 2008]

Entfernung von Hilfsprädikaten

- Bispiel: Tic-Tac-Toe
  - `(<= goal xplayer 100 (line x))`
  - `(<= (line x)`
    `(or (row x) (column x) (diagonal x)))`
  - `(<= (row x)`
    `(or`
    `(and (cell 1 1 x) (cell 2 1 x) (cell 3 1 x))`
    `(and (cell 1 2 x) (cell 2 2 x) (cell 3 2 x))`
    `(and (cell 1 3 x) (cell 2 3 x) (cell 3 3 x))))`
  - `(<= (column x)`
    `(or`
    `(and (cell 1 1 x) (cell 1 2 x) (cell 1 3 x))`
    `(and (cell 2 1 x) (cell 2 2 x) (cell 2 3 x))`
    `(and (cell 3 1 x) (cell 3 2 x) (cell 3 3 x))))`
  - `(<= (diagonal x)`
    `(or`
    `(and (cell 1 1 x) (cell 2 2 x) (cell 3 3 x))`
    `(and (cell 1 3 x) (cell 2 2 x) (cell 3 1 x))))`

Entfernung von Hilfsprädikaten

- Abhängigkeiten also:
  - goal: abhängig von line
  - line: abhängig von row, column und diagonal
  - row: nur von Fluents abhängig
  - column: nur von Fluents abhängig
  - diagonal: nur von Fluents abhängig
- topologische Sortierung etwa:
  - (column, diagonal, row, line, goal)
  - alle Hilfsprädikate, die nur von Fluents abhängen, zuerst ersetzen
  - dann überprüfen, ob weitere nur noch von Fluents abhängig
    - wenn ja, auch diese ersetzen
  - so fortfahren, bis alle ersetzt
- am Ende: alle Hilfsprädikate ersetzt; Formeln für Hilfsprädikate können entfernt werden

Entfernung von Hilfsprädikaten

- Beispiel: Tic-Tac-Toe
  - `(<= goal xplayer 100 (line x))`
  -
```
(<= (line x)
    (or
    (or
        (and (cell 1 1 x) (cell 2 1 x) (cell 3 1 x))
        (and (cell 1 2 x) (cell 2 2 x) (cell 3 2 x))
        (and (cell 1 3 x) (cell 2 3 x) (cell 3 3 x)))
    )
    (or
        (and (cell 1 1 x) (cell 1 2 x) (cell 1 3 x))
        (and (cell 2 1 x) (cell 2 2 x) (cell 2 3 x))
        (and (cell 3 1 x) (cell 3 2 x) (cell 3 3 x)))
    )
    (or
        (and (cell 1 1 x) (cell 2 2 x) (cell 3 3 x))
        (and (cell 1 3 x) (cell 2 2 x) (cell 3 1 x)))))
```

Entfernung von Hilfsprädikaten

- Beispiel: Tic-Tac-Toe
  - `(<= goal xplayer 100 (line x))`
  -
```
(<= (line x)
    (or
        (and (cell 1 1 x) (cell 2 1 x) (cell 3 1 x))
        (and (cell 1 2 x) (cell 2 2 x) (cell 3 2 x))
        (and (cell 1 3 x) (cell 2 3 x) (cell 3 3 x))
        (and (cell 1 1 x) (cell 1 2 x) (cell 1 3 x))
        (and (cell 2 1 x) (cell 2 2 x) (cell 2 3 x))
        (and (cell 3 1 x) (cell 3 2 x) (cell 3 3 x))
        (and (cell 1 1 x) (cell 2 2 x) (cell 3 3 x))
        (and (cell 1 3 x) (cell 2 2 x) (cell 3 1 x))))
```

Entfernung von Hilfsprädikaten

- Beispiel: Tic-Tac-Toe
  - ```
    (<= goal xplayer 100
        (or
            (and (cell 1 1 x) (cell 2 1 x) (cell 3 1 x))
            (and (cell 1 2 x) (cell 2 2 x) (cell 3 2 x))
            (and (cell 1 3 x) (cell 2 3 x) (cell 3 3 x))
            (and (cell 1 1 x) (cell 1 2 x) (cell 1 3 x))
            (and (cell 2 1 x) (cell 2 2 x) (cell 2 3 x))
            (and (cell 3 1 x) (cell 3 2 x) (cell 3 3 x))
            (and (cell 1 1 x) (cell 2 2 x) (cell 3 3 x))
            (and (cell 1 3 x) (cell 2 2 x) (cell 3 1 x))))
    ```
    ```

Ausgabegenerierung

- mehrere mögliche Formate
  - instanziiertes GDL
  - GDDL (instanziiert, aber näher an PDDL)

Instanziiertes GDL

- alle Formeln komplett instanziiert
- Ausgabe straight-forward
  - vollständige Ausgabe ohne Hilfsprädikate

GDDL

- ähnlich PDDL, bestehend aus 3 Dateien
  - Domänendatei: Informationen über Züge und Siegbedingungen
  - Problemdatei: Informationen über Initial- und Terminalzustand
  - Partitionsdatei: Informationen über Gruppen sich gegenseitig ausschließender Prädikate
- Problemdatei und Partitionsdatei direkt möglich

GDDL - Domänendatei

- Beschreibung von Zügen durch Multi-Actions
  - Verknüpfung von Zügen aller Spieler
  - Verbindet legal- und next-Relationen für entsprechende Züge

GDDL - Domänendatei

```
(:multi-action
 :player xplayer
 :action mark-2-2
 :player oplayer
 :action noop
 :global-precondition
    (and (cell-1-1-b) (control-xplayer))
 :precondition
    (foo)
 :effect
    (cell-1-1-x)
 :precondition
    (not (foo))
 :effect
    (cell-1-1-b)
 :precondition
    (cell-1-1-x)
 :effect
    (cell-1-1-x)
 ...
)
```

global-precondition: Konjunktion aller relevanten legal-Relationen

precondition/effect Paare: aus relevanten next-Relationen

foo: ausgezeichneter Prädikatsname (immer erfüllbar)

GDDL - Domänendatei

- welche Züge können kombiniert werden?
  - erstmal alle
  - dann prüfen, ob entsprechende legal-Relationen nicht gleichzeitig erfüllbar (mit Hilfe von Gruppen sich gegenseitig ausschließender Prädikate)
  - Alternative: während Simulation (in Gruppenfindung) herausfinden, ob Spiel turn-taking und welches noop-Züge sind
    - dann: alle möglichen Züge von einem Spieler kombiniert mit allen mögliche noop-Zügen aller anderer Spieler

GDDL - Domänendatei

- wie Precondition/Effect Paare berechnen?
  - resultieren aus next-Relationen
  - in allen next-Relationen does-Terme durch $\top$ bzw. $\bot$ ersetzen
  - Formeln vereinfachen
  - Resultat ausgeben
    - $\top$ wird zu foo → Effekt gilt in Nachfolgezustand bei Anwendung dieser Züge immer
    - $\bot$ wird zu (not foo) → Effekt gilt in Nachfolgezustand bei Anwendung dieser Züge nie

GDDL - Domänendatei

- Beschreibung von Siegbedingungen ziemlich einfach:
  - ```
    (:reward
     :player xplayer
     :value 100
     :condition
        (or
            (and (cell 1 1 x) (cell 2 1 x) (cell 3 1 x))
            (and (cell 1 2 x) (cell 2 2 x) (cell 3 2 x))
            (and (cell 1 3 x) (cell 2 3 x) (cell 3 3 x))
            (and (cell 1 1 x) (cell 1 2 x) (cell 1 3 x))
            (and (cell 2 1 x) (cell 2 2 x) (cell 2 3 x))
            (and (cell 3 1 x) (cell 3 2 x) (cell 3 3 x))
            (and (cell 1 1 x) (cell 2 2 x) (cell 3 3 x))
            (and (cell 1 3 x) (cell 2 2 x) (cell 3 1 x)))
    )
    ```
    ```

Quellen

- P. Kissmann & S. Edelkamp: *Instantiating General Games using Prolog or Dependency Graphs*, KI, pp. 255-262, 2010
- B.U. Borowsky & S. Edelkamp: *Optimal Metric Planning with State Sets in Automata Representation, AAAI*, pp. 874-879, 2008