UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Miriam Bützken, Andrea Matuszewski, Rachid Karmouni, Michael Nelskamp, Arne Wiggers,Abdelaziz Elalaoui, Khalid Lahiane, Mohammed Nazih, Kenneth Kahl und Roman Klinger

Moderne Handlungsplanung

Endbericht der Projektgruppe ModPlan

August 31, 2005

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl 5

Fachbereich Informatik
Universität Dortmund

Betreuer:
Stefan Edelkamp
Tilman Mehler, Shahid Jabbar

GERMANY · D-44221 DORTMUND


PLANNING
Planning is like a compass. It keeps you on course.

Inhaltsverzeichnis

Kapitel 1.

Integrierte Arbeitsplattform für Moderne Handlungsplanung

Miriam Bützken, Stefan Edelkamp, Abdelaziz Elalaoui, Kenneth Kahl, Rachid Karmouni, Roman Klinger, Khalid Lahiane, Andrea Matuszewski, Tilman Mehler, Mohammed Nazih, Michael Nelskamp und Arne Wiggers

1.1. Einleitung

Der Fokus der Forschung im Bereich Handlungsplanung verschiebt sich zunehmend in Richtung praktischer Probleme mit Szenarien der Logistik, der Fahrstuhlkoordination, Raum- und Luftfahrt, Spielen, des Handheld-Setup, der Softwareverifikation, der Diagnose von (Strom-)Netzwerken und Ölpipelines, etc., wie der Umfang der aktuell verwendeten Benchmarks [24] auf internationalen Planungswettbewerben [35, 1, 33, 12] zeigt.

Angespornt durch den Erfolg bei den Wettbewerben steigert sich die Effizienz der Planungstechnologien kontinuierlich. Viele aktuelle Planungssysteme können komplexe Probleme schnell lösen. Was bisher noch fehlt, sind intelligente Werkzeuge, die einen Großteil des Designprozesses von neuen Domänen übernehmen und gleichzeitig State-Of-The-Art Planer integrieren.

Im domänenunabhängigen Planen sollten alle Berechnungsschritte automatisiert sein. Bei den meisten existierenden Planungssystemen jedoch können automatisch inferierte Entscheidungen durch möglichst eingeschränkte Benutzereingaben optimiert werden. Als Konsequenz haben wir eine Arbeitsplattform entwickelt, in der Möglichkeiten bereitstehen, auf Ausgaben aus statischen Analysen zuzugreifen und diese zu modifizieren. Des weiteren beinhaltet die Werkbank Werkzeuge um die Erstellung von neuen Domänen zu vereinfachen, neue bzw weiterentwickelte Planer, und Umgebungen zur Analyse und Optimierung des Planungsprozesses. Die Werkbank beherrscht große Teile der derzeitigen Planbeschreibungssprache[1], inklusive zeitlichen Literalen und abgeleiteten Prädikaten.

Im Folgenden werden die einzelnen Programmteile kurz beschrieben. Der genauere strukturelle Aufbau, die Anwendungsweise und die algorithmischen Abläufe werden in den einzelnen Kapiteln differenziert erläutert.

1.2. Architektur der Werkbank

Die *ModPlan*-Werkbank ist als integrierte Benutzeroberfläche für das Windows Betriebssystem mit einfacher Anpassung an Unix/Linux entwickelt worden. Die Architektur ist in Schichten unterteilt. Interaktionen sind mittels der übergeordneten Benutzeroberfläche, Steuerung der Plananimation (in Vega), XML Editierung (in Pollo) und dem direkten Dateizugriff möglich. Es wird ein wachsender Teil von PDDL-Ausdrücken unterstützt, wobei das Lern-Modul mit einfachen temporalen Aktionen und in begrenztem Umfang mit ADL-Konstrukten arbeitet; die Vorverarbeitung, Planer und Symmetrieerkennung allerdings mit der kompletten PDDL-Beschreibungsmächtigkeit arbeiten können.

Abbildung 1.1 gibt eine grobe Übersicht von der Architektur des Systems. Es wurden die Bereiche gekennzeichnet, in denen der Benutzer mit dem System interagieren kann. Domänen können entweder vom Lern-Modul inferiert oder von der Festplatte geladen werden. Der Benutzer kann verschiedene Optionen der Vorverarbeitung und der Aktionsinstantiierung wählen und die Planer mit den Original- oder transformierten Dateien starten. Als *Backend* wurde die Visualisierung temporaler Pläne und Animation von Planabläufen integriert. Als zusätzliche Option stehen zur Optimierung die Zielanordnung und Symmetrieerkennung

[1] ADL (*abstract description language*), präpositionales Planen mit negierten Vorbedingungen, konditionalen Effekten und quantifizierten Ausdrücken

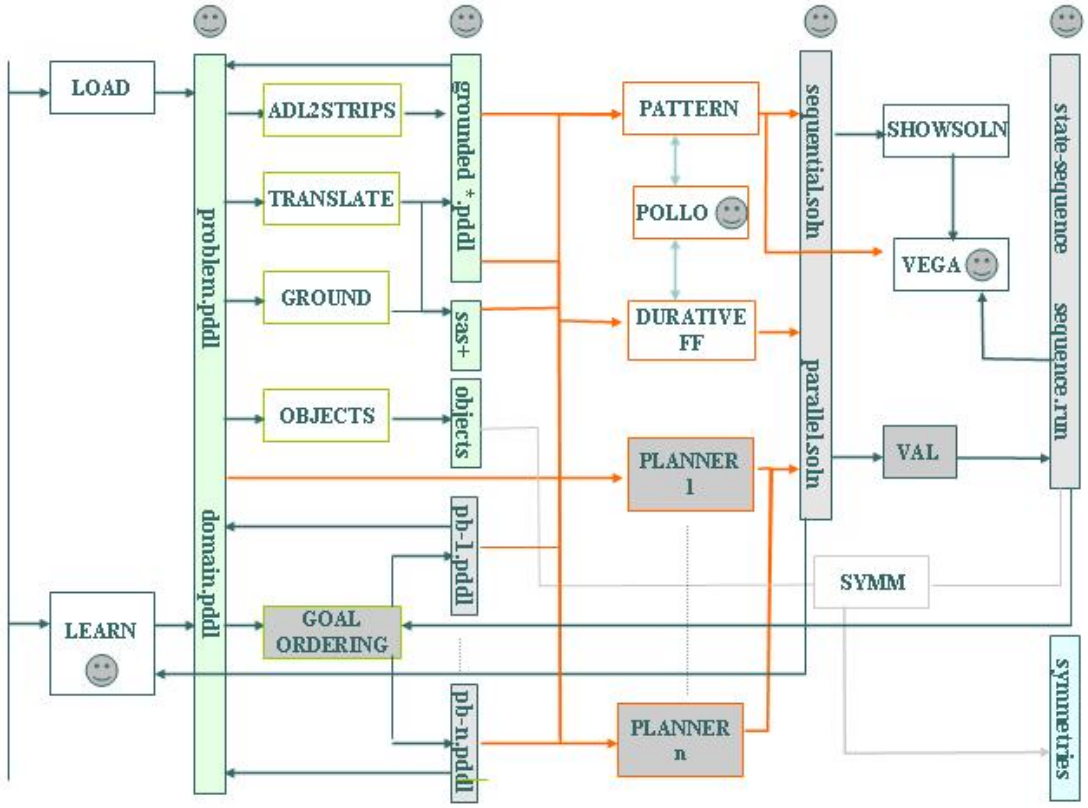


Abbildung 1.1.: Struktur der ModPlan-Werkbank

für generierte Pläne zur Verfügung. Die zwei neuen Planer, die mit eingebunden wurden sind: *DurativeFF*, ein auf heuristischer Suche basierender Planer für die gesamte PDDL-Beschreibungsmächtigkeit und *PatternPlan*, ein optimaler, propositionaler Planer, der mit Musterdatenbanken (*Pattern Databases*) arbeitet.

1.3. Domäneninstantiierung und Codierung

PDDL [14] ist eine ausdrucksstarke Sprachhierarchie für Domänenbeschreibungen. Stufe 1 umfasst typisierte Domänenbeschreibungen und ADL Expressivität. Auf Stufe 2 können gemischt propositionale und nummerische Probleme eingeführt und optimiert werden. In Stufe 3 wird zeitliches Planen eingeführt: einer Aktion kann eine Ausführungsdauer zugeordnet werden. Aktionsgenerierung wie in der Planung wird mit Aktionsanordung wie beim Scheduling kombiniert. Neueste Erweiterungen von PDDL [11] beinhalten Domänen-Axiome in Form von abgeleiteten Prädikaten (derived predicates) und zeitliche Literale (timed initial literals), die (wenn auch eingeschränkt) die Modellierung exogener Ereignisse ermöglichen.

Üblicherweise besteht die PDDL-Eingabe für das Planungsverfahren aus zwei unterschiedlichen ASCII-Dateien: dem problemunabhängigen *domain file* und dem instanzspezifischem *problem file*. In der Domänenbeschreibung werden parametrisierte Prädikate und Funktionen, sowie Operatoren mit Vorbedingungen und Effekt-Liste deklariert. Die Problembeschreibung beinhaltet die Definition der Objekte und ihrer Typen, den Startzustand, die Definition des Planungsziels und die Zielfunktion zur Optimierung.

Grounding ist der Prozess des Findens von Obermengen von erreichbaren Aktionen, Fakten und Fluents durch Instantiierung der Operatoren, Prädikate und Funktionen mit den Objekten, die in der Problembeschreibung spezifiziert wurden. Die meisten gegenwärtigen Planer führen irgendeine Form des Grounden durch, um die Zustandsraumexploration zu verwenden. In der Werkbank haben wir drei Techniken integriert: *translate*, der Instanziierungs-Prozess des Fast-Downward Planners [19], *adl2strips*, die Domänen-Translation des FF Planers [25] und *ground*, die Vorverarbeitung des Mips Planers [6]. Während *translate* und *adl2strips* dazu in der Lage sind, die ziemlich komplizierten ADL-Ausdrücke des propositionalen Planens zu behandeln, kann *ground* mit metrischen und temporalen Domänen umgehen. Da *adl2strips* die Syntax der PDDL-Stufe 1 anbietet, erzeugten wir ein Programm um die Syntax der PDDL-Stufen 1-3 zu generieren.

Eine rein propositionale Codierung kann Effizienzeinbußen während der Exploration hervorrufen. Eine multivariate Darstellung der Atome ist vorzuziehen. Bei der SAS$^+$ Codierung [19] werden Gruppen mit sich gegenseitig ausschließenden Atomen erstellt. Diese Codierung dient als optionale Eingabe für vorhandene Planer, die diese Möglichkeit nutzen können. Für die Werkbank wählten wir die multivariate Codierung von *translate* und *ground*. Das Format der Ausgabedatei für diese Domänen-Analyse ist eine Lisp-ähnliche Darstellung der Menge der erreichbaren Atome und ihrer Einteilung in die Domänen der SAS$^+$-Variablen.

1.4. Symmetrierkennung

Sofern nicht richtig behandelt, verursachen nicht erkannte Symmetrien eine Explosion des Planungssuchraums. Symmetrien vollautomatisch zu ermitteln ist nicht einfach, da dieses Problem mit dem rechnerisch harten Problem der Graphisomorphie verwandt ist. Man beachte, dass das generelle Problem nicht voll klassifiziert ist. Es wird angenommen, dass es nicht NP-vollständig ist [40]. Einige Einblicke in die Resultate aus der Komplexitätstheorie sind: Wenn GI NP-vollständig ist, dann ist $\Sigma_2 = \Pi_2$; $\overline{GI}$ hat ein interaktives Beweissystem mit zwei Kommunikationsrunden, und GI hat ein interaktives Beweissystem mit perfekter *Zero knowledge* Eigenschaft.

Um einen Zustandsraum unter Berücksichtigung einer Zustandssymmetrie zu untersuchen, muss die Explorationsmaschine zusätzlich ein repräsentatives Element für jede Äquivalenzklasse bestimmen. In den meisten Annäherungen [38, 16] werden die Symmetrien komplett vom Nutzer angegeben. Einige Planer verwenden automatische *Objekttranspositionen* [13]. Zwei Objekte sind symmetrisch, wenn sie im momentanen Zustand ausgetauscht werden können, ohne dass die Lösbarkeit oder die Optimalität im restlichen Planungsproblems beeinflusst werden. Solche Symmetrien kommen häufig in Benchmark-Problemen vor. Ein zusätzliches Problem bei Objekttranspositionen ist, dass Symmetrien während der Exploration verschwinden und wieder auftauchen können.

Die wichtigste Beobachtung ist, dass Objekttranspositionen für die Domäne transparent sind, so dass die Symmetrieerkennung allein unter Berücksichtigung des aktuellen Zustands und der Zielspezifikationen möglich ist. Für n Objekte haben wir $n!$ mögliche Objektpermutationen. Berücksichtigt man alle Typinformationen, reduziert sich die Anzahl aller möglichen Permutationen auf $n!/(t_1!\cdot\ldots\cdot t_k!)$, wobei t_i die Anzahl der Objekte vom Typ i, $i \in \{1,\ldots,k\}$ ist. Um die Anzahl potentieller Symmetrien auf eine handhabbare Größe zu reduzieren, nutzt man Objekttranspositionen, für die wir höchstens $n(n-1)/2$ Kandidaten haben.

Unter Einbeziehung von Typinformationen kann sich die Anzahl weiter auf $\sum_{i=1}^{k} t_i(t_i-1)/2$ reduzieren. Für forwärts suchende Planer können wir diese Menge möglicher Objektsymmetrien auf diejenigen beschränken, die auch im Ziel gültig sind. Symmetrieerkennung in unserer Werkbank basiert auf Planschnappschüssen. Das System leitet eine Folge von Objektsymmetrien her, während der Nutzer komplexe Symmetrien, die nicht gefolgert werden können, hinzufügen kann.

1.5. Abhängigkeit von Operatoren und parallele Pläne

In der Planung existieren zwei wesentliche Optimierungs-Metriken. Die Planlänge (Zahl der Operatoren) und der Makespan (minimale parallele Ausführungszeit). Für propositionale Planung bekommt jeder Operator die Ausführungszeit 1, so dass das Makespan der minimalen parallelen Planlänge entspricht. Um parallele Pläne zu berechnen, muss eine Abhängigkeitsrelation der Operatoren zur Verfügung gestellt werden. Dafür müssen zusätzlich Konflikte zwischen nummerische Variablen betrachtet werden.
Zwei Operatoren sind abhängig, falls einer der folgenden Konflikte vorliegt:

1. Die propositionalen Vorbedingungen eines Operators haben einen nicht-leeren Schnitt mit den Effekten des anderen Operators.

2. Der Kopf eines nummerischen Zuweisungsoperators einer Aktion ist enthalten in einer Bedingung der anderen Aktion.

3. Der Kopf einer nummerischen Zuweisung in einer Aktion taucht innerhalb des Restes einer Zuweisung in einer anderen Aktion auf.

Für temporale Planung mit *at start, over all,* und *at end* Modalitäten, haben wir acht unterschiedliche Abhängigkeiten, d.h. start/start, start/over, start/end, over/start, over/end, end/start, end/over und end/end. Wenn es mehr als ein Konflikt für ein Operatorpaar gibt, müssen wir den Maximalwert berechnen, der vom den einzelnen Konflikten abgeleitet wird. In der PDDL 2.1 Semantik zum temporalen Planen erlauben Operatoren Überlappung und erfordern einen minimalen Zeitverzug ε zwischen zwei abhängigen Teilaktionen. Der Wert für ε ist 0,01. Die Idee liegt darin, dass, wenn eine Proposition oder einen nummerischen Ausdruck durch unterschiedliche Aktionen verwendet wird, einige Zeit hierfür berücksichtigt werden muss.
Die Optimierung von Plänen ohne Vorgängerrelation wird miteinbezogen [2], denn die Berechnung des Makespan für ein Folge von Operatoren ist NP-vollständig. Für eine Reihenfolge der Operatoren in einem Plan, wird unter ihnen, ein optimaler paralleler Plan in Polynomialzeit berechnet, der die Bedingungen der Ausführungszeiten respektiert. Mit der

kritischen Pfadanalyse (PERT) kann solch Plan in optimalen Zeit [6] berechnet werden. Der Ansatz lässt sich zu zeitlichen Literalen und Aktionen mit Ausführungszeitfenster erweiterten. Die Operatorabhängigkeit bedingt eine Vorgängerrelation und eine Ausführung der Aktionen in einer bestimmten Reihenfolge. Um ein optimalen Anordnungen (Schedules) von sequentiellen Plänen in der temporal Planung zu berechnen, werden die Abhängigkeiten der Operatoren paarweise vorberechnet, und, nach jeder Berechnung eines Plans, dem Benutzer zur Verfügung gestellt. Er kann diese Abhängigkeiten manipulieren, d.h. hinzufügen, oder löschen, bevor temporale Planordnungen (Schedules) berechnet werden.

1.6. Domänen inferieren

PDDL-Domänen aus partiellen Plänen ohne jegliche Hilfe eines Experten zu erschließen ist eine herausfordernde Aufgabe. Um Operatoren einer Domänenbeschreibung zu inferieren, benötigt der Mechanismus Benutzereingaben. Wir haben einen Lern-Algorithmus implementiert, der beaufsichtigt durch einen Benutzer interaktiv Domänenbeschreibungen erstellt.

Wir gehen davon aus, dass der Benutzer versucht eine korrekte Domäne aus einer Sequenz eines gültigen Planes zu erstellen. Zusätzlich zum Plan werden Prädikate benötigt. Sind die Operatoren gegeben, wird der Benutzer anhand von Auswahlmöglichkeiten aufgefordert Vorbedingungen und Effekte eines Operators zu wählen. Der gesteuerte PDDL Lernmechanismus wählt die folgenden Operatoren und reduziert weitestgehend die Optionen und generiert selbständig Informationen, bis die Domänenbeschreibung vollständig ist. Der Algorithmus orientiert sich am zugrunde liegenden *OpMaker*-Algorithmus [34]. Ein entscheidendes Merkmal ist die Möglichkeit temporale Aktionen zu erstellen und durch wiederholte Eingabe von Plänen inkrementell zu lernen.

1.7. Zufriedenstellende und optimale Planung

Da die Zwischenresultate, die durch Instantiierungsprozeduren erzeugt werden, valide PDDL-Dateien sind, kann jeder Planer als Problemlöser verwendet werden. Um die Entwicklung der Planungstechnologie voranzutreiben und die Anwendbarkeit unseres Knowledge-Engineering-Werkzeugs hervorzuheben, haben wir zwei weitere Planer hinzugefügt: Einen optimalen Planer und ein suboptimales, metrisches und temporales Lösungsmodul. In unsere Werkbank haben wir für beide Planer eine Schnittstelle implementiert.

Aktuelle STRIPS-Planer weisen unterschiedliche Strukturen auf. Während die meisten suboptimalen Planer heuristische und/oder lokale Suche [4] verwenden, reichen optimale Planer von Erfüllbarkeits- [29] und Graphplan-Ansätzen [3], bis hin zu Integer Programmierung [30] und heuristischer Suche [18]. Mit Pattern-Plan steuern wir einen optimalen Musterdatenbank-Planer bei.

In [5] ist die automatisierte Auswahl der möglichen Abstraktionsfunktionen, die zu informativen Pattern Databases führen sollen, ein schweres kombinatorischen Problem. Das gilt insbesondere für die Erzeugung von disjunkten Datenbanken [32], bei denen jeder Operator nur in einer Abstraktion Veränderungen bewirkt. Beim Planen sind Pattern Database-Abstraktionen am effektivsten, wenn sie SAS^+-Gruppen benutzen. Es gibt verschiedene Bin-Packing-Approximationsalgorithmen [5], die zu einer Aufteilung von verschiedenen Gruppen

führen, bevor die Pattern Databases aufgebaut werden. Die maximale Größe einer Pattern Database ist beschränkt durch die Mutiplikation der Kardinalitäten der ausgewählten verschiedenen Domänen. Wir haben einem genetischen Algorithmus implementiert, um die erste Aufteilung der Gruppen zu verbessern. Des Weiteren ermöglichen wir dem Benutzer die approximierte disjunkte Einteilungen in Musterdatenbanken zu verfeinern, was für das Inferenzmodul hilfreich sein kann. Diese Interaktionsmöglichkeiten stehen durch einen XML-Editor zu Verfügung.

Was metrische Planung betrifft, haben wir den Pert Scheduling-Ansatz [6] implementiert und Metric-FF [22] hinzugefügt, um parallele Pläne zu erzeugen. Wir verwenden die kritische Pfadanalyse für komplette Pläne sowie für die partielle und relaxierte Pläne. Während temporale Planung dazu führt, Pläne mit Zeitstempeln anstatt eine Planfolge zu berechnen, haben wir erfolgreich Metric-FF zu Durative-FF erweitert, einen neuen Planer, der PDDL2.1 Level 3 parst und PERT auf den erzeugten sequentiellen Plan anwendet. Erste Ergebnisse auf den Planungwettbewerb-Benchmarks sind vielversprechend. Da der Ansatz zu zeitlicher Literalen in der Form von Ausführungszeitfenster [7] erweitert wurde, wurden diese mit einzubezogen. Durch die Wahl des zugrundeliegenden Planers ist jedoch die Erweiterung der relaxierten Planungsheuristik zu den nicht linearen Problemen [8] offen.

1.8. Validierung und Visualisierung von Plänen

Zur Validierung von Plänen wurde VAL [27] integriert, das von der Strathclyde Planning Group bereitgestellt wird. Die wichtigste Option von VAL ist die Möglichkeit Simulationen zu erstellen, d.h. nahezu jeden Plan in PDDL-Syntax auszuführen. Das Programm wurde erweitert um kontinuierliche Effekte, Ereignisse und Prozesse zu extrahieren.

Um die Pläne zu visualisieren und somit einen detaillierten Blick auf Ausschnitte des Plans zu erlangen, wurde das Animationssystem VEGA [21] in die Werkbank eingebaut. VEGA ist in Client-Server-Architektur konzipiert, wobei der Algorithmus serverseitig und die Visualisierung auf dem Client in einem *Java-Frontend* läuft. Die Animation erweitert den existierenden Code, um mit einfachen Befehlen wie *send point(x,y)* die Visualisierung zu ermöglichen.

In dem System können Objekte zur Visualisierung mit hierarchisch strukturierten Identifikatoren assoziiert werden. Der Client wird als *Frontend* zur Benutzerinteraktion und zur Visualisierung genutzt. Dadurch können Server und Algorithmus gewählt werden, Daten eingegeben, Algorithmen ausgeführt und Visualisierungen angepasst werden. Szenenbeschreibungen können manipuliert werden, die Liste der Algorithmen auf dem Server und die dazu gehörigen Informationen können abgerufen werden, der Algorithmus kann über eine Bedienoberfläche im Stil eines Mediaplayers gesteuert werden und Attribute können direkt oder über den Objektbrowser angepasst werden. Es können mehrere Algorithmen gleichzeitig in verschiedenen Szenen angezeigt werden und verschiedene Sichtweisen einer Szene geöffnet, diese wiederum einzeln geladen bzw gespeichert werden, Durchläufe und Attributlisten vervollständigt werden und Szenen im *xfig*- oder *gif*-Format exportiert werden.

Vega ermöglicht *online* und *offline* Präsentationen. Die Hauptaufgabe des Servers ist die Bereitstellung der Algorithmen per TCP/IP. Der Server kann mehrere Clientanfragen gleichzeitig bearbeiten. Er stellt dem Client eine Liste von Algorithmen zur Verfügung über die der

Client Informationen abrufen, Eingabedaten spezifizieren, den Algorithmus starten kann und die Ausgabedateien erhält. Die Algorithmenliste kann ohne Neustart des Servers modifiziert werden.

Gantt charts sind grafische Darstellungen von temporären Plänen, in denen in horizontaler Form durch verschieden große Rechtecke die Dauer jeder Aktion symbolisieren. Um z.B. auf genauere Informationen eines Operators zuzugreifen können die entsprechenden Rechtecke per Mouse ausgewählt werden. Unser Visualisierungsmodul stellt u.a. *Gantt charts* von Plänen dar.

Domänenspezifische Visualisierung ist eine größere Herausforderung. Basierend auf der Verknüpfung von sequentiellen Operatorabfolgen mit den zugehörigen Zustandssequenzen, wurde eine instanzunabhängige Visualisierung für viele Domänen aus den Wettbewerben entwickelt. Propositionale Atome werden durch Bilder gekennzeichnet die zustandsabhängig gezeigt/nicht gezeigt werden. Durch skalierbare Grafiken werden numerische Größen wie Füllmengen realisiert. Bisher wurden vom Visualisierungsprogramm in Kombination mit speziellen Planern (sequentielle) Pläne mit zugehörigen Statusinformationen erstellt. In der Werkbank wird zusätzlich die Möglichkeit angewandt, parallele und temporäre Pläne zu animieren. Hierfür wurde der Validator erweitert, um eine Zustandssequenz bei jedem Ereignis mit zugehörigem Zeitstempel zu speichern.

Kapitel 2.

Lernalgorithmus zum Erstellen von Problemdomänen

Rachid Karmouni, Arne Wiggers

2.1. Einleitung

Der in der Workbench integrierte Lernalgorithmus dient der vereinfachten Entwicklung von Problemdomänen und -instanzen für Planungsprogramme. Er soll Anwendern ohne besondere Vorkenntnisse der Problembeschreibungssprache PDDL die Möglichkeit geben, einfache Domänen zu entwerfen. Erfahrenen Anwendern dient das Programm als Unterstützung und Vereinfachung des Entwicklungsprozesses. Es wurde ein Programm entwickelt, das auf Basis minimaler Informationen und Eingaben Strukturen der Problembeschreibung erkennt und weitestgehend selbstständig generiert.

2.2. OpMaker

Als Vorbereitung wurde die Arbeitsweise des OpMaker-Algorithmus [34] analysiert. Dieser in den graphischen Domäneneditor GIPO eingebundene Algorithmus induziert aus wenigen Informationen (Eingabesequenz, Benutzerabfrage) eine vollständige Problembeschreibung. Als Eingabe ist ein partielles Domänenmodell in OCL (*Object Constraint Language*[1]) gefordert, welches Beschreibungen der Objekte, Typen der Objekte, Prädikate, Klassifizierung der Prädikate und Invarianten benötigt.

Aufgrund dieser Initialinformationen wird eine Beispielsequenz von Aktionen bearbeitet, die einen korrekten Teilabschnitt einer Problemlösung darstellt. Alle für eine Problembeschreibung nötigen Informationen werden dadurch und durch Benutzerabfragen zu einzelnen Aktionen generiert.

Der OpMaker-Algorithmus In Abbildung 2.1 wird der Ablauf des OpMaker-Algorithmus dargestellt: Für jeden Operator der Eingabesequenz werden Name und Parameter bestimmt.

Die Effekte (RHS: *right hand state*) werden vom Benutzer aus den möglichen Prädikatklassen (*substate-classes*: Zusammenfassung aller Prädikate, deren erster Parameter als Ordnung gilt) gewählt. Aus diesen und den Effekten der vorangegangen Aktion induziert der Algorithmus die Vorbedingungen des Operators. Im Falle konditionaler Aktionen müssen Effekte und Vorbedingungen (LHS: *left hand state*) komplett vom Benutzer gewählt werden.

2.3. Lern-Algorithmus

Im Folgenden wird der grundsätzliche Ablauf zum Erstellen einer Domäne, also die Funktionsweise des Algorithmus besprochen. Anhand eines Beispiels wird später schrittweise durch das Programm geführt.

Bei der Entwicklung des Algorithmus wurde zum Parsen der Eingabe der StringTokenizer von Arash Partow[2] genutzt.

[1] http://www.klasse.nl/ocl
[2] http://www.partow.net/programming/stringtokenizer/

```
program opmaker(OS: training sequence)
In: partial domain model
Out: parameterised operator description
1.  for each op in OS do
2.   form name and parameter list P;
3.   for each dynamic O of sort S in P do
4.     get RHS from user input
5.     induce nessery substate class LHS
6.     form transition T = (S, O, LHS ? RHS)
7.     match free vars in T with those in P
8.   end for
9.   for all conditional transitions
10.        get LHS from user input
11.        get RHS from user input
12.        form ‘ ? O ?  S, (S, O, LHS ? RHS)‘
13.      end for
14.  end for
procedure match free vars in T with those in P
1.  repeat
2.   for each parameter X in transition T, X ? O
3.     choose a parameter Y in P to match with
4.     X such that Y ? O, sort(X) = sort(Y),
5.   end for
6.  until parameter match set is consistent
7.  end
```

Abbildung 2.1.: Pseudocode des OpMaker-Algorithmus

2.3.1. Eingabedaten

Bei der Eingabe der Initialinformationen kann das vorhandene Template mit entsprechenden Daten gefüllt werden. Hier können der gewünschte Domänenname, Prädikate und Aktionssequenz eingegeben werden. Die Prädikate sind uninitialisiert mit Typen anzugeben. In der Aktionssequenz werden alle benötigten Aktionen für die spätere Domäne eingegeben, so dass sie einen sinnvollen Ablauf des Problems darstellen. Diese intuitive Sequenz kann beliebig lang sein und mit beliebig vielen Objekten beschrieben werden.

Nach abgeschlossener Eingabe werden die Daten geparst und in den zur Weiterverarbeitung benötigten Vektoren gespeichert. Der Benutzer wird aufgefordert, die vom Algorithmus aus der Aktionssequenz extrahierten Objekte mit den Typen aus der Prädikateingabe abzugleichen. Bei einfachen Problemen wie zum Beispiel Blocksworld mit nur einem Objekttyp wird dieser automatisch vom Programm zugewiesen. Um den Eingabeaufwand weiterhin einzuschränken, gruppiert der Algorithmus die als zusammengehörig erkannte Objekte bei der Abfrage.
Der dabei verwendete Algorithmus läuft nach folgendem Prinzip ab:

```
IF Aktion.name(i) == Aktion.name(j)
   Parameter(i).index=0
   Parameter(j).index=0
   IF Parameter(i) != Parameter(j)
   Typgruppe(i) = Typgruppe(j)
   Parameter(i).index++
   Parameter(j).index++
```

Aus diesen Informationen generiert das Programm alle möglichen Prädikatinstanzen, die in der späteren Effekt- und Vorbedingungeingabe benutzt werden.

2.3.2. Auswahl der Effekte

Die Zuordnung der Effekte orientiert sich am o.g. OpMaker-Algorithmus. Der grobe Ablauf lässt sich am Aktionsdiagramm in Abbildung 2.2 verdeutlichen.

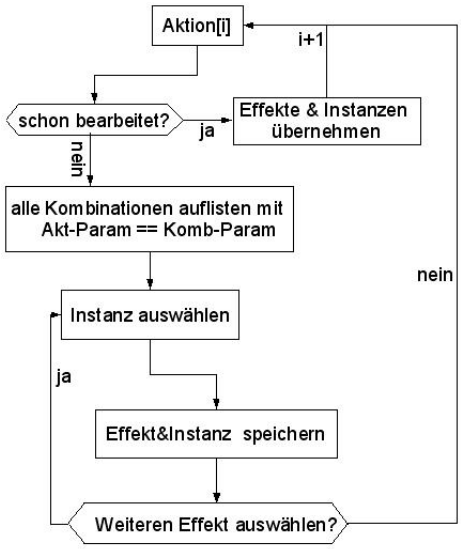


Abbildung 2.2.: Aktionsdiagramm der Effektauswahl

Zum weiteren Verständnis wird hier noch einmal der Algorithmus als Pseudocode aufgeführt:

```
Aktion a
Prädikatinstanz p
   IF Anzahl.Parameter(a)<=Anzahl.Parameter(p)
   FOR ALL Parameter(p)
    IF Parameter(p) == Parameter(a)
    Auswahlmöglichkeiten += p
```

Jeder Aktion aus der Aktionssequenz werden vom Benutzer die gewünschten Effekte aus den Prädikatinstanzen zugewiesen. Um die Auswahl der Effekte einzuschränken, können nur relevante Prädikatinstanzen gewählt werden. Diese Einschränkung erfolgt anhand eines

Abgleichs der übereinstimmenden Parameter und der Anzahl der Parameter, dass heisst in der Prädikatinstanz können höchstens so viele und nur die Parameter vorkommen, die auch in der Aktion stehen. Ist die Eingabe für die Aktion abgeschlossen, geht der Algorithmus zur nächsten über. In jedem Schritt werden die bereits bearbeiteten Aktionen mit der aktuellen verglichen. Wurde bereits eine Aktion mit gleichem Namen bearbeitet, werden die bereits gewählten Effekte übernommen und die entsprechenden Effektinstanzen vom Programm generiert. Jede Aktion muss also nur einmal abgefragt werden. Bei der Effektauswahl hat der Benutzer zusätzlich die Möglichkeit, die Aktion mit einem temporalen Attribut oder einer einfachen Bedingung zu erstellen. Bei temporalen Aktionen muss die Dauer und für jeden Effekt den der Benutzer wählen möchte, das entsprechende Präfix (*at start, at end, over all*) angegeben werden. Für konditionale Effekte (*for all*) kann der Typ gewählt werden, für welchen dieser Effekt gelten soll. Des Weiteren kann zu jeder Auswahl das *not*-Präfix gewählt werden, um diesen Effekt als Negation zu übernehmen. Diese Prädikate werden automatisch ohne das Präfix als Vorbedingung der Aktion übernommen.

2.3.3. Auswahl der Vorbedingungen

Die Auswahl der Vorbedingungen verläuft in drei Phasen:

1. Suche nach Prädikaten, die von einem bestimmten Objekt abhängig sind.
2. Suche nach allen möglichen Kombinationen, die aus den gefundenen Prädikaten resultieren.
3. Aus den Kombinationen werden die Bedingungen generiert.

Für jeden Schritt aus der Aktionssequenz wird folgendes gemacht: Ähnlich zur Auswahl der Effekte werden nur die Prädikate gesucht, deren erster Parameter mit einem Parameter der Aktion übereinstimmt (Phase 1). Diese werden in dem Vektor *curPredicates* gespeichert. Wenn es keine Prädikate in diesem Vektor gibt, wird das nächste Objekt betrachtet. Wenn der Vektor leer ist, werden alle möglichen Kombinationen, die man aus diesen Prädikaten und den Objekten, die in dem aktuellen Schritt vorkommen, konstruiert. Diese werden in einem Vektor (*tempPredicates*) gespeichert (Phase 2). In der dritten Phase wird hauptsächlich der Vektor der Kombinationen betrachtet. Aus diesen Kombinationen wählt der Algorithmus passende Bedingungen direkt aus, unpassende Bedingungen werden nach einer im Folgenden erläuterten Heuristik gelöscht. Es werden alle Kombinationen durchsucht. Für jede Kombination werden zwei Abfragen gemacht:

- Gehört die Bedingung zu den Effekten dieser Aktion? Wenn ja, wird diese Kombination gelöscht und wird nicht mehr betrachtet.
- Gehört diese Bedingung zu den Effekten der Aktion des vorherigen Schrittes? Wenn ja, wird diese Kombination als Bedingung direkt übernommen und aus dem Vektor*tempPredicates* gelöscht.

Wenn am Ende noch Kombinationen vorhanden sind, wird der Benutzer gefragt, ob und welche von diesen noch als Bedingung übernommen werden sollen. Diese Schritte werden beim ersten Bearbeiten der Aktion durchlaufen. Wenn die Aktion schon vom Algorithmus

bearbeitet wurde, folgt das Programm einem anderen Verfahren (siehe auch Flussdiagramm Abbildung 2.3): Es werden noch einmal die Phasen 1 und 2 durchgearbeitet. Die Bedingungen

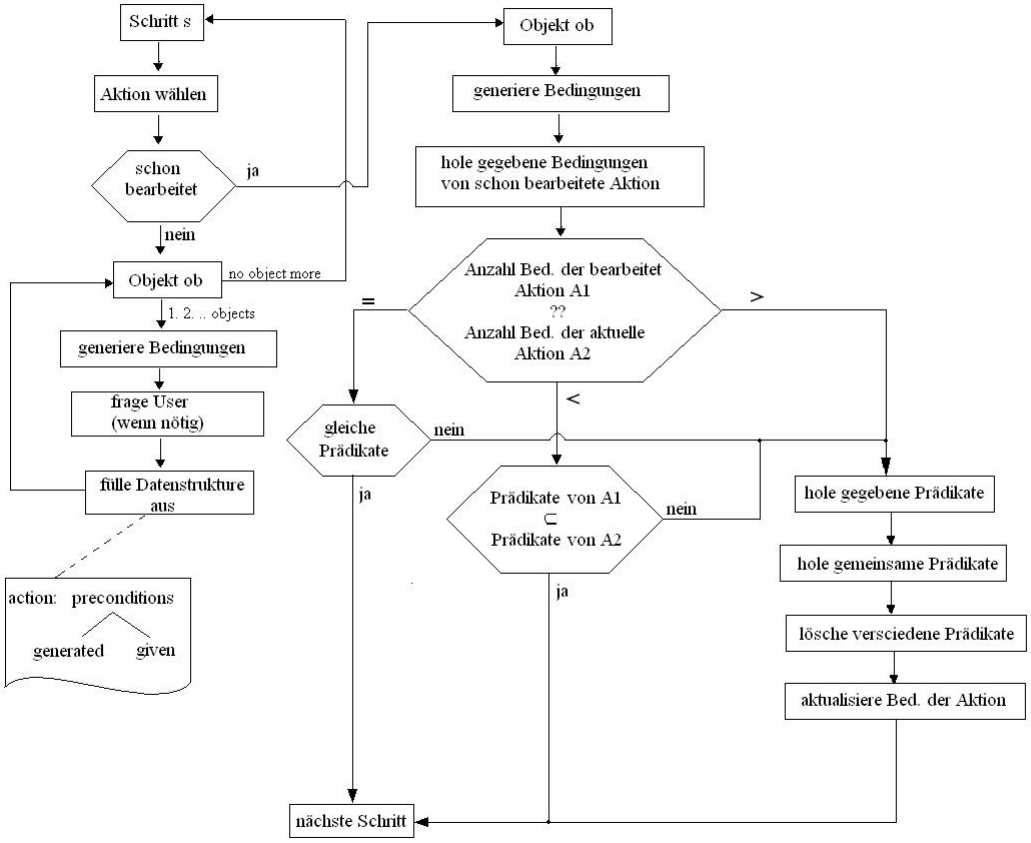


Abbildung 2.3.: Aktionsdiagramm zur Auswahl der Vorbedingungen

werden automatisch generiert. Der Benutzer wird also nicht mehr gefragt; die vom Benutzer gewählten Bedingungen aus dem ersten Aufruf der Aktion werden direkt übernommen.
Die Anzahl der generierten Bedingungen wird mit der Anzahl der Bedingungen des ersten Aufrufs der Aktion verglichen:

1. Sind gleichviele Bedingungen vorhanden, wird noch verglichen, ob es sich hierbei auch um die gleichen handelt.

   a) wenn ja, war das Generieren erfolgreich und es wird die nächste Aktion betrachtet.
   b) wenn nicht, werden zuerst die (vom Benutzer) gegebenen Bedingungen, dann die übereinstimmenden Bedingungen gespeichert.

   Die übrigen Bedingungen (die wir als überflüssig bezeichnen) werden gelöscht und die Aktion (beziehungsweise Bedingung der Aktion) wird aktualisiert. Es kann die nächste Aktion betrachtet werden.

2. Wenn die Anzahl der generierten Bedingungen größer als die Anzahl der Bedingungen des ersten Aufrufs der Aktion ist, wird geprüft, ob die kleinere in der größeren Menge enthalten ist:

   a) wenn ja, wird die nächste Aktion betrachtet.

   b) wenn nicht, wird nochmal, wie in 1(b) beschrieben, vorgegangen.

3. Wenn die Anzahl der generierten Bedingungen kleiner als die Anzahl der Bedingungen des ersten Aufrufs der Aktion ist, wird geprüft, ob die kleinere Menge in der größeren enthalten ist. Dann wird das beschriebene Teilverfahren aus 1(b) durchgeführt.

Durch den Vergleich mit den gewählten Effekten, das Einbeziehen der bereits erstellten Vorbedingungen durch die not-Effekte und im Falles des wiederholten Auftretens durch die Heuristik zum Abgleich mit bearbeiteten Aktionen wird auch hier der Eingabeaufwand minimiert. Wurde in der Effekteingabe eine temporale Aktion erstellt, muss zu dieser Aktion auch bei der Eingabe der Vorbedingungen der entsprechende Präfix gewählt werden. Wurden vom Algorithmus Bedingungen generiert, wird der Benutzer über eine gesonderte Abfrage aufgefordert diese Eingabe zu tätigen. Die Angabe der Dauer ist fix und kann nicht mehr geändert werden.

2.4. Erstellung der Problem- und Domänendatei

Das Auswerten der gesammelten Daten und Umwandeln in PDDL-Notation erfolgt durch Einfügen der Strings aus den Datenstrukturen (Aktionsname, Parameter, etc.) an entsprechenden Stellen in syntaktisch korrekter Umgebung. Hierbei wird unterschieden zwischen der Strips-Erstellung und der Domänenerstellung mit einfachen temporalen Attributen. Jede Aktion wird über Schleifen abgearbeitet, in deren Verlauf die Vorbedingungen, Effekte und andere Informationen (zum Beispiel Duration) eingefügt werden. Die Objektnamen der Eingabesequenz werden für die Problembeschreibung in *neutrale*, PDDL-typische Bezeichner umgewandelt.

Zum Erstellen der Probleminstanz wird die Struktur der Aktionssequenz betrachtet. Für den Initialzustand der Objekte werden diese in den Vorbedingungen der jeweiligen Aktionen gesucht. Gesucht wird die erste Stelle, an der das Objekt in einer Vorbedingung steht und nicht aus dem Effekt der vorherigen Aktion resultiert. Wird ein solches Prädikat gefunden, wird es in den Initialzustand übernommen. Der Vorgang zum Generieren der Zielzustände erfolgt nach dem gleichen Prinzip. Die Objekte werden hierfür in den Effekten der Aktionen gesucht und aus der letzten Aktion, in der die darauffolgende das Objekt nicht mehr in den Vorbedingungen benötigt, übernommen. Durch dieses Verfahren entsteht eine partielle Probleminstanz, die vom Benutzer erweitert werden kann. Im folgende Pseudocode wird die Probleminstanzerstellung nochmals verdeutlicht.

```
FOR ALL Objekt o
   IF Initialzustand(o) empty
  FOR Aktion a = 0
      IF o == Vorbedingung(a)
       AND FOR ALL s < a
```

```
        o != Effekt(s)
        Initialzustand(o) = Vorbedingung(a)
ELSE a++


 IF Zielzustand(o) empty
FOR ALL Aktion a = n
IF o == Effekt(a)
     AND FOR ALL s > a
        o != Vorbedingung(s)
        Zielzustand(o) = Effekt(a)
ELSE a--
```

2.5. Programmablauf anhand eines Beispiels

Anhand der Eingabesequenz in Abbildung 2.4 werden die einzelnen Schritte nochmals verdeutlicht.
Nachdem die Daten erfolgreich eingelesen und zu einem String konvertiert wurden, werden die einzelnen Zeilen nach Markierungen durchsucht. Da in der Eingabeoption ein Template bereitgestellt wird, werden folgende Markierungen immer gefunden: *Domain* markiert die Zeile, in welcher der Name steht. Nach *Predicates* stehen die Prädikate der Domäne und nach *Actions* die Aktionssequenz. Die zwischen diesen und den Endindikatoren (Sequenzenden werden durch # markiert) gefundenen Eingaben werden in Token aufgeteilt und in den entsprechenden Vektoren zur Weiterverarbeitung gespeichert.

Als Erstes sucht das Programm nun nach übereinstimmenden Objekten in den Aktionen. Das Programm gruppiert die angegebenen Objekte und es muss zu jedem Typ eine Gruppe zugeordnet werden. Hierbei werden zunächst alle Objekte in individuelle Gruppen eingeordnet. Danach werden gleichnamige Aktionen gesucht, deren an gleicher Position stehenden Parameter verglichen werden. Sind die Parameter ungleich wird einem der beiden die Gruppe des anderen zugeordnet. Die erzeugten Gruppen werden noch einmal sortiert, um leere Gruppen zu vermeiden. Im Beispiel wird so zum Beispiel die Typengleichheit von 'tom' und 'jerry' erkannt, denen vom Benutzer der Typ 'person' zugeordnet wird. Sollte das Programm erkennen, das nur eine Objektgruppe existiert (wie zum Beispiel in Blocksworld) wird der einzige Typ automatisch zugewiesen. Nach dieser Zuordnung werden intern Methoden aufgerufen, in denen die Datenstrukturen erstellt werden, welche die Weiterverarbeitung vereinfachen und dem Benutzer intuitive Zuordnungen ermöglichen sollen.

Wird mit der Effektauswahl begonnen, muss der Benutzer im Beispiel eine durative Aktion erstellen, indem die Checkbox aktiviert und eine ganzzahlige Dauer angegeben wird. Zur Aktion 'board tom mig london' wählt der Benutzer als Duration '20' und aus den anhand der Parameter reduzierten Prädikatkombinationen die beiden Effekte 'in tom mig' mit dem durativen Präfix 'at end' und für den Effekt 'at-pers tom london' den Präfix 'at start' und dem Bezeichner 'not'. Das Programm speichert den durativen Wert, die gewählten Instanzen und das Prädikat selbst, welches durch Abgleich der Namen von der gewählten Instanz und den Prädikatnamen erkannt wird. Das Prinzip wiederholt sich bei der nächsten Aktion 'fly

```
Domain: zeno-travel
Predicates:
at-pers person city
at-aircr aircraft city
in person aircraft
fuellevel aircraft flevel
next flevel flevel
#
Actions:
board tom mig london
fly mig london newyork full medium
debark tom mig newyork
refuel mig newyork medium full
board jerry mig newyork
zoom mig newyork berlin full medium low
debark jerry mig berlin
refuel mig berlin low medium
board jerry mig berlin
fly mig berlin newyork medium low
refuel mig newyork low full
board tom mig newyork
zoom mig newyork london full medium low
#

%
```

Abbildung 2.4.: Beispielhafte Eingabesequenz für Zenotravel

mig london newyork full medium' (Ebenso mit 'debark...' und 'refuel...'). Da nun in der Sequenz eine Aktion folgt, die schon bearbeitet wurde ('board jerry mig newyork') werden die bereits gewählten Effekte übernommen. Die Instanzen werden anhand der Positionen der Parameter in Prädikat und Aktion generiert und ebenfalls übernommen.

Nachdem alle nötigen Effekte entweder eingegeben oder generiert wurden, folgt die Bearbeitung der Vorbedingungen. Im ersten Schritt wird wieder die Aktion 'board tom london' bearbeitet. Aus den Effekten wurden bei dieser Aktion schon Vorbedingungen inferiert, denen der Benutzer lediglich noch den durativen Präfix zuweisen muss. Bei nicht-durativen Aktionen ist diese Abfrage natürlich nicht nötig. Aus dem mit 'not' gewählten Effekt wird die Vorbedingung 'at-pers tom london' und zusätzlich vom Algorithmus die Vorbedingung 'at-aircr mig london' generiert. Der Algorithmus erkennt keine weiteren Kombinationen, die zur aktuellen Aktion passen könnten und somit ist das Auswahlfenster leer. Um Konsistenz zu wahren ist es nach initialer Eingabe der Duration in der Effektauswahl nicht mehr möglich den Wert (zum Beispiel bei der Wahl der Vorbedingungen) zu ändern, da sonst u.U. mehrere verschiedene Angaben gemacht werden könnten. Bei der nun folgenden Aktion 'fly mig london newyork full medium' wurden wieder zwei Prädikate generiert, die nach der

```
(define (domain zeno-travel) (:requirements :typing :strips
:durative-actions) (:predicates
    (at-pers ?x1 - person ?x2 - city)
    (at-aircr ?x1 - aircraft ?x2 - city)
    (in ?x1 - person ?x2 - aircraft)
    (fuellevel ?x1 - aircraft ?x2 - flevel)
    (next ?x1 - flevel ?x2 - flevel)
) (:durative-action debark
    :parameters (  ?x1 - person ?x2 - aircraft ?x3 - city)
    :duration (= ?duration 30)
    :condition ( and (at start (in ?x1 ?x2))
            (over all (at-aircr ?x2 ?x3)))
    :effect ( and (at end (at-pers ?x1 ?x3))
            (at start (not (in ?x1 ?x2)))))
(:durative-action fly
    :parameters (  ?x1 - aircraft ?x2 - city ?x3 - city
                ?x4 - flevel ?x5 - flevel)
    :duration (= ?duration 180)
    :condition ( and (at start (at-aircr ?x1 ?x2))
            (at start (fuellevel ?x1 ?x4))
            (at start (next ?x4 ?x5)))
    :effect ( and (at end (at-aircr ?x1 ?x3))
            (at end (fuellevel ?x1 ?x5))
            (at start (not (at-aircr ?x1 ?x2)))
            (at end (not(fuellevel ?x1 ?x4)))))
```

Abbildung 2.5.: Generierte Zenotravel-Domäne (Ausschnitt)

Präfixabfrage übernommen werden. Zusätzlich werden dem Benutzer alle möglichen Prädikatinstanzen, also übereinstimmend mit Anzahl und Typ der Parameter und eingeschränkt durch die bereits generierten Vorbedingungen, angeboten (in diesem Fall 'next full medium' und 'next medium full'), aus denen noch die Vorbedingung 'next full medium' mit Präfix 'at start' zu wählen ist. Das Prinzip wiederholt sich bei den folgenden Aktionen, bis alle nötigen Eingaben abgeschlossen sind. Das Programm erstellt nun die Problembeschreibung und die partielle Probleminstanz aus den gewonnenen Informationen.

Für die Problem.pddl-Datei (siehe auch Abbildung 2.5) durchsucht der Algorithmus für jedes Objekt einen Initialzustand, also einem Zustand der vorher nicht verändert wurde, und einen Zielzustand, der in nachfolgenden Aktionen nicht mehr geändert wird. Auf diese Weise werden im Beispiel die Initialzustände für 'tom' und 'jerry' gefunden ('at-pers tom london' und 'at-pers jerry newyork') und die Abfolge 'next full medium' erkannt. Für den Zielzustand konnte lediglich der Effekt 'in tom mig' erkannt werden (siehe auch Abbildung 2.6) . Diese Beschreibung kann der Benutzer ohne weiteres zu einer vollständigen Probleminstanz ausbauen.

```
(define (problem PROBLEM_NAME) (:domain zeno-travel)
(:objects
tom - person
mig - aircraft
london - city
newyork - city
full - flevel
medium - flevel
jerry - person
berlin - city
low - flevel
)(:init
(at-pers tom london )
(at-pers jerry newyork )
(next full medium)
)
(:goal (and
(in tom mig)
))
)
```

Abbildung 2.6.: Partiell generierte Probleminstanz

2.6. Erweiterungen, Ausblick

Als Erweiterung für den Algorithmus wurde ein Ansatz [41] implementiert, der durch Eingabe von Plänen, Initial- und Zielzuständen Informationen extrahiert und somit die Entwicklung einer Domäne nahezu unabhängig von Anwenderentscheidungen macht. Das Prinzip dieses Ansatzes kehrt die Erstellung der Probleminstanzen um. Der gegebene Plan wird zuerst dekrementell durchlaufen, wobei die Objekte der Zielzustände mit den Parametern der Aktionen verglichen werden. Stimmen die Objekte überein und ist deren Anzahl im Zielzustand höchsten so gross wie in der Aktion, wird der Zielzustand als Effekt der aktuellen Aktion übernommen und aus der Liste der Ziele gelöscht. Daraufhin werden im Plan gleichnamige Aktionen gesucht, für deren Parameter geprüft wird, ob ein entsprechender Zielzustand existiert. Trifft dies zu, wird dieser Zielzustand gelöscht, um zu verhindern, dass er als Effekt für andere Aktionen akzeptiert wird, also insbesondere um zu vermeiden, dass viele Aktionen mit gleichen Effekten auftreten. Nachdem eine Aktion bearbeitet wurde, werden gleichnamige Aktionen, die schon durchlaufen wurden, mit den hinzugekommenen Effekten aktualisiert. Für die Vorbedingungen wiederholt sich der Ablauf inkrementell mit den Initialzuständen.

Der Algorithmus erlaubt mehrere Eingaben von Plänen und Zuständen und generiert so mit wachsender Information 'vollständigere' Domänen. Die Korrektheit des Algorithmus kann nicht gewährleistet werden, da die verwandte Heuristik von der Struktur der eingegebenen Informationen abhängig ist. Geplant ist, die generierten Informationen als Input des bestehenden Algorithmus zu verwenden, wobei der Benutzer hier Fehler korrigieren kann

und eventuell noch nicht gelernte Vorbedingungen und Effekte einzufügen.

2.7. Zusammenfassung

Zusammenfassend lässt sich sagen, dass für Benutzer ohne PDDL-Hintergrundwissen beziehungsweise Kenntnisse der Notation eine korrekte Problembeschreibung und eine partielle Instanz erzeugt werden können, die nun mit Planern getestet werden können. Beim Eingeben der Vorbedingungen und Effekte werden falsche Auswahlmöglichkeiten von vornherein weitestgehend ausgeschlossen. Im Algorithmus werden gelernte Informationen weiter verwendet. Auch für das Erstellen von einfachen temporalen Domänen ist nicht mehr Wissen nötig, als zum Beispiel die Vorstellung zu welchem Zeitpunkt welcher Effekt eintritt und wie lange die Aktion dauern soll. Es ergibt sich ein einfach zu bedienender Domäneneditor, der mit wenig Initialinformationen auskommt, keine fundierten Kenntnisse in Problembeschreibungsformalismen erfordert und schrittweise anhand intuitiver Abfragen durch den Entwicklungsprozess führt.

Kapitel 3.

Vorverarbeitung

Miriam Bützken, Andrea Matuszewski

3.1. Einleitung - Wozu Vorverarbeitung?

Die Vorverarbeitung dient dazu, die spätere Suche nach einem Plan zu vereinfachen. Es wird versucht, das Problem - ohne den Inhalt zu verändern - so umzuformen, dass die Laufzeiten bei der Plansuche reduziert werden und eventuell bessere Pläne gefunden werden können.

3.2. Übersetzung

3.2.1. Aufgabenstellung

Unsere Aufgabe war es, eine vereinfachte Problemeingabe zu erzeugen, die von möglichst vielen Planungssystemen verarbeitet werden kann. Da so gut wie alle Planer PDDL verarbeiten können, sollte das vereinfachte Problem auch in dieser Form vorliegen. Da die Vorverarbeitung des Downward-Planers unserer Meinung nach sehr gut ist, entschieden wir uns dafür, die dort erzeugte Ausgabe *test.sas*, die als Textfile in kodierter Form vorliegt, in eine PDDL-konforme Darstellung zu übersetzen. Dazu sollte ein Programm geschrieben werden, das die Dateien *test.groups* und *test.sas* als Eingaben erhält und dementsprechend eine Problem- und Domänenbeschreibung in PDDL erzeugt und auch wieder in Textfiles speichert, die dann als vereinfachte Eingabe dienen sollen.

3.2.2. Der Downward-Planer

Warum Downward?

Fast Downward ist ein propositionales Planungssystem, das auf heuristischer Suche basiert. Verglichen mit anderen heuristischen Planern wie HSP oder FF hat Downward zwei entscheidende Vorteile. Zum Einen ist der Planer sehr gut für Planungsaufgaben mit nicht binären Zustandsvariablen geeignet, zum Anderen erschließt der Planer kausale Abhängigkeiten zwischen Zustandsvariablen, um relaxierte Planungsprobleme auf hierarchische Art und Weise zu lösen.

Fast Downward basiert auf heuristischer Zustandsraumsuche ähnlich wie HSP oder FF. Dabei wird die *Kausalgraph* (englisch *causal graph CG*) Heuristik verwendet. Während die CG-Heuristik [36] für einfache STRIPS-Domänen eingeführt wurde, ist Fast Downward in der Lage, mit dem kompletten propositionalen und nicht-temporalen Teil von PDDL umzugehen, kann also beliebige ADL-Konstrukte und abgeleitete Prädikate (Axiome) verarbeiten.

Der Downward-Planer besteht aus drei Programmen:

1. Übersetzer (geschrieben in Python)
2. Vorverarbeiter (geschrieben in C++)
3. Suchmaschine (geschrieben in C++)

Um ein Planungsproblem zu lösen, werden die drei Programme nacheinander aufgerufen. Sie kommunizieren mittels Textfiles. Diese klare Trennung ermöglicht eine schnellere Einarbeitung in die einzelnen Teile und deren simultane Weiterentwicklung auch von verschiedenen Personen. Das war ein wichtiges Kriterium, das uns in unserer Vorbereitungsphase dazu bewogen hat, ausgerechnet Teile dieses Planers für unsere Aufgabe zu nutzen.

Relevante Bestandteile

Der für unsere Arbeit relevante Teil des Downward-Planers ist der Übersetzer.
Der Übersetzer hat die folgenden Möglichkeiten:

- Wegkompilierung der (meisten) ADL-Konstrukte
- Instanziierung der Operatoren und Axiome
- Konvertierung der propositionalen Darstellung in eine Repräsentation mit mehrfach bewerteten Zustandsvariablen.

Es ist bekannt, dass einige ADL-Konstrukte einfach wegkompiliert werden können, ohne die Problemdarstellung dadurch erheblich zu vergrößern, bei anderen funktioniert das nicht so einfach. Fast Downward verwendet die folgenden Transformationen, um die Problembeschreibung zu vereinfachen:

- Übersetzung von Implikationen zu Disjunktionen und übersetzung aller Vorbedingungen in Negations-Normalform (NNF).
- Kompilierung von Allquantoren in Bedingungen.
- Übersetzung von Bedingungen in *prenex* Normalform
- Übersetzung von Bedingungen ohne Quantoren in DNF.
- Aufteilung von Operatoren oder Axiomen mit disjunktiven Vorbedingungen in einzelne Operatoren und Aufteilung konditionaler Effekte mit disjunktiven Vorbedingungen in einzelne Effekte.

Nach diesen Transformationen sind alle Vorbedingungen einfache Konjunktionen von Literalen und die Planungsaufgabe wird somit in STRIPS mit Negationen plus konditionale Effekte und Axiome beschrieben.

Für solche Planungsprobleme ist die Instantiierung vergleichbar einfach. Ähnlich wie bei Mips wird versucht, die Instanziierung von Operatoren zu vermeiden, die nie angewendet werden können. Dies passiert, indem zuerst die Menge der Propositionen ermittelt wird, die in einer relaxierten Exploration erreichbar sind. Negative Vorbedingungen und Effekte werden hierbei ignoriert. Der letzte Arbeitsschritt von Translate führt die Ersetzung der Menge der binären Zustandsvariablen durch. Die Ausgabe des Übersetzers sind zwei Textfiles namens *test.sas* und *test.groups*. In *test.groups* werden die Fakt-Gruppen mit den zugehörigen Atomen aufgelistet. In dem File *test.sas* wird die vereinfachte Beschreibung der Planungsaufgabe in kodierter Form gespeichert.

3.2.3. Vorgehensweise

Die erste Frage, die sich uns stellte, war die der zu verwendenden Programmiersprache. Wir entschieden uns für Python, da der Übersetzer ebenfalls in Python geschrieben ist und wir der Meinung waren, dass bei dieser Sprache durch Module viele der für unsere Arbeit hilfreichen Funktionen bereits vordefiniert sind und dass das sicherlich hilfreich sein könnte.

Zunächst arbeiteten wir uns in die für uns neue Programmiersprache Python ein und schauten uns den Quellcode des Übersetzers an. Danach entschlüsselten wir die Kodierung der Textdateien *test.sas* und *test.groups*:

Die Datei *test.groups* beinhaltet - wie oben erwähnt - die Aufteilung der Atome in Fakt-Gruppen. Für jede Fakt-Gruppe wird eine "Nummer" *var0, var1, ...* eingeführt. Die zu einer Gruppe gehörenden Atome werden ebenfalls durchnummeriert:

```
var0:
    0: Atom holding(i)
    1: Atom ontable(i)
    2: Atom on(i, j)
    3: Atom on(i, i)
    4: Atom on(i, h)
    5: Atom on(i, g)
    6: Atom on(i, f)
    7: Atom on(i, e)
    8: Atom on(i, d)
    9: Atom on(i, c)
    10: Atom on(i, b)
    11: Atom on(i, a)
    12: <none of those>
var1:
    0: Atom holding(h)
    1: Atom ontable(h)
    ...
```

Die Datei *test.sas* beinhaltet - wie oben erwähnt - die vereinfachte Problembeschreibung in kodierter Form. Der Aufbau der Datei und ihre Bedeutung ist wie folgt:

```
begin_variables
21                      ⇔       Anzahl der Variablen
var0 13 -1              ⇔       var0:  Variablennummer
                                13:    Anzahl der Atome der Variablen
                                -1:    keine Axiome
                                {0:    Axiom}
var1 13 -1
...
end_variables
begin_state             ⇔       Initialzustand
1                       ⇔       var0:  1  Atom ontable (i)
9                       ⇔       var1:  9  Atom on (h, a)
6                       ⇔       var2:  6  Atom on (j, b)
2                       ⇔       var3:  2  Atom on (d, i)
...
end_state
```

```
begin_goal              ⇔          Zielzustand
9                       ⇔          Anzahl der Atome
1 10                    ⇔          var1:  10
2 5                     ⇔          var2:  5
...
end_goal
200                     ⇔          Anzahl der Aktionen
begin_operator
pick-up f               ⇔          Aktionsbezeichnung
0                       ⇔          Anzahl der Preconditions
3                       ⇔          Anzahl der Effekte
0 18 0 1                ⇔          0:     keine Precondition für Effekte
                                   var18: 0 (alter Wert)
                                          1 (neuer Wert)
0 5 0 1
0 15 0 1
end_operator
...
```

Es können auch Regeln enthalten sein:

```
...
begin_rule
3             ⇔          Anzahl der Regelelemente
242 0         ⇔          var242:  0
85 0          ⇔          var85:   0
175 0         ⇔          var175:  0
48 1 0        ⇔          var48:   1 (alter Wert)
                                   0 (neuer Wert)
end_rule
...
```

Auch Aktionen mit konditionalen Effekten können vorkommen:

```
...
begin_operator
wait                    ⇔          Aktionsbezeichnung
1                       ⇔          Anzahl der Preconditions
103 0                   ⇔          Precondition
2                       ⇔          Anzahl der Effekte
2 104 0 85 0 85 -1 1    ⇔          2:       Anz. Bed. für Effekte
                                   var104:  0
                                   var85:   0
                                   var85:  -1 (alter Wert)
                                            1 (neuer Wert)
2 103 0 87 0 87 -1 1
```

```
end_operator
...
```

Aus diesen beiden Dateien haben wir Listen erzeugt, so dass jedes Element der Liste einer Zeile der zugehörigen Datei entspricht und so einfach darauf zugegriffen werden kann.

Um die Atome in PDDL-Form umzuwandeln, haben wir eine Datei `parse_atom.py` erzeugt, in der wir zuerst die Methode `open_groups()` eingefügt haben, die aus *test.groups* eine Liste macht und diese zurückgibt. Danach schrieben wir die Methode `create_vars()`, die z.B. aus "`0: Atom holding (i)`" in *test.groups* "`(holding-i)`" und aus "`<none of those>`" "`none-of-those`" macht. Da wir mit einem einfachen Blocks-Problem begonnen hatten, reichte diese Umformung zu Beginn noch aus. Bei anderen Problemen aus anderen Domänen kommen jedoch auch andere Konstrukte vor. Daher fügten wir später noch die Hilfsmethoden `remove_komma()`, `remove_n()` und `remove_tab()` hinzu.

Der Methode `remove_komma()` wird eine Zeile aus *test.sas* übergeben, aus der sie die Kommata entfernt, die bei Atomen mit mehreren Parametern diese Parameter voneinander trennen. Der Methode `remove_n()` wird ebenfalls eine Zeile aus *test.sas* übergeben, aus der sie das "`\n`" am Ende dieser Zeile entfernt. Dies dient zur Formatierung der Ausgabedatei. Auch der Methode `remove_tab()` wird eine Zeile aus *test.sas* übergeben. Sie entfernt eventuell vorhandene Leerzeichen vor ")".

Als nächstes haben wir in einer Datei `dict_construction.py` die Methode `vardata()` implementiert. Dieser wird `begin`, `end` und die Datei *test.sas* in Listenform übergeben. Die Werte `begin` und `end` geben den Anfang und das Ende der zur Wörterbuch-Erzeugung relevanten Zeilen aus *test.sas* an (z.B. `var0 13 -1 ...`). Die Zeilen werden aufgesplittet und auch als Listen verwaltet (`[var0, 13, -1][...]...`), die wiederum als Listenelemente einer großen Liste gespeichert werden. Für jede dieser Listen werden nun folgende Aktionen durchgeführt: Die Anzahl der Atome (in der Liste Element 1) werden in einer Variablen gespeichert. Das Element 0 der Liste wird als Wörterbuch (Dictionary) initialisiert. Die Wörterbuch-Einträge werden innerhalb einer Schleife erstellt, indem die Methode `create_vars()` aus `parse_atom.py` aufgerufen wird, der jeweils eine Zeilennummer übergeben wird. Diese Zeilennummer wird unter Berücksichtigung der Anzahl der Atome innerhalb der Schleife erhöht, so dass das Wörterbuch der aktuellen Variable am Ende der Schleife vollständig ist. Das Ergebnis ist also ein Wörterbuch, in dem unter der Nummer des jeweiligen Atoms einer Variablen als Schlüssel der pddl-String als Wert gespeichert ist.

```
[var0 13 -1,          var1 13 -1,          ..., var20 2-1]
     ↓                     ↓                       ↓
[[var0, 13, -1],      [var1, 13, -1],      ..., [var20, 2, -1]]
  ↓                     ↓                       ↓
{0:(holding i)        {0:(holding h)       ..., {0:(clear h)
 1: (ontable i)        1: ... }                    1: none }
 2: (on i j)
 3: (on i i)
 ... }
```

Diese Datenstruktur namens "`liste`" wird von der Methode `vardata()` zurückgegeben

und man kann nun also durch einen Befehl wie `liste[5][0][2]` auf `var5:2` zugreifen. Zurückgegeben wird dann das entsprechende Atom in PDDL-Form. Wir haben diese Datenstruktur erzeugt, um bei der weiteren Bearbeitung schnell und effizient auf die Konstrukte zugreifen zu können. Dies erwies sich im Laufe unserer Arbeit als äußerst hilfreich.

Im Folgenden haben wir zwei neue Dateien `sas_to_pddl.py` und `output_data.py` erzeugt, die wir im Laufe unserer Arbeit für jedes Konstrukt einer PDDL-Beschreibung erweitert haben. Die Datei *output_data.py* ist dazu da, die übersetzten Konstrukte in die entsprechenden Dateien zu schreiben. Sie enthält folgende Methoden:

- `output_groups()`: Es wird die Anzahl der Variablen und das Wörterbuch übergeben. Mit Hilfe dieser Angaben werden die Fakt-Gruppen mit den zugehörigen Variablen in die Textdatei *problem.sas+* geschrieben.
- `output_init()`: Es wird der Initialzustand in Form einer Liste übergeben. Dieser wird formatiert in die Textdatei *problem.pddl* geschrieben.
- `output_goal()`: Es wird der Zielzustand in Form einer Liste übergeben. Dieser wird formatiert in die Datei *problem.pddl* geschrieben.
- `output_pre()`: Es wird die Anzahl der Variablen und das Wörterbuch übergeben. Mit Hilfe dieser Angaben werden die Predicates in die Textdatei *domain.pddl* geschrieben.
- `output_op()`: Es wird der Name der Aktion, die Liste der Preconditions, die Liste der Effects und die Liste der Effect-conditions übergeben. Mit Hilfe dieser Angaben wird die Aktion in die Textdatei *domain.pddl* geschrieben.
- `output_rule()`: Es wird Head und Body der Regel übergeben. Mit Hilfe dieser Angaben wird die Regel in die Textdatei *domain.pddl* geschrieben.
- `end()`: Bildet das Ende der Datei *domain.pddl*, indem ein ")" an das Ende geschrieben wird.

Die Datei `sas_to_pddl.py` ist dazu da, die test.sas-Datei zu übersetzten. Die verschiedenen PDDL-Konstrukte werden jeweils mit einer eigenen Methode übersetzt. Zur Ausgabe der Resultate werden die Methoden aus `output_data.py` aufgerufen. `sas_to_pddl.py` enthält die folgenden Methoden:

- `open_sas()`: Erzeugt aus *test.sas* eine Liste und gibt diese zurück.
- `groups()`: Ruft `output_groups()` aus `output_data.py` auf; dient also zur Erzeugung der Datei *problem.sas+*.
- `init()`: Erzeugt mit Hilfe der Angaben aus *test.sas* und der zugehörigen Wörterbuch-Einträge eine Liste der Atome im Zielzustand, ruft die Methode `output_init()` aus `output_data.py` auf und übergibt dieser die Liste.
- `goal()`: Erzeugt mit Hilfe der Angaben aus *test.sas* und der zugehörigen Wörterbuch-Einträge eine Liste der Atome im Startzustand, ruft die Methode `output_goal()` aus `output_data.py` auf und übergibt dieser die Liste.

- `parse_operators()`: Geht in einer Schleife alle in *test.sas* vorhandenen Aktionen durch, erzeugt mit Hilfe der zugehörigen Wörterbuch-Einträge für jede Aktion eine Liste der Preconditions, eine Liste der Effect-conditions und eine Liste der Effects, ruft die Methode `output_op()` aus `output_data.py` auf und übergibt dieser den Aktionsnamen und die Listen. Die Nummer der Zeile aus *test.sas*, in der die Aktionen enden, wird zurückgegeben, um diese Angabe bei der weiteren Übersetzung nutzen zu können.

- `parse_rules()`: Die Nummer der Zeile, in der die Regeln beginnen, wird übergeben. In einer Schleife werden alle in *test.sas* vorhandenen Regeln durchgegangen. Mit Hilfe der zugehörigen Wörterbuch-Einträge wird für jede Regel eine Liste der Preconditions (entspricht dem Body der Regel) erzeugt. Anschließend wird die Methode `output_rule()` aus *output_data.py* aufgerufen, der der Effekt (entspricht dem Head der Regel) und die Liste der Preconditions übergeben wird. Auch hier wird wieder die Nummer der Zeile aus *test.sas* zurückgegeben, in der die Regeln enden, um diese Angabe bei der weiteren Übersetzung nutzen zu können.

- `main-Methode`: Es werden die Methoden `groups()` und `predicates()` aufgerufen, dann wird die Variable `i` auf den Rückgabewert von `parse_operators()` gesetzt und die Methode `parse_rules(i)` aufgerufen. Zum Schluss werden noch `init()`, `goal()` und `output_data.end()` aufgerufen.

In der Datei selbst wird zu Beginn die Methode `open_sas()` aufgerufen und die zurückgegebene Liste wird für alle späteren Zugriffe in der Variablen `file_sas` gespeichert. In der Variablen `number_of_variables` wird die Anzahl der in dem Problem vorhandenen Variablen hinterlegt. Diese wird später beim Aufruf von `output_data.output_groups()` als Übergabewert benötigt. In `begin` wird die Nummer der ersten Zeile, in der so etwas wie "`var0 ...`" steht, in `end` die Zeile, in der `end_variables` steht, gespeichert. Anschließend wird die Methode `dict_construction.vardata()` aufgerufen, der `begin`, `end` und `file_sas` übergeben werden. Das durch diesen Aufruf erzeugte Wörterbuch haben wir für alle späteren Zugriffe in der Variablen `variable_dictionary` gespeichert. Die Variable `i` wird nun auf die Nummer der Zeile aus *test.sas* gesetzt, in der `begin_state` steht. Dieses `i` wird in `initline` zwischengespeichert, da diese Zeilennummer in der Methode `init()` benötigt wird. Anschließend wird `i` auf die Zeilennummer gesetzt, in der die Anzahl der Zielatome steht. Diese Anzahl wird in der Variablen `number_goal`, das aktuelle `i` in `goalline` gespeichert, da diese Angaben unter anderem in der Methode `goal()` benötigt werden. Das `i` wird jetzt auf die Zeilennummer gesetzt, in der die Anzahl der Operatoren steht, In `opline` wird diese Zeilennummer wieder zwischengespeichert, die Anzahl der Operatoren wird in `number_of_operators` hinterlegt, um diese beiden Angaben in der Methode `parse_operators()` verwenden zu können. So "hangeln" wir uns durch die Datei *test.sas* und übersetzten sie Stück für Stück.

Um unsere Ausgabe-Dateien PDDL-konform zu gestalten, mussten wir sowohl dem Problem als auch der Domäne einen Namen geben. Da die Dateien *test.groups* und *test.sas* die Namen der ursprünglichen Domäne und des Problems nicht mehr beinhalten, wäre der Aufwand, diese Informationen abzufangen, sehr groß gewesen und wir entschieden uns dazu, ihn nicht zu betreiben, sondern die Domäne einfach als "domain translated" und das Problem dementsprechend als "problem translated" zu definieren. Auch "requirements" müssen in

der PDDL-Beschreibung angegeben werden. Da an dieser Stelle dasselbe Problem auftrat wie bei den Namen der Dateien, haben wir hier einfach nur "adl" eingefügt.

Ein weiteres Problem unserer Ausgabe bestand darin, dass der Übersetzer not-Effekte wegkompiliert. Nachdem wir einige Planer auf den von uns erzeugten Dateien hatten laufen lassen, mussten wir feststellen, dass zwar Pläne erzeugt wurden, die sogar kürzer waren als die, die bei Eingabe der originalen PDDL-Beschreibung erzeugt werden, jedoch schienen diese Pläne unvollständig oder zumindest nicht regelkonform zu sein. Daraufhin fügten wir die negierten Vorbedingungen zu den Effekten hinzu, erzeugten also eine Delete-Liste wie folgt:

- Atome die in der Vorbedingung stehen und in der gleichen Faktgruppe wie einer der Effekte sind, müssen in negierter Form den Effekten hinzugefügt werden.
- Atome, die mit ihrem "`not_`" - Gegenstück in einer Faktgruppe sind, müssen ebenfalls in negierter Form den Effekten hinzugefügt werden. Das bedeutet, dass wenn das Atom ohne "`not_`" in den Effekten steht, das Atom mit "`not_`" negiert hinzugefügt werden muss, also "`(not(not_...))`". Wenn das Atom mit "`not_`" in den Effekten steht, muss das Atom ohne "`not_`" negiert hinzugefügt werden.

3.2.4. Zusammenfassung

Die Textdateien *test.sas* und *test.groups* werden mit Hilfe des Übersetzers des Downward Planers erzeugt, woraufhin durch den Aufruf `./sas_to_pddl.py` unser Programm gestartet wird und die Ausgabedateien in PDDL-Form erzeugt werden. Diese Ausgaben sind instanziiert und vereinfacht. Ausgaben unseres Programms können von Planern wie Mips und FF verarbeitet werden, es wird ein gültiger Plan erstellt.

Durch den Übersetzer werden oftmals Konstrukte der Form 'new-axiom@..' erzeugt. Das kommt zum Beispiel dann vor, wenn allquantifizierte Vorbedingungen wegkompiliert werden sollen. Der dabei entstehende existenzquantifizierte Teil wird durch ein neues Axiom dieser Form ersetzt. Die neuen Axiome sind dann auch in unseren Ausgaben enthalten, können aber von den Planern nicht verarbeitet werden. Wir haben das '@' durch einen Bindestrich ersetzt. Die Probleme, in denen die @-Konstrukte vorkommen, enthalten dann aber weitere Konstukte wie "derived" oder "when", die FF oder Mips momentan nicht verarbeiten können.

Im Laufe der Integration unseres Programms in das gemeinsame Tool haben wir zur einfacheren Handhabung noch eine Namensänderung durchgeführt: Die Datei *problem.pddl* heißt nun *grounded_problem.pddl* und die Datei *domain.pddl* heißt *grounded_domain.pddl*.

Hier noch ein Auszug aus den von uns erzeugten Ausgabedateien:

- **problem.sas+:**

```
(:partition
 (:fact-group-0
   (holding-i)
   (ontable-i)
   (on-i-j)
```

```
    ...
    none-of-those
 )
 (:fact-group-1
    ...
```

- **grounded_problem.pddl:**

```
define (problem translated)
(:domain translated)

 (:init
   (clear-c)
   (clear-f)
   (handempty)
   (on-a-d)
   ...
 )
 (:goal
  (and
   (on-a-g)
   (on-b-a)
   (on-c-f)
   ...
  )
 )
)
```

- **grounded_domain.pddl:**

```
(define (domain translated)
 (:requirements :adl)
 (:predicates
        (clear-a)
        (clear-b)
        (clear-c)
        ...
)
(:action pick-up-f
 :precondition
   (and
    (clear-f)
    (handempty)
    (ontable-f)
   )
```

```
 :effect
   (and
    (holding-f)
    (not (clear-f))
    (not (handempty))
    (not (ontable-f))
   )
)
...
```

Aktionen mit konditionalen Effekten:

```
...
(:action wait
 :precondition
   (and
    (affected-cb2)
   )
 :effect
   (and
    (not (affected-cb2))
    (when
     (affected-cb2)
     (closed-cb2)
    )
    (when
     (affected-cb1)
     (closed-cb1)
    )
   )
)
...
```

Regeln:

```
...
(:derived (upstream-cb2-side1-sd6-side1)
   (and
    (closed-sd9)
    (closed-cb2)
    (upstream-cb2-side1-sd9-side1)
   )
)
...
```

Wir haben unser Programm auf verschiedenen Domänen getestet und in folgender Tabelle festgehalten, wie viele instanziierte Operatoren unser Programm erzeugt. Zum Vergleich

Domain	Problem	Translate	AdlToStrips	Ground
Blocks	Track1/Typed/p6-0	73	88	88
	Track1/Untyped/ p5-2	30	60	60
	Track2/p6-1	72	85	85
Elevator	s1-0	4	4	4
Gripper	p01	34	36	36
Satellite	strips/typed/p01	52	59	59
	strips/untyped/p01	52	59	59

Abbildung 3.1.: Ergebnisse unseres Übersetzungsprogramms im Vergleich zu *AdlToStips* und *Ground*

haben wir die Anzahl der instanziierten Operatoren, die AdlToStrips und Ground erzeugen ebenfalls angegeben.

3.3. Symmetrieerkennung

3.3.1. Was sind Symmetrien?

Ein wichtiges Merkmal parametrisierter Prädikate, Funktionen und Aktionsbeschreibungen ist, dass der Tausch von Objekten in der Problembeschreibung für die Domänenbeschreibung transparent ist.

Im Falle von typisierten Bereichen kompilieren viele Planer, so auch MIPS [6] alle typisierten Informationen in zusätzliche Prädikate, verbinden Aktionen mit zusätzlichen Vorbedingungen und erweitern die Initialzustände mit passenden Objekt-zu-Typ Atomen. Als Konsequenz ist eine **Symmetrie** eine Permutation von Objekten im momentanen Zustand, in der Zielrepräsentation und transparent zur Menge der n Operatoren.

Wie bereits in Kapitel 1 erwähnt, gibt es $n!$ mögliche Permutationen der Menge der Objekte. Bei der Beschränkung auf Symmetrien von Objekten des gleichen Typs, reduziert sich die Anzahl aller möglichen Permutationen auf

$$\begin{pmatrix} n \\ t_1, t_2, ..., t_k \end{pmatrix}$$

wobei t_i die Anzahl der Objekte mit Typ i, $i \in \{1, ..., k\}$ ist.

Um die Anzahl der potentiellen Symmetrien auf eine handhabbare Größe zu reduzieren, beschränken wir Symmetrien auf Objekttranspositionen, für die wir höchstens $n(n-1)/2$ Kandidaten haben. Unter Verwendung von Typinformationen reduziert sich diese Anzahl auf

$$\sum_{i=1}^{k} \begin{pmatrix} t_i \\ 2 \end{pmatrix} = \sum_{i=1}^{k} t_i(t_i - 1)/2.$$

Im Folgenden wird die Menge der typisierten Objekttranspositionen mit $\mathcal{SYMM}$ bezeichnet. Auch hier sind wieder einige Definitionen nötig.

Eine **Transposition von Objekten** (o, o') *geltend für ein Fluent* $f = (p\ o_1, ..., o_{k(p)}) \in \mathcal{F}$, *kurz* $f[o \leftrightarrow o']$, *ist definiert als* $(p\ o'_1, ..., o'_{k(p)})$, *mit* $o'_i = o_i$ *wenn* $o_i \notin \{o, o'\}$, $o'_i = o'$ *wenn* $o_i = o$, *und* $o'_i = o$ *wenn* $o_i = o'$, $i \in \{1, ..., k_{(p)}\}$. Mit anderen Worten: Durch Objekttranspositionen werden die Objekte o und o' ausgetauscht.

Die Zeitkomplexität zur Errechnung von $f[o \leftrightarrow o'] \in \mathcal{F}$ hat die Größenordnung $O(k(p))$, mit $k(p)$ als der Anzahl Objektparameter in p. Durch vorherige Berechnung einer Lookup-Tabelle, die den Index von $f' = f[o \leftrightarrow o']$ für alle (o, o') enthält, kann diese Zeitkomplexität auf $O(1)$ reduziert werden.

Eine **Objekttransposition** $v[o \leftrightarrow o']$ *geltend* **für Zustand** S, *kurz* $S[o \leftrightarrow o']$, *ist gleichbedeutend mit* $S[o \leftrightarrow o'] = \{f' = f[o \leftrightarrow o']\}$.

Ein **Planungsproblem** $\mathcal{P} =< \mathcal{S}, \mathcal{O}, \mathcal{I}, \mathcal{G} >$ ist **symmetrisch** unter Berücksichtigung der Objekttransposition $[o \leftrightarrow o']$, abgekürzt als $\mathcal{P}\ [o \leftrightarrow o']$, wenn $\mathcal{I}\ [o \leftrightarrow o'] = \mathcal{I}$ und für alle $G \in \mathcal{G} : G[o \leftrightarrow o'] \in \mathcal{G}$. Wir nehmen an, dass die Zielbeschreibungskomplexität von $\mathcal{G}$ durch $O(|\mathcal{G}| + |\mathcal{V}|)$ begrenzt ist.

Symmetrien können während der Entwicklung in einem vorwärts-suchenden Planer wie MIPS [6] verschwinden oder sich neu bilden. Zielbedingungen ändern sich jedoch mit der Zeit nicht. Nur der Initialzustand $\mathcal{I}$ geht in den momentanen Zustand $\mathcal{C}$ über. Daher verfeinern wir die Menge $\mathcal{SYMM}$ aller potentiell existierenden Objekttranspositionen in einer Pre-Compiling - Phase zu $\mathcal{SYMM}' \leftarrow \{(o,o') \in \mathcal{SYMM} \ |\mathcal{G}[o \leftrightarrow o'] = \mathcal{G}\}$. Normalerweise ist $|\mathcal{SYMM}'|$ wesentlich kleiner als $|\mathcal{SYMM}|$.

Nun können wir auch die Menge $\mathcal{SYMM}''(\mathcal{C}) \leftarrow \{(o,o') \in \mathcal{SYMM}' \ |C[o \leftrightarrow o'] = C\}$ der Symmetrien, die bereits im momentanen Zustand bestehen, effizient berechnen. Die Voraussetzung $\mathcal{C}[o \leftrightarrow o'] = C$ beinhaltet keine symmetrischen Pfade, die von unterschiedlichen Zuständen ausgehen! Mit anderen Worten: Komplexe Objektsymmetrien der Form $[o_1 \leftrightarrow o'_1][o_2 \leftrightarrow o'_2]$ werden nicht entdeckt [6].

Sei $\Gamma(\mathcal{C})$ die Menge der Operatoren, die im Zustand $\mathcal{C} \in \mathcal{S}$ anwendbar sind. Die **Pruning Menge** $\Delta(\mathcal{C})$ ist definiert als die Menge aller Operatoren, die einen symmetrischen Partner und keinen minimalen Index haben. Die Symmetriereduktion $\Gamma'(\mathcal{C})$ ist definiert als $\Gamma(\mathcal{C}) \setminus \Delta(\mathcal{C})$. Auf diese Art und Weise kann der Verzweigungsfaktor des Problems bedeutend reduziert werden. Man kann beweisen, dass die Reduzierung der Menge der Operatoren $\Gamma(\mathcal{C})$ zu $\Gamma'(\mathcal{C})$ während der Untersuchung des Planungsproblems die Vollständigkeit erhält.

Da die Zielfunktion des Plans ähnlich wie die Initial- und Zielzustände auf instanziierte Prädikate und Objekte verweist, kann sie symmetrieverletzend sein. Um Optimalität zu erhalten, muss zusätzlich überprüft werden, ob durch das Austauschen von Objekten das Planziel beeinflusst wird.

3.3.2. Aufgabenstellung

Unsere Aufgabe war es, ein Programm zu schreiben, das Symmetrien in Planungsproblemen erkennt. Diese Erkenntnisse sollen dann spätere Planungsprozesse vereinfachen und verkürzen. Die Vorgehensweise wurde uns selbst überlassen.

3.3.3. Vorgehensweise

Auch hier stellten wir uns zuerst die Frage der zu verwendenden Programmiersprache. Wir entschieden uns wieder für Python, da wir die Sprache bereits gut kannten und Python viele für unsere Arbeit hilfreiche Module zur Stringmanipulation anbietet.

Unsere erstes Ziel war es, die Symmetrien in den Zielzuständen zu erkennen. Dazu benötigten wir zuerst die Objekte der Problem-Eingabe. Da diese Objekte in unserer instanziierten Textdatei *grounded-problem.pddl* nicht mehr enthalten sind, mussten wir die Objekte der originalen Eingabe verwenden. Wir haben einige Veränderungen in der Datei `translate.py` vorgenommen, so dass zusätzlich die Textdatei *objects* erstellt wird, in der die Objekte mit zugehörigem Typ aufgelistet sind.

Auszug der Textdatei *objects*:

```
satellite0: satellite
instrument0: instrument
image1: mode
spectrograph2: mode
thermograph0: mode
star0: direction
groundstation1: direction
groundstation2: direction
phenomenon3: direction
phenomenon4: direction
star5: direction
phenomenon6: direction
```

Im Zuge unserer Arbeit haben wir die Datei `translate.py` später vereinfacht, so dass nur noch die zur Ausgabe nötigen Schritte durchgeführt werden und haben die Datei in `objects.py` umbenannt.

Zur Erstellung eines Wörterbuchs haben wir die Datei `obj_to_objtype_dictionary.py` erzeugt, in der wir zu Beginn die Methode `open_objects()` implementiert haben. Diese Methode bildet mit Hilfe der Datei *objects* eine Liste, so dass jede Zeile der *objects* - Datei einem Listenelement entspricht. Dieser Vorgang dient später zur besseren Handhabung der Datei. Die Methode `dictionary()` erstellt auf Basis der Liste ein Wörterbuch, in dem alle Objekte des Problems als Schlüssel und die zugehörigen Typen als Werte gespeichert werden. Die Liste wird dazu elementweise bearbeitet und bei ':' gesplittet, so dass der erste Teil als Objekt und der zweite als Typ abgespeichert werden kann. Desweiteren wird eine Liste 'valuelist' erstellt, in der alle im Problem enthaltenen Objekttypen gespeichert und doppelte Einträge entfernt werden.

In der ersten Version unseres Programms haben wir durch eine Veränderung in der Methode `output_goal()` der Datei `output_data.py` unseres Übersetzungsprogramms zusätzlich noch eine Datei *atome* erstellt, in der die Zielatome aufgelistet werden.

Im Laufe unserer Arbeit erschien uns diese Vorgehensweise als zu umständlich. Daher wird nun im Aufruf des Programms zur Symmetrieerkennung die Pfadangabe des Problems erwartet, so dass die Zielatome mit Hilfe der Methode `parse_goal()` direkt aus dem Ori-

ginalproblem übernommen werden können. Mit Hilfe dieser Zielatomliste, die wir in einer Textdatei namens *goal_atoms* dem Benutzer zu Verfügung stellen, haben wir die Objekttypen in solche, die im Ziel vorhanden sind, und solche, die es nicht sind, unterteilt.

Auszug der Datei *goal_atome*:
```
(have_image phenomenon4 thermograph0)
(have_image star5 thermograph0)
(have_image phenomenon6 thermograph0)
```

Die Methode `goal_obj()` durchläuft die Atomliste und findet für jedes enthaltene Objekt den entsprechenden Typen mit Hilfe des Wörterbuchs. Diese Objekttypen werden - falls noch nicht vorhanden - an eine Liste gehängt, so dass man nach diesem Durchlauf eine Liste aller im Ziel vorhandenen Objekttypen hat.

Die Methode `find_obj()` erstellt mit Hilfe der von `goal_obj()` erzeugten Liste für jeden vorhandenen Objekttypen eine weitere Liste, die als erstes Element den Objekttypen selbst und dann alle zugehörigen Objekte enthält. Diese Listen werden als Elemente an eine große Liste angehängt.

Im Laufe unserer Arbeit stellten wir fest, dass diese Unterteilung eigentlich nicht nötig war, da *alle* Objettypen - ob im Ziel vertreten oder nicht - in einer Matrix gespeichert werden sollten. Also änderten wir unseren Code dementsprechend. Wir konnten auf die Methode `goal_obj()` verzichten und haben die Methode `find_obj()` umbenannt und verändert:

Wir erzeugten die Datei `state_sym.py`, in die wir die Methode `find_objects()` (vorher `find_obj()`) einfügten, die jetzt *alle* Objekte zu vorhandenen Objekttypen auf die oben erwähnte Art und Weise in einer Liste speichert.

Zur Festhaltung der Symmetrien haben wir Matrizen erstellt, in denen eine 1 steht, wenn eine Symmetrie zwischen Objekten besteht und eine 0 sonst. Für jeden Objekttypen wird eine eigene Matrix erstellt. Eine Matrix besteht in unserem Fall aus einer großen Liste, deren Elemente jeweils einer Matrixzeile entsprechen. Das erste Element dieser Liste ist zur übersichtlicheren Darstellung die Auflistung aller zu dem aktuellen Objekttypen gehörenden Objekte. Die restlichen Elemente sind Listen, in denen das erste Element jeweils das Objekt selbst ist (auch hier zur besseren Darstellung), gefolgt von einer 0-1-Sequenz, die die vorhandenen Symmetrien darstellt und im weiteren Verlauf des Programms angehängt wird.

Dazu implementierten wir die Methode `build_matrix()` in der Datei `matrix.py`. Dort werden die Methoden `exchange()` und `exchange2()` aufgerufen. Diese bekommen als Übergabewerte den jeweiligen Zustand (bis zu diesem Punkt also lediglich das Ziel) in Form einer alphabetisch sortierten Auflistung der Atome, das auszutauschende Objekt (altes Objekt) und das Objekt, gegen das ausgetauscht werden soll (neues Objekt). Die Methode `exchange()` führt dabei eine einfache Substitution durch, während die Methode `exchange2()` einen wechselseitigen Austausch vornimmt. Die Ergebnisse werden wieder alphqabetisch sortiert zurückgegeben und mit dem ursprünglichen Zustand verglichen. Sind die Zustände äquivalent, wird eine 1 an die entsprechende Zeilenliste der Matrix gehängt, sonst eine 0. So wird mit sämtlichen Kombinationen der Objekts verfahren.

In der Datei `matrix.py` befinden sich zudem noch zwei Methoden zur Ausgabe der Symmetrien: `output_symmetry_matrix()` schreibt die Matrizen in die Textdateien *subsymmetries* und *transsymmetries*, `output_symmetry()` schreibt nur die vorhandenen Symmetrien

getrennt nach Substitutionssymmetrien und wechselseitigen Symmetrien in die Textdatei *symmetries*.

Die eben beschriebene Methode `build_matrix()` wird in der Datei `state_sym.py` durch die Methode `find_symmetries()`aufgerufen.

Somit konnten wir die Zielsymmetrien ermitteln.

Als nächstes beschäftigten wir uns damit, die Symmetrieerkennung auf *alle* Zuständen zu erweitern:

Die einzelnen Zustände inklusive Start- und Zielzustand erhielten wir durch eine Erweiterung des Validators, wodurch zusätzlich eine Textdatei *state-sequence* erzeugt wird.

Auszug der Datei *state-sequence*:

```
''(on_board instrument0 satellite0)
(power_avail satellite0)
(pointing satellite0 phenomenon6)
(calibration_target instrument0 groundstation2)
(supports instrument0 thermograph0)
(total-time) [1]'',
''(on_board instrument0 satellite0)
(pointing satellite0 phenomenon6)
(calibration_target instrument0 groundstation2)
(power_on instrument0)
(supports instrument0 thermograph0)
(total-time) [1]'',
...
```

Wir mussten allerdings weiterhin den Zielzustand aus dem Originalproblem verwenden, da der in der *state-sequence* bereits zu stark instanziiert war. Hätten wir diesen Zustand dennoch benutzt, wären viele Zielsymmetrien verloren gegangen.

In der Datei `state_sym.py` befindet sich nun also die Methode `open_stateseq()`, der der Pfad der *state-sequence* übergeben wird. Die Datei wird geöffnet und zur Liste konvertiert. Weiter implementierten wir die Methode `parse_state()`, die als Übergabewert die state-sequence-Liste bekommt. Sie formatiert die Sequenz und erzeugt eine Liste, die als Elemente Listen der Zustandsatome des jeweiligen Zustands enthält.

Es werden nun erst die Symmetrien des Zielzustands ermittelt, um dann die Symmetrien in den anderen Zuständen mit denen im Ziel vergleichen zu können. Denn nur die Symmetrien, die auch im Ziel noch gelten, sind 'echte' Symmetrien. Die Organisation der Aufrufe übernimmt die Methode `find_symmetries()`.

Als zusätzliche Möglichkeit haben wir noch die Datei `delete.py` mit der Methode `delete()` erstellt. Mit dieser kann der Benutzer optional einen Objekttypen inklusive aller zugehörigen Objekte aus der objects-Liste entfernen. Dazu erhält `delete()` als Übergabewert den zu löschenden Objekttypen. In der Liste wird danach gesucht und die Datei *objects* - ohne diesen Objekttypen - neu erzeugt. Dann kann die Symmetrieerkennung mittels der 'neuen' Datei *objects* abermals angestoßen werden. Es muss beachtet werden, dass die Datei *objects* durch das Entfernen nicht leer werden darf. Wird vom Benutzer ein Objekttyp eingegeben,

der nicht in *objects* enthalten ist, erscheint eine Fehlermeldung.

3.3.4. Zusammenfassung

Der Programmablauf ist jetzt also wie folgt:

Der Nutzer ruft `./objects.py <problem> (<domain>)` auf, um die Textdatei *objects* zu erzeugen. Anschließend muss er einen gültigen Plan mit Hilfe eines beliebigen Planers erzeugen und diesen Plan durch den modifizierten Validator laufen lassen, um die Textdatei *state-sequence* zu erzeugen. Dann kann `./state_sym.py <state-sequence> <problem>` aufgerufen werden.

Werden fehlerhafte Angaben gemacht, erscheint eine der folgenden Meldungen:

```
Cannot find state-sequence at ../
Cannot find problem at ../
Usage: state_sym.py <state-sequence> <problem>
```

Mit Hilfe der ersten Pfadangabe wird die Datei *state-sequence* geöffnet und geparsed, so dass die Zustände einzeln zur Verfügung stehen. Das Problem muss ebenfalls angegeben werden, da daraus die Zielatome gewonnen werden. Anschließend wird mit Hilfe der Methode `dictionary()` aus `obj_to_objtype_dictionary.py` das Wörterbuch erstellt und die Methode `find_objects()` aufgerufen. Danach wird die Methode `find_symmetries()` ausgeführt, die wiederum die Methode `build_matrix()` aus `matrix.py` nutzt. Dieser wird zuerst der Zielzustand übergeben, für den die Symmetrien mit Hilfe der Methoden `exchange()` und `exchange2()` ebenfalls aus `matrix.py` erkannt und in einer separaten Matrix gespeichert werden. Anschließend werden diese Methoden für alle anderen Zustände wiederholt ausgeführt, die erkannten Symmetrien werden mit denen im Ziel verglichen und durch die Aufrufe der Ausgabe-Methoden in die oben genannten Textdateien *subsymmetries*, *transsymmetries* und *symmetries* geschrieben.

Auszug aus den von uns erzeugten Ausgabedateien am Beispiel der satellite-Domäne:

- **subsymmetries:**

```
goal-symmetries:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
['-', ['thermograph0', 'spectrograph2', 'image1']]
['thermograph0', 1, 0, 0]
['spectrograph2', 1, 1, 1]
['image1', 1, 1, 1]
-----------------------------------------------
['-', ['phenomenon6', 'star5', 'phenomenon4', 'phenomenon3',
'groundstation2', 'groundstation1', 'star0']]
```

```
['phenomenon6', 1, 0, 0, 0, 0, 0, 0]
['star5', 0, 1, 0, 0, 0, 0, 0]
['phenomenon4', 0, 0, 1, 0, 0, 0, 0]
['phenomenon3', 1, 1, 1, 1, 1, 1, 1]
['groundstation2', 1, 1, 1, 1, 1, 1, 1]
['groundstation1', 1, 1, 1, 1, 1, 1, 1]
['star0', 1, 1, 1, 1, 1, 1, 1]
-----------------------------------------------

init-symmetries:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
['-', ['thermograph0', 'spectrograph2', 'image1']]
['thermograph0', 1, 0, 0]
['spectrograph2', 1, 1, 1]
['image1', 1, 1, 1]
-----------------------------------------------
['-', ['phenomenon6', 'star5', 'phenomenon4', 'phenomenon3',
'groundstation2', 'groundstation1', 'star0']]
['phenomenon6', 1, 0, 0, 0, 0, 0, 0]
['star5', 0, 1, 0, 0, 0, 0, 0]
['phenomenon4', 0, 0, 1, 0, 0, 0, 0]
['phenomenon3', 1, 1, 1, 1, 1, 1, 1]
['groundstation2', 0, 0, 0, 0, 1, 0, 0]
['groundstation1', 1, 1, 1, 1, 1, 1, 1]
['star0', 1, 1, 1, 1, 1, 1, 1]
-----------------------------------------------
...

symmetries in state 12:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
...
```

- **transsymmetries:**

```
goal-symmetries:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
['-', ['thermograph0', 'spectrograph2', 'image1']]
['thermograph0', 1, 0, 0]
['spectrograph2', 0, 1, 1]
['image1', 0, 1, 1]
-----------------------------------------------
['-', ['phenomenon6', 'star5', 'phenomenon4', 'phenomenon3',
'groundstation2', 'groundstation1', 'star0']]
['phenomenon6', 1, 1, 1, 0, 0, 0, 0]
['star5', 1, 1, 1, 0, 0, 0, 0]
['phenomenon4', 1, 1, 1, 0, 0, 0, 0]
['phenomenon3', 0, 0, 0, 1, 1, 1, 1]
['groundstation2', 0, 0, 0, 1, 1, 1, 1]
['groundstation1', 0, 0, 0, 1, 1, 1, 1]
['star0', 0, 0, 0, 1, 1, 1, 1]
-----------------------------------------------

init-symmetries:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
['-', ['thermograph0', 'spectrograph2', 'image1']]
['thermograph0', 1, 0, 0]
['spectrograph2', 0, 1, 1]
['image1', 0, 1, 1]
-----------------------------------------------
['-', ['phenomenon6', 'star5', 'phenomenon4', 'phenomenon3',
'groundstation2', 'groundstation1', 'star0']]
['phenomenon6', 1, 0, 0, 0, 0, 0, 0]
['star5', 0, 1, 1, 0, 0, 0, 0]
['phenomenon4', 0, 1, 1, 0, 0, 0, 0]
['phenomenon3', 0, 0, 0, 1, 0, 1, 1]
```

```
['groundstation2', 0, 0, 0, 0, 1, 0, 0]
['groundstation1', 0, 0, 0, 1, 0, 1, 1]
['star0', 0, 0, 0, 1, 0, 1, 1]
-----------------------------------------------
...

symmetries in state 12:

['-', ['satellite0']]
['satellite0', 1]
-----------------------------------------------
['-', ['instrument0']]
['instrument0', 1]
-----------------------------------------------
['-', ['thermograph0', 'spectrograph2', 'image1']]
['thermograph0', 1, 0, 0]
['spectrograph2', 0, 1, 1]
['image1', 0, 1, 1]
-----------------------------------------------
['-', ['phenomenon6', 'star5', 'phenomenon4', 'phenomenon3',
'groundstation2', 'groundstation1', 'star0']]
['phenomenon6', 1, 0, 0, 0, 0, 0, 0]
['star5', 0, 1, 0, 0, 0, 0, 0]
['phenomenon4', 0, 0, 1, 0, 0, 0, 0]
['phenomenon3', 0, 0, 0, 1, 0, 1, 1]
['groundstation2', 0, 0, 0, 0, 1, 0, 0]
['groundstation1', 0, 0, 0, 1, 0, 1, 1]
['star0', 0, 0, 0, 1, 0, 1, 1]
-----------------------------------------------
...
```

- **symmetries:**

```
goal-symmetries:

spectrograph2 -> thermograph0
spectrograph2 -> image1
image1 -> thermograph0
image1 -> spectrograph2
phenomenon3 -> phenomenon6
phenomenon3 -> star5
phenomenon3 -> phenomenon4
phenomenon3 -> groundstation2
phenomenon3 -> groundstation1
```

```
phenomenon3 -> star0
groundstation2 -> phenomenon6
groundstation2 -> star5
...
groundstation1 -> groundstation2
groundstation1 -> star0
...
star0 -> groundstation2
star0 -> groundstation1
-------------------------------------------------
spectrograph2 <-> image1
image1 <-> spectrograph2
phenomenon6 <-> star5
phenomenon6 <-> phenomenon4
star5 <-> phenomenon6
star5 <-> phenomenon4
phenomenon4 <-> phenomenon6
phenomenon4 <-> star5
phenomenon3 <-> groundstation2
phenomenon3 <-> groundstation1
phenomenon3 <-> star0
groundstation2 <-> phenomenon3
groundstation2 <-> groundstation1
groundstation2 <-> star0
groundstation1 <-> phenomenon3
groundstation1 <-> groundstation2
groundstation1 <-> star0
star0 <-> phenomenon3
star0 <-> groundstation2
star0 <-> groundstation1
-------------------------------------------------

init-symmetries:

spectrograph2 -> thermograph0
spectrograph2 -> image1
image1 -> thermograph0
image1 -> spectrograph2
phenomenon3 -> phenomenon6
phenomenon3 -> star5
phenomenon3 -> phenomenon4
phenomenon3 -> groundstation2
phenomenon3 -> groundstation1
phenomenon3 -> star0
groundstation1 -> phenomenon6
```

```
groundstation1 -> star5
groundstation1 -> phenomenon4
groundstation1 -> phenomenon3
groundstation1 -> groundstation2
groundstation1 -> star0
star0 -> phenomenon6
star0 -> star5
star0 -> phenomenon4
star0 -> phenomenon3
star0 -> groundstation2
star0 -> groundstation1
--------------------------------------------------
spectrograph2 <-> image1
image1 <-> spectrograph2
star5 <-> phenomenon4
phenomenon4 <-> star5
phenomenon3 <-> groundstation1
phenomenon3 <-> star0
groundstation1 <-> phenomenon3
groundstation1 <-> star0
star0 <-> phenomenon3
star0 <-> groundstation1
--------------------------------------------------
...

symmetries in state 12:

spectrograph2 -> thermograph0
spectrograph2 -> image1
image1 -> thermograph0
image1 -> spectrograph2
phenomenon3 -> phenomenon6
phenomenon3 -> star5
phenomenon3 -> phenomenon4
...
--------------------------------------------------
spectrograph2 <-> image1
image1 <-> spectrograph2
phenomenon3 <-> groundstation1
phenomenon3 <-> star0
groundstation1 <-> phenomenon3
groundstation1 <-> star0
star0 <-> phenomenon3
star0 <-> groundstation1
--------------------------------------------------
```

...

Optional kann die Methode `./delete <objecttype>` aufgerufen werden.

Wir haben unser Programm auf verschiedenen Domänen getestet und in folgender Tabelle festgehalten, wie viele Symmetrien jeweils vorkommen.

Domain	Problem	Sub / Trans goal	Sub / Trans init	state nr./ Sub/ Trans	obj
Airport	nontemporal/adl/p03	242/ 221	32/ 2	24/ 32/ 2	ok
	nontemporal/strips/p04	1.529/ 1.490	5/ 1.334	20/ 5/ 1.333	ok
Blocks	Track1/Typed/p6-0	-/ -	-/ -	6/ -/ -	ok
	Track1/Untyped/ p5-2	-/ -	-/ -	20/ -/ -	ok
	Track2/p6-1	-/ -	-/ -	20/ -/ -	ok
Elevator	s1-0	2/ 2	-/ -	-/ -	ok
Grid	p01	652/ 620	-/ -	20/ -/ -	†
Gripper	p06	3/ 184	-/ 184	46/ -/ 88	†
Logistics	98/p01	163/ 130	-/ 2	55/ -/ -	†
	00/p7-1	30/ 18	-/ -	69/ -/ 2	ok
Mprime	p01	94/ 86	-/ -	7/ -/ -	†
Mystery ???	???	94/ 86	-/ -	7/ -/ -	†
Psr	large/adl-derived/p01	320/ 296	-/ -	8/ -/ -	ok
	middle/adl-derived/p01	224/ 238	-/ -	10/ -/ -	ok
Schedule	2000/Typed/p10-0	114/ 88	29/ 16	17/ 21/ 8	ok
Pipesworld	nontankage/nontemp/ strips/p01	44/ 34	-/ -	9/ -/ -	ok
Satellite	strips/typed/p01	28/ 20	22/ 10	14/ 22/ 8	ok
	strips/untyped/p01	88/ 62	-/ 10	14/ -/ 8	†

Sub = Substitutionssymmetrien Trans = wechselseitige Symmetrien obj = Textdatei *objects*
†= in Textdatei *objects* werden Objekttypen nicht erkannt

Abbildung 3.2.: Ergebnisse Symmetrieerkennung

3.3.5. Ausblick

Eine wesentlich *Effizienzsteigerung* könnte dadurch erreicht werden, dass die Matrizen nicht jedes mal von Neuem erzeugt werden, sondern dass man Matrizen für jeden Objekttypen erzeugt und mit 1 vorinitialisiert. Somit müssten dann triviale Substitutionen und wechselseitige Vertauschungen nicht mehr durchgeführt werden, da diese immer symmetrisch sind. Darüber hinaus sind die wechselseitigen Matrizen symmetrisch. Das bedeutet, dass nur nur die Kombinationen unter- oder oberhalb der Diagonalen wirklich berechnet werden müssen und die restlichen Einträge einfach übernommen werden können.

Wir haben mit dieser Arbeit bereits begonnen, konnten die Implementierung jedoch aus zeitlichen Gründen nicht abschließen.

Eine sinnvolle *Erweiterung* unserer Symmetrieerkennung wäre es, auch permutativen Austausch von Objekten durchzuführen. Auf diese Art und Weise werden dann auch komplexe Objektsymmetrien der Form $[o_1 \leftrightarrow o_1'][o_2 \leftrightarrow o_2']$ erkannt.

Kapitel 4.

Berechnung und Anwendung einer Ziel-Ordnung

Michael Nelskamp

Zunächst einmal wird nach einer geeigneten Definition einer Ziel-Ordnung gesucht, welche das Problemziel in Etappenziele gliedern und effizient berechenbar sein soll, wobei die Optimalitätsgarantie vernachlässigt werden darf [31].

Eine geeignete Zielordnung hat sich dann als reine Preprocessing-Prozedur in Java implementieren lassen, welche den Einsatz innerhalb einer Planungsschleife ermöglicht hat. Dabei werden die durch die Vorverarbeitung mittels der Python-translate-Methode gegroundeten STRIPS-Domänen und -Probleme mittels eines zweckmässigen Parsers in die Java-VM eingelesen. Die Planungsschleife, ebenfalls in Java implementiert, lässt dann nach einmaliger Berechnung der Zielordnung iterativ den ff-Planer auf den Etappenzielen und den sich ergebenden Zwischenzuständen laufen. Da ff nur einen Plan ausgiebt, wird der Validator verwendet, um an die Zwischenzustände zu kommen.

4.1. Theoretische und praktische Bedeutung verschiedener Ziel-Ordnungen

Zunächst werden also die theoretischen Ergebnisse vorgestellt, welche das Feld der uns zur Verfügung stehenden Möglichkeiten abstecken.

Als erstes versuchen wir, für STRIPS-Probleminstanzen eine Ordnungsrelation über der Menge der Zielatome zu definieren. Diese Ordnungsrelation soll uns dazu dienen, Teilziele zu definieren, die von einem Planungsalgorithmus nacheinander gelöst werden können.

Definition Für eine Probleminstanz (O, I, G) bezeichne $s_{(A,\neg B)}$ einen beliebigen erreichbaren Zustand, in dem das Atom A gerade erreicht wurde, und in dem B nicht gilt. D.h. $B \notin s_{(A,\neg B)}$ und es gibt eine Folge von Aktionen welche von I in den Zustand $s_{(A,\neg B)}$ führt und deren letzte Aktion A wahr macht.

Definition (Erzwungene Ordnung) Sei (O, I, G) ein Planungsproblem, $A, B \in G$. Dann gelte $B \leq_f A$ genau dann, wenn

$$\forall s_{(A,\neg B)} : \neg\exists P^O : B \in Result(s_{(A,\neg B)}, P^O),$$

wobei P^O einen Plan bzgl. der Operatormenge O bezeichne.

Definition (vernünftige Ordnung) Sei (O, I, G) ein Planungsproblem, $A, B \in G$. Dann gelte $B \leq_r A$ genau dann, wenn

$$\forall s_{(A,\neg B)} : \neg\exists P^{O_A} : B \in Result(s_{(A,\neg B)}, P^{O_A}).$$

Hierbei sei O_A die Menge der Aktionen, die A nicht in ihrer Löschliste haben.

Komplexität Das Entscheidungsproblem F_ORDER, ob für O, I, A, B denn $B \leq_f A$ gilt, ist PSPACE-schwierig. Analog ist R_ORDER PSPACE-schwierig.

4.1.1. Berechnung von Zielordnungen

$A \leq_r B$ zu entscheiden, ist also schwierig. Was die erzwungene Ordnung $\leq_f$ betrifft, so wird sich zeigen, dass sie bisher in der Anwendung gleich der leeren Menge ist, also keine Rolle spielt. Man kann sagen, dass in fast allen in der Literatur betrachteten Domänen und Probleminstanzen $A \leq_f B$ nicht vorkommt. Dehalb konzentrieren wir uns darauf, eine effizient entscheidbare Ordnung $\leq_e$ zu finden, welche $\leq_r$ zumindest hinreichend bedingt.
Eine Möglichkeit ist es, Graphplan zu verwenden. Sind I und O gegeben berechnet Graphplan nacheinander die Schichten des Plangraphen, bis ein Fixpunkt erreicht wird. Alle Atome der letzten Schicht, welche als *mutual exclusive* zu A gekennzeichnet sind, können in einem erreichbaren Zustand nie zusammen mit A gelten. Sei also

$$F_{GP}^{A} := \{p \mid p \text{ ist exklusiv zu } A\}.$$

Dass heißt, der Plangraph kann einmal berechnet werden, um dann für alle $A \in G$ die Menge F_{GP}^{A} zu berechnen.
Es gilt also $F_{GP}^{A} \cap s_A = \emptyset$ für alle Zustände s_A mit $A \in s_A$, die von I aus mittels Operatoren aus O erreichbar sind.

Definition (Effiziente Ordnung) Für ein Planungsproblem (O, I, G) mit $A, B \in G$ gelte $B \leq_e A$ genau dann, wenn

$$\forall o \in O_A : B \in add(o) \Rightarrow pre(o) \cap F_{GP}^{A} \neq \emptyset.$$

Satz Es gilt $B \leq_e A \Rightarrow B \leq_r A$.

Im Folgenden wollen wir zeigen, dass die erzwungene Ordnung in den Benchmark- Planungsproblemen nicht vorkommt.

Definition (Inverse Aktion) Operator $\overline{o}$ wird *invers* zu Operator o genannt falls

(i) $pre(\overline{o}) \subset pre(o) \cup add(o) - del(o)$
(ii) $add(\overline{o}) = del(o)$
(iii) $del(\overline{o}) = add(o)$

Falls $del(o) \subset pre(o)$ und $s \cap add(o) = \emptyset$ gilt, kann $\overline{o}$ angewendet werden. Dann gilt $Result(Result(s, \langle o \rangle), \langle \overline{o} \rangle) = s$.

Definition und Satz (Invertierbares Problem) Ist also (O, I, G) gegeben so dass für alle Aktionen o und erreichbaren Zustände s gilt $del(o) \subset pre(o)$ und $pre(o) \subset s \Rightarrow add(o) \cap s = \emptyset$, und gibt es ausserdem zu jedem o ein Inverses, so ist (O, I, G) *invertierbar*, wobei ein Planungsproblem (O, I, G) genau dann invertierbar heiße, wenn

$$\forall s : \forall P^O : \exists \overline{P}^O : Result(Result(s, P^O), \overline{P}^O) = s$$

wobei s die erreichbaren Zustände bezeichne.

Satz Ein invertierbares Problem (O, I, G) hat keinen *Deadlock*, d.h. es gibt keinen erreichbaren Zustand s, von dem aus man G nicht erreichen kann.

Folglich gibt es bei einem invertierbaren Problem keine erzwungene Ordnung. Typischerweise betrachtete Probleme sind invertierbar. Betrachte z.B. eine Domäne, die einen Stack modelliert. Wenn der Initialzustand dem leeren Stack entspricht, so sollte man von jedem erreichbaren Zustand aus wieder zurück in den Initialzustand kommen, indem man den Stack leert.

4.1.2. Die Goal Agenda

Gegeben sei ein Problem (O, I, G). Zunächst wird für jedes Paar $A, B \in G$ berechnet, ob $A \leq B$ oder $B \leq A$ gilt, oder ob beides gilt, oder ob keines von beidem gilt. Dann wird eine geeignete Partition $G_1, ..., G_n$ von G berechnet. Dazu betrachten wir zunächst den Graphen (V, E) mit $V = G$ und $E = \{(A, B) \in G \times G \mid A \leq B\}$. Liegen A, B auf einem Kreis, so soll $A, B \in G_i$ gelten. Gibt es einen Pfad von A nach B aber nicht von B nach A so gelte $A \in G_i, B \in G_k, i < k$.
Zur Berechnung der G_i betrachte den transitiven Abschluss des Graphen. Sei $d(A) := d_{in}(A) - d_{out}(A)$ der Grad des Knotens A. Dann enthält G_i genau die Atome i-kleinsten Grades.

Der Agenda-gesteuerte Planungsalgorithmus

Gegeben sei $P = (O, I, G)$.
Setze $I_1 := I$ Sei G'_i die Vereinigung von $G_1, ..., G_i$. Sei $I_{i+1} := Result(I_i, G'_i)$, d.h. löse das neue Teilproblem (O, I_i, G'_i).

Durch die Goal-Agenda kann z.B. in der Blocksworld-Domï¿½e die Performance erheblich verbessert bzw. die Lösung der Probleme ermöglicht werden. Allerdings können die gefundenen Pläne länger sein; Blöcke werden z.B. an Positionen gesetzt, welche den Weg zum momentanen Ziel nicht verstellen, jedoch den Weg zu noch kommenden Zielen, so dass Blöcke später wieder zurückgesetzt werden müssen.

4.2. Unsere Implementierung einer Ziel-Ordnung

Unser Ziel in der PG ist es folglich gewesen, $\leq_r$ effizient zu approximieren, und wir haben uns für einen heuristischen Ansatz entschieden, der ohne Graphplan auskommt, dafür schneller ist und bei manchen Domänen in gewisser Hinsicht weniger vollständig.

Eine solche Approximation haben wir in der Java-Klasse GoalOrdering.java innerhalb der Methode computeGoalAtomRelation() dargestellt in Abbildung 4.1 realisiert.

Für jedes Zielatom A wird seperat in einem Durchlauf der for-Schleife die Menge von Atomen B mit $B \leq A$ berechnet.

Dazu wird zunächst in Zeile 4 die Menge der Atome berechnet, die den Schnitt der Delete-Listen aller Operatoren, welche A in ihrer Add-Liste enthalten, bilden. D.h. set_F enthält nur Atome, welche in jedem ersten erreichten Zustand der A enthält nicht gelten können.

Zu dieser initialen Menge set_F wird in Zeile 8 die Menge set_O der Operatoren berechnet, die A nicht löschen und deren Vorbedingung kein Atom aus set_F enthält.

```
 1: ordering = new boolean[goal.length][goal.length];

 2: for ( int ii = 0; ii < goal.length; ii++ ) {
 3:    int A = goal[ii];
 4:    int[] set_F = directAnalysis(A);
 5:    set_F = removeEmptyEntries( set_F );

 6:    //berechne die Menge der Operatoren, die A nicht
         in ihrer delete-List haben:
 7:    int[] set_O_A = getOpsNotDeleting(A);
 8:    int[] set_O = computeSpecialOperatorSet(set_O_A,set_F);
 9:    set_O = removeEmptyEntries( set_O );

10:    // Fixpunkt-Iteration:
11:    boolean fixpoint_reached = false;
12:    while(!fixpoint_reached) {
13:       fixpoint_reached = true;
14:       for(int i = 0; i < set_F.length; i++)
15:          if(set_F[i]!=-1)
16:             if(possiblyAchievable(set_F[i], set_O)) {
17:                set_F[i]=-1;
18:                set_O = computeSpecialOperatorSet(set_O_A,set_F);
19:                fixpoint_reached = false;
20:             }

21:    }

22:    // Berechnung der Ordnungs-Relation
23:    for ( int i = 0; i < goal.length; i++ ) {
24:       ordering[i][ii] = false;
25:       if ( i != ii )
26:          if ( !possiblyAchievable(goal[i], set_O) ) ordering[i][ii] = true;
27:    }

28: }
```

Abbildung 4.1.: die Methode ComputeGoalAtomRelation()

Nun wird in der while-Schleife set_F weiter verkleinert und set_O weiter vergrössert, bis ein Fixpunkt erreicht ist. Unmittelbar vor der while-Schleife umfasst set_F die Atome, die in einem Plan mit dem wahrmachen des Zielatoms A erstmal nicht wahr sein können, und entsprechent umfasst set_O die Menge der Operatoren, die in einem Plan direkt auf dem Operator der A wahr gemacht hat folgen können. Durch die Fixpunktreduktion soll set_F auf die Atome reduziert werden, die zu garkeinem Zeitpunkt nach erreichen von A mehr wahr werden können, wobei sich set_O nach und nach auf die Menge der Operatoren erweitern soll, die zu irgendeinem Zeitpunkt nach erreichen von A angewendet werden können.

Dazu haben wir einen approximativen Test $possiblyAchievable(f, set\_O)$ angewendet, der entscheiden soll, ob ein Atom f durch Anwenden der Operatoren aus set_O erreicht werden kann.

In Zeile 16 der Fixpunktiteration wird also, falls set_O im vorherigen Durchlauf Operatoren dazugewonnen hat, für jedes f aus set_F getestet, ob es immer noch nicht erreichbar ist, und ggf. in set_F beibehalten, ansonsten entfernt.

Ist mit Beendigung der Fixpunktiteration in Zeile 21 die Menge der nach erreichen von A anwendbaren Operatoren berechnet worden, wird letztendlich in Zeile 26 wieder mit Hilfe von $possiblyAchievable$ getestet, ob ein Zielatom B nach A noch erreichbar ist, und falls nicht, $B \leq A$ gesetzt.

Definition: possiblyAchievable Für ein Atom a und eine Operatorenmenge O gilt $possiblyAchievable(a, O)$ genau dann wenn

$$\exists o \in O : a \in add(o) \wedge \forall b \in pre(o) : \exists q \in O : b \in add(q)$$

Nach dem parsen der gegroudeten Problem- und Domänenbeschreibung haben wir also dann diese Heuristik benutzt, um eine geordnete Partition der Zielmenge zu berechnen, und diese dann als Input für unsere Planungsschleife zu verwenden. Vorher wird dem User noch die Möglichkeit gegeben, die berechneten paarweisen Abhängigkeiten der Zielatome zu verifizieren.

4.3. Implementierung einer Planungsschleife

Als Input für die Planungsschleife haben wir unter anderem eine geordnete Partition der Ziemenge, welche je nach Problem eher fein oder eher grob sein kann. Die Ordnung dieser Partition entspricht der Anzahl an Schleifendurchläufen. Innerhalb des Rumpfes wird zunächst eine neue Problemdatei erzeugt. Diese Problemdatei beschreibt eine Etappe des ursprünglichen Problems. Beim ersten Schleifendurchlauf wird der Startzustand übernommen und als Zielzustand das erste Teilziel in die Problem-pddl-Datei geschrieben. Ansonsten wird der im vorherigen Durchlauf berechnete Zwischenzustand sowie das entsprechende neue Teilziel hineingeschrieben. Dann wird der Planer FF auf dieser Problemdatei aufgerufen und der erzeugte partielle Plan in eine Datei gespeichert. Dieser partielle Plan wird dann durch den Validator ausgeführt und der erreichte Zielzustand als neuer Startzustand für den nächsten Schleifendurchlauf gespeichert.

Somit wird in der Schleife nach und nach der ganze Plan berechnet.

Kapitel 5.

Implementierung zulässiger Heuristiken für STRIPS

Roman Klinger, Kenneth Kahl

5.1. Einleitung und Ausgangsposition

Den Ausgangspunkt unserer Arbeiten bildet das bestehende Planungssystem MIPS [10]. Hier sind die Suchalgorithmen Iterative Deepening A* (IDA*), A* sowie Hill-Climbing implementiert. Als Heuristiken stehen die Relaxed Planning Heuristic (FF) und die Pattern Database Heuristik (PDB) zur Verfügung. Die bisherige Implementierung lässt allerdings keine optimale Planung zu. Hier sehen wir unseren Ansatzpunkt. Dazu orientieren wir uns am bestehenden Planungssystem HSP* [18] und seinen Algorithmen sowie der Idee einer Patterneinteilung (siehe auch [5]), die optimales Planen möglich macht. In diesem Artikel stellen wir nun unsere von HSP* adaptierten Techniken MaxAtom (sowie die nicht zulässige Variante AddAtom) und MaxPair in einer Implementierung zur Verwendung in einer Vorwärtssuche im Zustandsraum vor. Darauf folgt die Darstellung von Ideen, welche Effizienzsteigerungen durch Preprocessing möglich machen. Unser nächster Ansatz ist die Erweiterung der bestehenden PDB Heuristik zu einer zulässigen Heuristik.

5.2. Architektur des bestehenden Systems

Der Aufruf des Systems findet in der Hauptklasse `mips` statt. Von hier aus werden die Aufrufparameter geparst (`option`) und die grundlegenden Datenstrukturen angelegt (`domain`). Durch die Repräsentationen des geparsten Planungsproblems in `factMap` wid die Struktur zur Berechnung der Heuristiken sowie zur Suche zur Verfügung gestellt. Hier setzen wir an den Oberklassen `heuristic` und `search` an. Eine graphische Darstellung findet sich in Abbildung 5.1.

5.3. Implementierung HSP

5.3.1. Max-Atom-Heuristik (und Add-Atom-Heuristik)

Die Idee ist, zu jeder Expansion eines Zustands (im folgenden *Akt* genannt) die Entfernung zum Zielzustand abzuschätzen. Hierzu schätzen wir erst einmal die Kosten $g_s(p)$ von *Akt* zu allen Atomen p. Dies geschieht durch Initialisierung der Atome in $p \in Akt$ mit $g_s(p) = 0$ sowie $g_s(p) = \infty$ für $p \notin Akt$. Die Berechnungen für die Atome finden dann statt durch

$$g_s(p) := \min_{a \in O(p)} [g_s(p), 1 + g_s(Prec(a))].$$

$O(p)$ sind hierbei die Operatoren, welche p als Add-Effekt haben. Die Berechnung entspricht also einer relaxierten Suche, bei der die Delete-Effekte nicht betrachtet werden. Hierdurch wird also der Zustand *Akt* erweitert.

Der Wert einer Menge ergibt sich durch $h(M) = \max_{p \in M} g_s(p)$. Die Heuristik MaxAtom ergibt sich dann durch die höchsten geschätzten Atomkosten der Atome im Zielzustand:

$$h(Akt) = \max_{p \in Ziel} g_s(p).$$

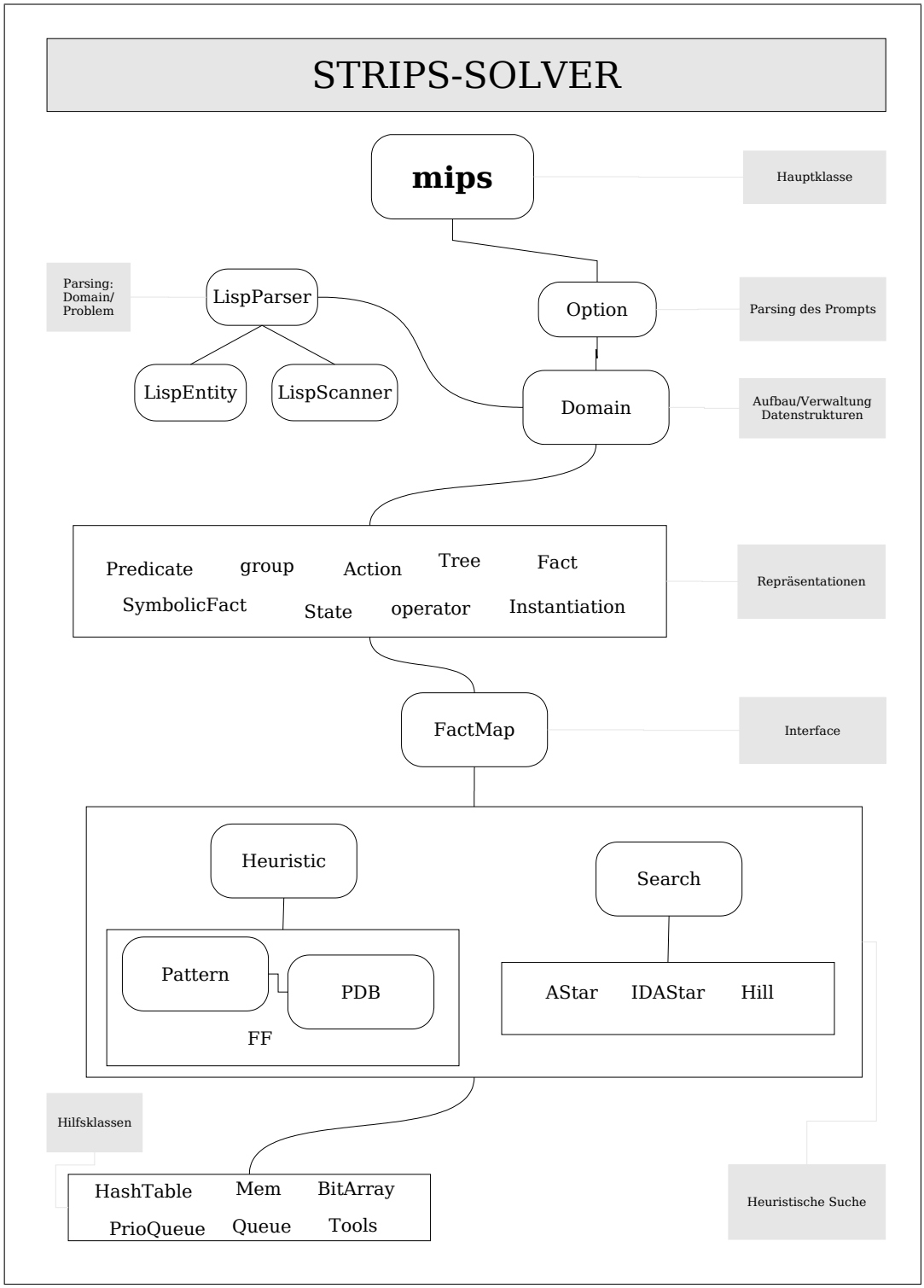


Abbildung 5.1.: Überblick über den MIPS-Planer

```
Uebergabe: Zustand S, Zielzustand Z;
Initialisere g-Werte-Array;
while(S hat sich geändert) {
   fuer alle auf S anwendbaren Operatoren op {
      max_Prec = der Wert der teuersten Vorbedingung von op
        fuer alle add-Effects a von op
           aktualisiere g(a) wenn 1+max_Prec kleiner ist als g(a);
        wende Operator (ohne DelListe) auf S an;
   }
}
return hoechsten g-Wert der Atome in z;
```

Abbildung 5.2.: Unsere Implementierung der HSP-Max-Atom-Heuristik

Diese Heuristik ist offensichtlich zulässig. Eine kleine Variation hiervon ist die Add-Atom-Heuristik, die den Wert mit

$$h(Akt) = \sum_{p \in Ziel} g_s(p)$$

berechnet, was zwar informativer, aber nicht zulässig ist (Der Wert einer Menge ergibt sich hier also durch $h(M) = \sum_{p \in M} g_s(p)$). Also bringt uns dies unserem Ziel nicht näher. Für detailliertere Betrachtung sei [18] empfohlen.

Eine Darstellung unserer Implementierung von HSP-Max-Atom findet sich in Abbildung 5.2. In der Methode wird der geschätzte Abstand $g_s(S)$ von Zustand S zu Zustand Z berechnet. Wir benutzen ein Array in dem die geschätzten Entfernungen der Atome zum Zustand Z gespeichert werden. Dieses wird zu Anfang mit 0 für alle Atome im Zustand S und mit ∞ sonst initialisiert. Solange sich Zustand S noch ändert, wird die while-Schleife durchlaufen. Nun wird für alle Operatoren, welche auf S anwendbar sind, der Wert der teuersten Vorbedingung `max_Prec` bestimmt. Diese Werte können nicht ∞ betragen, da der Operator sonst nicht anwendbar wäre (wie am Schleifendurchlauf leicht erkennbar ist). Nun werden die Atome, die sich im Add-Effekt des Operators befinden aktualisiert, wenn der neue Wert, welcher sich eben aus der teuersten Vorbedingung zuzüglich 1 (für den aktuellen Operator) ergibt, kleiner als der alte Wert ist. Zuletzt wird der Operator ohne Berücksichtigung der Delete-Liste auf den Zustand angewendet.

Nachdem nun keine Atome mehr zum Zustand S hinzugefügt werden beinhaltet das Array g_s die Werte aller Atome, die erreichbar sind. Für alle anderen Atome gilt $g_s(p) = \infty$.

Der heuristische Wert wird nun durch den höchsten $g_s(p)$-Wert mit $p \in Z$ bestimmt und zurückgegeben.

5.3.2. Max Pair

In der Max-Atom-Heuristik wird zur Abschätzung der Kosten zum Zielzustand nur ein Atom genutzt. Add-Atom nutzt alle Atome, ist aber nicht mehr zulässig. Die Max-Pair-Heuristik versucht, eine gesteigerte Informativität ohne Verlust der Zulässigkeit zu erreichen. Hierzu

werden nicht nur einzelne Atomkosten errechnet, sondern die Kosten von Atompaaren:

$$g_s^2(\{p,q\}) = \min\{g_s^2(p\&q), g_s^2(p|q), g_s^2(q|p)\}$$

wobei

$$g_s^2(p\&q) = \min_{a \in O(p\&q)}[1 + g_s^2(Prec(a))]$$

$$g_s^2(p|q) = \min_{a \in O(p|q)}[1 + g_s^2(Prec(a) \cup \{p\})]$$

$$g_s^2(q|p) = \min_{a \in O(q|p)}[1 + g_s^2(Prec(a) \cup \{q\})].$$

Hierbei bedeutet $p\&q$, dass p und q in der Addliste des Operators sind und $p|q$, dass p in der Addliste und q weder in Add- noch Deleteliste ist.

Der Fall, dass nur ein einzelnes Atom betrachtet wird sowie die Initialisierung sind äquivalent zur Max-Atom-Heuristik. Für weitere Betrachtungen sei wiederum auf [18] verwiesen.

Eine Darstellung unserer Implementierung von HSP-Max-Pair findet sich in Abbildung 5.3.

Wiederum berechnen wir den geschätzten Abstand von Zustand S zu Zustand Z. Analog zu Max-Atom wird ein Array für alle Faktpaare mit $g(p,q) = 0$ für $p, q \in S$ und $g(p,q) = \infty$ ansonsten initialisiert (Zeile 1). Nun wird für jeden anwendbaren Operator der Wert des teuersten Vorbedingungspaares bestimmt und in `max1` gespeichert (Zeilen 8-9). Dieser Wert wird nun zuzüglich 1 mit allen Werten der Paare der Add-Liste des Operators verglichen (Zeile 13) und, wenn er kleiner ist als der alte Wert für das Paar neu gesetzt (Zeile 14). Für alle Paare von Fakten (p,q), bei denen p hinzugefügt wird und q nicht in der Add-Liste vorhanden ist, wird mit Hilfe von `max1` der Wert des teuersten Faktpaares, welche q enthalten aktualisiert (Zeilen 16-22). Entsprechend werden die Werte des Arrays angepasst (Zeilen 21-22).

Wie auch bei Max-Atom wird nun der Operator angewendet. Wenn die While-Schleife terminiert, wird das teuerste Faktpaar, welches aus den Atomen im Zustand Z entsteht, bestimmt und der entsprechende Wert als Heuristik zurückgegeben (Zeile 27).

5.3.3. Ansätze zur Zielzustandserweiterung für Preprocessing

In den bis hier dargestellten Ansätzen wird das vollständige Array, welches Abstände von Atomen zum Zielzustand schätzt, bei der Expansion eines Zustands neu berechnet. Würden im Vorfeld die Entfernungen aller Atome zum Zielzustand berechnet werden, könnte man die Entfernung von jedem Zustand sehr schnell durch Bestimmung des Maximums der Werte der zum Zustand gehörenden Atome errechnen. Ein anderer Ansatz wäre die Bestimmung der Entfernung der Atome vom Startzustand, dann müßte eine Rückwärtssuche durchgeführt werden.

Unsere Versuche stützen sich auf den ersten Ansatz.

Hierzu müssen neue Operatoren definiert werden, die eine Rückwärtssuche erlauben. Diese ergeben sich für jeden Operator aus:

$$P_{neu} := A_{alt} \cup P_{alt} \backslash D_{alt}$$

```
Gegeben: Zustand S, Zielzustand Z

 1 Initialisiere g-Array für alle Faktpaare mit 0 wenn in S,
                                              sonst unendlich;
 2 solange(S aendert sich)
 3     für jeden auf S anwendbaren Operator op
 4     {
 5             int max1 = 0;
 6             int max2 = 0;
 7             // wenn p&q: G^2(Prec(op)) vorberechnen
 8             für alle Paare (p,q) in prec(op)
 9                 bestimme max1 von gs(p,q)
10             für alle Fakten q
11                 wenn q in addliste(op)
12                     für alle p in addliste(op)
13                         wenn gs(p,q) > max1 + 1
14                             setze gs(p,q) = max1 + 1
15                 sonst:
16                     max2 = max1;
17                     // G^2(Prec(op) u {q}) (nur noch prec mit q
                                                  kombinieren)
18                         für jedes Paar (p in prec(op),q)
19                             bestimme max2 von (gs(p,q),max2)
20                     für alle Fakten in addliste(op)
21                         wenn gs(p,q) > max2 + 1
22                             setze gs(p,q) = max2 + 1
23     }
24     Anwendung von op;
25 Ende Solange;
26
27 Gebe höchsten Wert gs(p,q) für Paar (p,q) aus Z als
                                    heuristischen Wert zurück.
```

Abbildung 5.3.: Unsere Implementierung der HSP-Max-Pair-Heuristik

$$D_{neu} := A_{alt}$$
$$A_{neu} := D_{alt}$$

Hierbei stellt A die Add-Liste des Operators, D die Delete-Liste und P die Vorbedingungsliste dar. Weiterhin wird davon ausgegangen, dass $D \subseteq P$ gilt. Allerdings wird die Rückwärtssuche relaxiert durchgeführt, so dass auch hier D_{neu} ignoriert wird.

Unsere relaxierte Rückwärtssuche startet also bei dem Zielzustand. Dieser ist aber üblicherweise nicht vollständig definiert. Atome, welche im Zielzustand gelten könnten, werden so bei der Initialisierung mit Unendlich abgeschätzt, wobei der eigentlich korrekte Wert 0 sein müsste. Dieses Problem lässt sich umgehen, indem der Zielzustand um alle Atome, welche gemeinsam mit dem definierten Zielzustand gelten können, erweitert wird. Erweitert wird also mit allen Atomen, die nicht mutex[1] zu allen Atomen im Zielzustand sind. Diese Aufgabe lässt sich mit Hilfe eines variierten Graphplan Algorithmus [3] lösen. Hierzu implementieren wir einen Algorithmus von Refanidis, Vlahavas und Tsoukalas [37]. Leider werden durch diesen alle Atome hinzugefügt. Ob ein algorithmischer oder implementatorischer Fehler vorliegt, ließ sich nicht endgültig klären.

Ein weiterer Ansatz, welchen wir aktuell verfolgen ist daher die relaxierte Vorwärtssuche zur Bestimmung des heuristischen Werts sowie eine darauf folgende systematische Rückwärtssuche im Zustandsraum, was ebenfalls das Preprocessing ermöglichen wird.

5.4. Implementierung Pattern Databases

Die grundsätzliche Idee der Berechnung einer Heuristik mit Pattern Databases ist, den ursprünglichen Zustandsraum zu reduzieren. Auf diesem wird dann eine vollständige Rückwärtssuche zur Bestimmung der Entfernungen aller Zustände zum Zielzustand durchgeführt. Zur detailreicheren Darstellung sei hier auf [5] verwiesen.

Dem MIPS-System werden sogenannte Gruppen bereits durch die Eingabekodierung mit übergeben. Diese bestehen aus je einer Variable, welche eine Menge von Atomen impliziert, die alle mutex zueinander sind. So kann man sich zum Beispiel in Blocksworld die Variable $on(X, a)$ (wobei X für alle Blocks steht) vorstellen, aus der sich dann die Atome $on(d, a)$, $on(c, a)$, $on(b, a)$ und $on(a, a)$ ergeben. Da natürlich nur ein Block gleichzeitig auf a liegen kann und dies durch die entsprechende Operatorendefinition auch gewährleistet ist, sind diese Atome mutex.

Nun müssen diese Gruppen in sogenannte Patterns (Abstraktionen) eingeteilt werden, wobei jede Pattern eine Relaxierung des Zustandsraums darstellt. Diese wird dadurch erreicht, dass jeder dieser abstrahierten Zustandsräume nur noch eine Teilmenge der ursprünglichen Atome enthält, nämlich gerade die, welche den Gruppen in dem zugehörigen Pattern entsprechen. Auf diese Weise werden den eigentlichen Zuständen reduzierte Zustände zugeordnet. Selbiges geschieht mit den Operatoren, bei denen die Vorbedingungen, Add-Effekte und Delete-Effekte ebenfalls nur noch Atome aus der Pattern enthalten.

Die Einteilung in die Patterns ist ein schwieriges Problem, welches die Qualität des heuristischen Wertes stark beeinflussen kann.

[1] Kein Paar von Atomen kann gleichzeitig gelten. Siehe auch [3]

Eine Pattern Database zu einem Pattern besteht nun aus den abstrahierten Zuständen mit den jeweiligen Entfernungen zum Zielzustand, die dann untere Schranken für die realen Entfernungen darstellen. MIPS verwendet zur Bestimmung dieser Entfernungen eine Breitensuche, welche durch die Reduzierung der Atommenge effizient möglich ist.

Diese startet beim Zielzustand der jeweiligen Abstraktion und verwendet die oben bereits im Kapitel 5.3.3 eingeführten inversen Operatoren. Die Zustände bekommen hierbei die Pfadlängen des in der Breitensuche erstellten Baums zugewiesen. Die Kanten entsprechen hierbei den Operatoren.

In jeder Abstraktion i gibt es nun zu jedem Zustand s geschätzte Kosten c_i. Der heuristische Wert $h(s)$ ergibt sich nun durch Aufaddieren der einzelnen Werte:

$$h(s) = \sum_i c_i(s).$$

Im Allgemeinen kann nicht davon ausgegangen werden, dass jeder Operator nur in einer Pattern vorkommt[2]. Dadurch werden Operatoren mehrfach gezählt. Somit ist die Heuristik nicht zulässig. Dies ließe sich umgehen, in dem die Patterneinteilung so vorgenommen wird, dass jeder Operator tatsächlich nur in einer Pattern gilt. Eine andere Möglichkeit, für die wir uns entschieden haben, ist, bei Durchführen der Breitensuche einen Operator (und damit entsprechende Kante) nicht zu den Kosten zum Zustand hinzuzuzählen, wenn der Operator schon bei dem Aufbau einer anderen Pattern Database vorkam.

Wenn auf diesen Wegen eine zulässige Heuristik erreicht wird, spricht man von *Additivität* [17]. Diese ist durch eben beschriebenes Verfahren erreicht, so dass das nächste Ziel die Optimierung der Patterneinteilung ist. Diese wird im Moment durch ein Greedy-Bin-Packing durchgeführt.

5.4.1. Verbesserung der Patterneinteilung durch Benutzerinteraktion

In [26] formulieren die Autoren verschiedene Vor- und Nachteile unterschiedlicher Patterneinteilungen. Hierbei wird festgestellt, dass diese Unterscheidungen auch problemabhängig sind. Daher geben wir dem Benutzer unseres Planungssystems die Möglichkeit, domänen- und problemabhängig direkt die Patterneinteilung zu manipulieren. Hierbei entsteht der Vorteil, dass auch Expertenwissen direkt in den Planungsprozess einfliessen kann. Ebenso kann der Benutzer auch sein Wissen über die entsprechende Domäne durch Experimente mit der Einteilung vergrößern.

Technologie und Modifikationsmöglichkeiten

Es ist die Entscheidung auf die Verwendung von Standardtechnologien gefallen, da zum einen der Implementierungsaufwand stark reduziert werden kann, zum anderen aber auch die Erweiterbarkeit und Kommunikation mit weiterer Software gewährleistet ist. So verwenden wir zur Speicherung der Gruppeneinteilung in Patterns die Extensible Markup Language (XML). In Abbildung 5.4 ist eine solche von uns verwendete XML-Datei dargestellt. Das Attribut

[2] Ein Operator kommt nicht in einer Pattern vor, wenn er durch Reduktion der Atome keinen Add-Effekt hat.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE patternarrangement SYSTEM ".transfer.dtd">
<patternarrangement>
   <pattern>
      <group name="(on-a-a) (on-b-a) (on-c-a) (on-d-a)
         (clear-a) none" number="1"/>
   </pattern>
   <pattern>
      <group name="(on-a-b) (on-b-b) (on-c-b) (on-d-b)
         (clear-b) none" number="2"/>
   </pattern>
   <pattern>
      <group name="(on-a-c) (on-b-c) (on-c-c) (on-d-c)
         (clear-c) none" number="3"/>
   </pattern>
   <unused>
      <group name="(handempty) (holding-a) (holding-b)
         (holding-c) (holding-d)" number="0"/>
      <group name="(on-a-d) (on-b-d) (on-c-d) (on-d-d)
         (clear-d) none" number="4"/>
      <group name="(ontable-a) none" number="5"/>
      <group name="(ontable-b) none" number="6"/>
      <group name="(ontable-c) none" number="7"/>
      <group name="(ontable-d) none" number="8"/>
   </unused>
</patternarrangement>
```

Abbildung 5.4.: XML-Datei zur Speicherung einer Gruppeneinteilung in Pattern

`name` des Elements `group` beinhaltet eine Auflistung aller Instanziierungen der entsprechenden Variable. Diese wird direkt mit der Problemstellung dem Planungssystem übergeben. Für unser System relevant ist aber nur das Attribut `number`, da hiermit die entsprechende Gruppe eindeutig bestimmt ist. Der Name ist für den Benutzer aber natürlich nicht verzichtbar.

Zur komfortablen Editierung dieser XML-Datei greifen wir auf den Open-Source-Editor Pollo[3] zurück. Dieser lässt sich leicht konfigurieren und mit Hilfe einer XML-Schema-Datei auch auf unser verwendetes XML-Format spezialisieren. So stellt der Editor kontextabhängig entsprechend den Restriktionen des Schemas erlaubte Veränderungen zur Verfügung. So darf man zum Beispiel, wenn man eine Gruppe aktiviert hat, eine Pattern einfügen, aber keine Gruppe. Eine Abbildung von Pollo mit der in Abbildung 5.4 dargestellten XML-Datei findet sich in Abbildung 5.5.

Die Manipulation der Patterneinteilung verläuft auf Basis einer automatischen Einteilung,

[3] `http://pollo.sourceforge.net/`

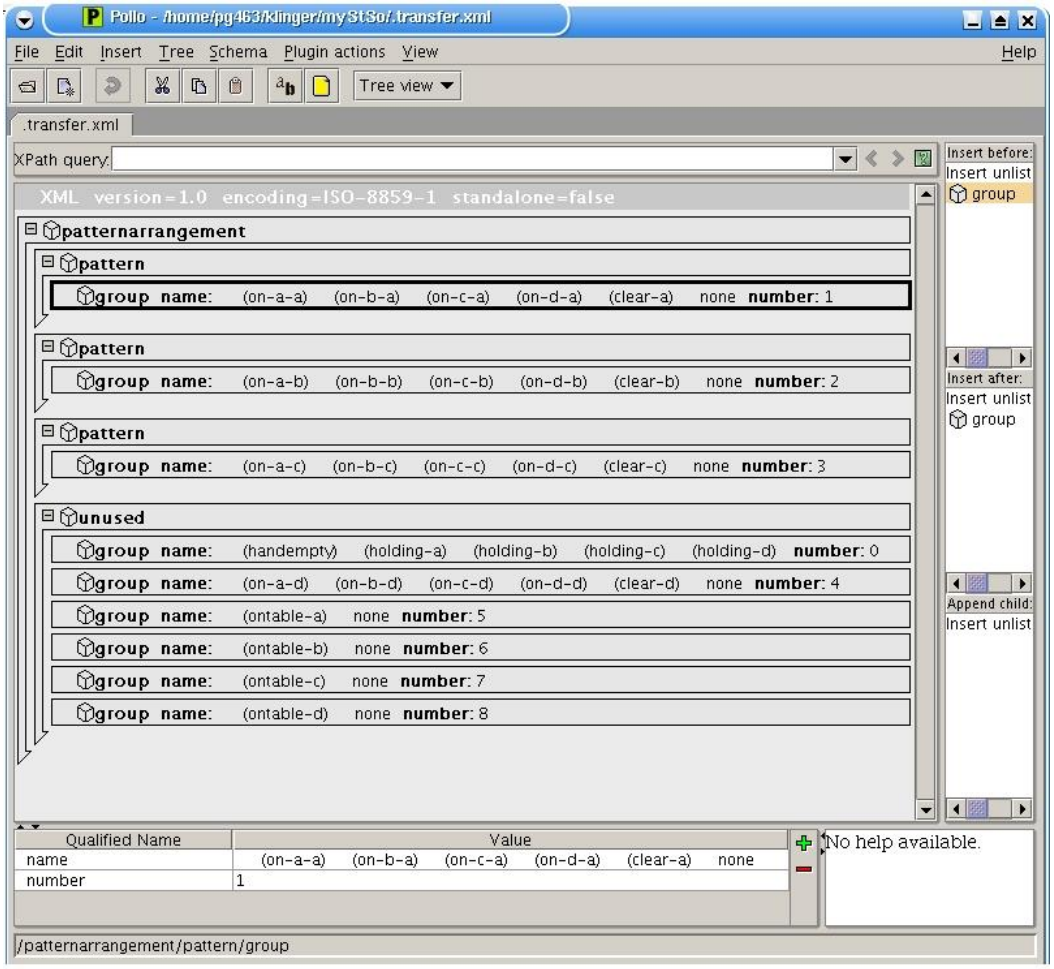


Abbildung 5.5.: Darstellung der XML-Datei aus Abbildung 5.4 in Pollo

wie zum Beispiel das bereits genannte Greedy-Bin-Packing. Diese Verteilung der Gruppen, also Variablen auf unterschiedliche Patterns, werden in einer XML-Datei gespeichert.

Nun kann der Benutzer Veränderungen vornehmen und diese zum einen direkt in die Datei, welche von MIPS gelesen wird, speichern. Zum anderen besteht auch die Möglichkeit, die Editierungen durch Sichern in eine andere Datei für spätere Planungsläufe zur Verfügung zu halten.

Nach Beenden des Editors Pollo liest MIPS die Veränderungen aus der XML-Datei ein und plant mit Hilfe dieser.

Des Weiteren ist die Möglichkeit implementiert, bei einer Einbettung des Planungssystems in eine integrierte Planungsumgebung (wie die von der Projektgruppe entwickelte ModPlan-Workbench) die Laufzeitzuteilung extern zu verwalten. Der Ablauf hierzu ist in Abbildung 5.6 dargestellt.

Als Erstes startet die Workbench und liest dabei den Pfad zum Editor, in unserem Fall Pollo (was aber beliebig durch einen anderen XML-Editor ersetzt werden kann). Die Steuerung der Programme findet durch die Datei `.temp` statt. In Folge startet MIPS und verändert den Wert in `.temp` nach Speichern der zu verändernden Patterneinteilung in die XML-Datei und teilt damit der Workbench seinen Zustand mit. Nun wird der Editor gestartet wobei nach dessen Beendigung wiederum der Inhalt von `.temp` verändert wird, wodurch nun MIPS die Planung fortsetzt. Durch die externe Steuerung lässt sich der Ablauf natürlich beliebig anderweitig vollziehen, so dass zum Beispiel auch externe Optimierverfahren der Patterneinteilung in Kombination mit unserer Methode eingesetzt werden könnten. In dem Fall würde mips erst nach Beendigung der unterschiedlichen Programme fortgesetzt werden.

Zur Unterstützung des Benutzers werden die Häufigkeiten der heuristischen Werte, welche bei der rückwärtigen Tiefensuche in den unterschiedlichen Patterns auftreten, in einer Datei gespeichert. Diese Datei kann von dem Visualisierungsprogramm Vega ("Visualization environment for geometric algorithms" von Christoph A. Hipke) als Histogramm dargestellt werden.

In Abbildung 5.7 sind die Histogramme von drei unterschiedlichen Patterneinteilungen zu finden. Die großen Abbildungen links und rechts basieren beide auf je einer Pattern mit zwei Gruppen gleicher Größe. Dennoch ist sofort erkennbar, dass sich die Verteilung der heuristischen Werte unterscheidet. Im Hauptdiagramm (links) kommen einige Werte (8 bis 11) gleich häufig vor und der höchste heuristische Wert ist 13. Die rechte Abbildung zeigt eine weniger gleichmäßige Verteilung, insbesondere aber weniger hohe Werte, was auf eine geringere Informativität schliessen lässt.

Die kleine Abbildung zeigt eine Pattern welche nur aus einer Gruppe besteht. Hier ist zu bemerken, dass der Zustandsraum der Breitensuche offensichtlich deutlich kleiner ist als bei den anderen beiden Patterns.

Ebenso trifft die Erwartung ein, dass die Anzahl der möglichen Zustande bei zusammenfügen der beiden anderen Patterns regelrecht explodiert, so dass gleiche heuristische Werte bis zu 1100 mal auftreten. Die Qualität steigt zwar nicht proportional dazu, aber wir erreichen einen maximalen heuristischen Wert von 23.

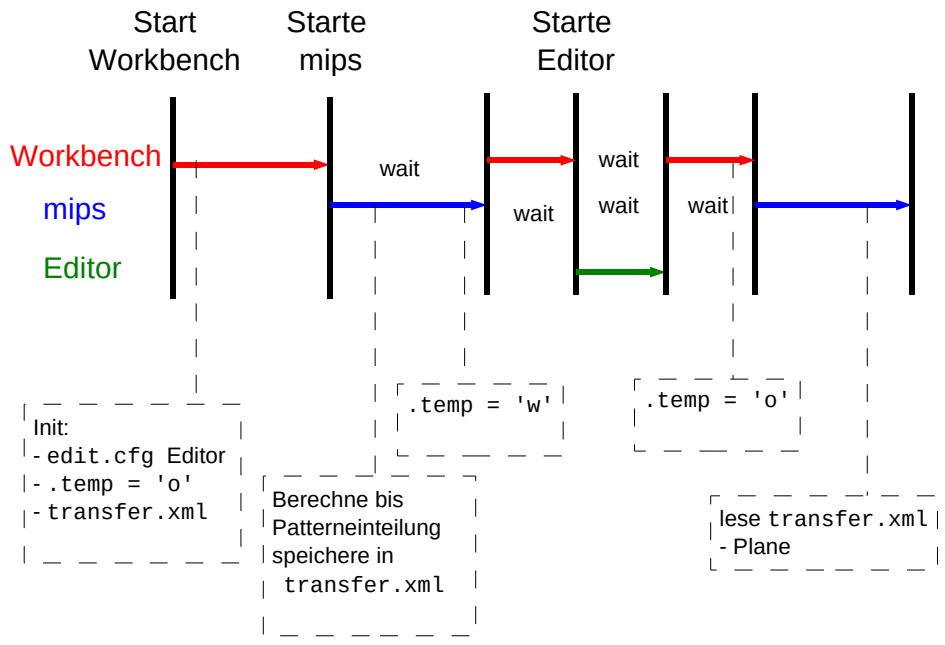


Abbildung 5.6.: Ablauf bei externer Laufzeitsteuerung

5.4.2. Verbessung der Patterneinteilung durch genetische Algorithmen

Motivation

Trotz zahlreicher Veröffentlichungen zu dem Themengebiet der Verbesserung der Patterneinteilung sind hierzu nur wenige aussagekräftige Heuristiken entstanden. So ist das Problem nach wie vor als schwierig einzustufen, insbesondere, da die Qualität der Patterneinteilung auch domänenabhängig zu sein scheint.

Gerade bei Problemen mit unbekannter Struktur bieten sich naturnahe Optimierverfahren wie Evolutionsstrategien, genetische Algorithmen oder auch Partikelschwarmoptimierung an.

Aufgrund der diskreten Struktur des Problems entscheiden wir uns gegen die Evolutionsstrategien. Unsere Ansätze folgen daher den Ideen der genetischen Algorithmen.

Auf diesem Wege ist auch zu erwarten, weitere Erkenntnisse über wichtige Eigenschaften der Patterneinteilungen zu erlangen.

Überblick genetische Algorithmen

Das Optimierparadigma der genetischen Algorithmen wurde von John Holland und David Goldberg entwickelt (siehe dazu [15]). Der grundsätzliche Ablauf ist in Abbildung 5.8 dargestellt.

In den Zeilen 0 und 1 findet die Initialisierung eines Zählers sowie der Population, die aus n Individuen besteht, statt. Ein solches Individuum ist im klassischen Fall ein Bitstring. Der

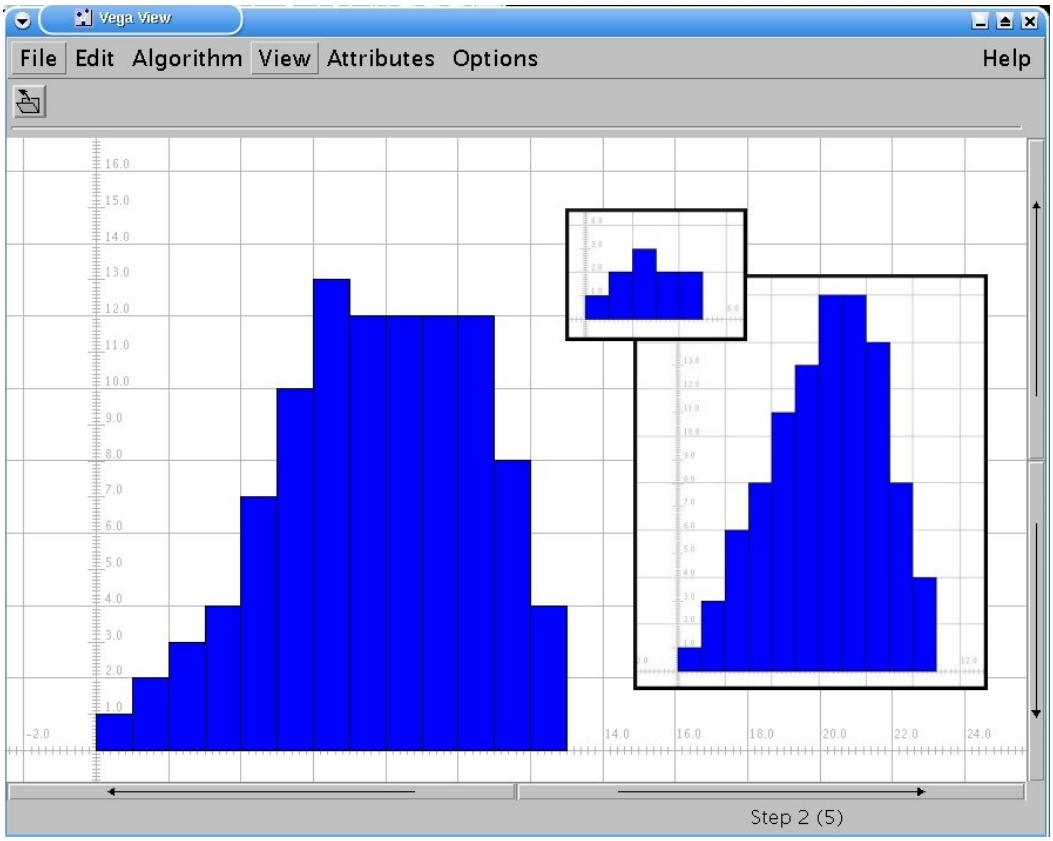


Abbildung 5.7.: Histogramm zur Darstellung der Häufigkeiten von heuristischen Werten in einer Pattern

```
0   t := 0;
1   initialisiere P^(t) = {I_1, ..., I_n};
2   bewerte P^(t);
3   while (κ(P^(t)) ≠ 1) do
4     P'^(t) := Rekombination(P^(t));
5     P''^(t) := Mutation(P'^(t));
6     bewerte P''^(t);
7     P^(t+1) := Selektion(P''^(t));
8     t := t + 1
9   od;
```

Abbildung 5.8.: Ablauf eines genetischen Algorithmus (nach [20])

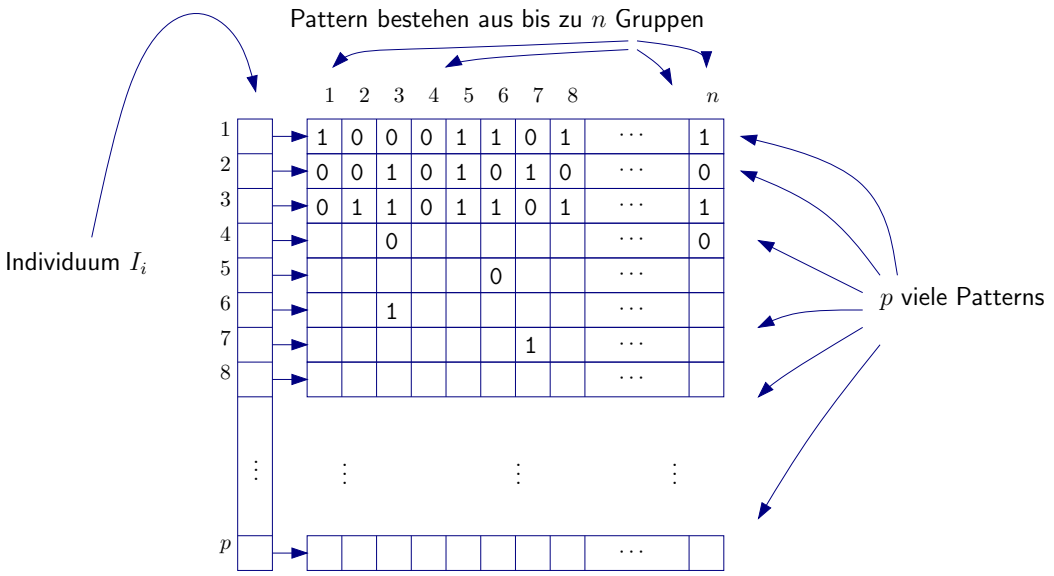


Abbildung 5.9.: Repräsentation eines Individuums

Evaluationsschritt zur Berechnung der Fitness in Zeile 3 ist notwendig, damit in Zeile 4 die Auswahl der Individuen zur Rekombination fitnessproportional stattfinden kann. Bei der Rekombination werden die Informationen zweier ("Eltern"-)Individuen zur Bildung von neuen ("Kind"-)Individuen nach bestimmten Regeln verknüpft. Mit einer geringen Wahrscheinlichkeit werden einzelne Stellen dieser aus der Rekombination erstellten Population verändert (Zeile 5). Auf Basis der in Zeile 6 durchgeführten Evaluation werden in Zeile 7 Individuen fitnessproportional ausgewählt um die Population auf die Ursprungsgröße zu bringen. Nach Hochsetzen des Zählers ist der nächste Schritt wiederum die Rekombination. Diese Schleife wird durch Erreichen eines Abbruchkriteriums κ beendet. Dies ist üblicherweise ein hinreichend guter Fitnesswert.

Wir nehmen an diesem grundsätzlichen Schema kleinere Änderungen vor, wie in den folgenden Abschnitten beschrieben wird.

Repräsentation

In der Implementation des Planungssystems werden die Patterneinteilungen folgendermaßen dargestellt. Es existiert eine zweidimensionale bool'sche Tabelle der Größe $g \times g$, wobei g die Anzahl der Gruppen sei. In der ersten Dimension werden die Gruppen aufgetragen, in der zweiten Dimension die Patterns.

Da bei genetischen Algorithmen die Notwendigkeit besteht, aus einem Individuum eine vollständige Lösung zu interpretieren, bietet es sich an, genau diese Tabelle zur Repräsentation unserer Individuen zu nutzen.

In Abbildung 5.9 findet sich ein solches Individuum. Hierbei ist $n = g$ und $p \leq g$. So sind zum Beispiel in Pattern 1 die Gruppen 1,5,6,8 und n und in Pattern 2 die Gruppen 3,5 und 7 eingefügt.

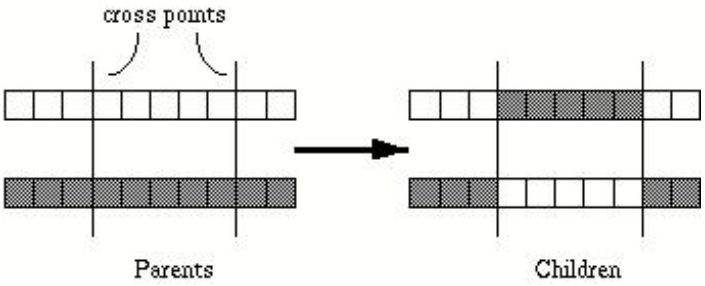


Abbildung 5.10.: 2-Punkt-Crossover [28]

Initialisierung

Bei der Initialisierung wird die Größe eines jeden Individuums uniform zufällig in dem Intervall $[1, g]$ gewählt. Die Anzahl der Individuen, also die Populationsgröße wird vom Benutzer vorgegeben.

In einer Variante haben wir getestet, in wieweit eine rein zufällige Belegung der Individuen zum Ziel führt. Hierbei hat sich jedoch herausgestellt, daß der Aufwand eine akzeptable Verteilung zu erreichen deutlich höher ist als der des Greedy-Bin-Packings, jedoch keine bemerkenswerte Verbesserung der Suche herbeiführt.

Daher initialisieren wir die Individuen mit Hilfe des bereits implementierten Greedy-Bin-Packing Algorithmus. Um zu vermeiden, daß aus der Systematik dieses Algorithmus folgt, daß alle Individuen der Startpopulation identisch sind, ist eine Variation der bisherigen Implementation notwendig. So werden die Gruppen dem Algorithmus in einer immer wieder neu zufällig erstellten Reihenfolge präsentiert. Dies erzeugt mehrere, ähnlich gute Gruppenverteilungen auf die Patterns, welche jedoch unterschiedlich sind. So ist dies eine gute Ausgangsposition für die Optimierung mit dem genetischen Algorithmus.

Rekombination

Die Motivation der Rekombination zweier sogenannter Elternindividuen ist die Hoffnung, die guten Eigenschaften des einen mit den guten Eigenschaften des anderen zu kombinieren. Die einfachste Technik ist das *n-Point-Crossover* welches anhand zweier eindimensionaler Individuen in Abbildung 5.10 dargestellt ist.

Sinnvoll in unserer Repräsentation ist die Verwendung von 1- oder 2-Point-Crossover in der Dimension der Pattern. Dies bedeutet, dass die Elternindividuen einen Teil ihrer Patterns austauschen. Hierbei entstehen zwei Individuen, wenn die Eltern eine unterschiedliche Anzahl Patterns beinhalten, entsteht ein längeres und ein kürzeres Kind.

Mutation

Bei der Mutation werden mit einer üblicherweise geringen Wahrscheinlichkeit die einzelnen Bits in der das Individiuum repräsentierenden Tabelle geflippt. Dieser Operator stellt eine wichtige Funktion dar, denn durch ihn werden die Gruppen zu Patterns hinzugefügt oder ihnen entzogen.

Weiterhin nutzen wir einen Mutationsoperator, welcher Patterns entfernt oder eine zufällig neu erzeugte Pattern einfügt.

Selektion

Mit Hilfe der Selektion wird die durch die Rekombination möglicherweise (je nach Rekombinationsoperator) vergrößerte Population in Abhängigkeit von der Fitness der Individuen auf die ursprüngliche Größe gebracht. Dazu wird die auf eine Summe von 1 skalierte Fitness zur Bildung einer Wahrscheinlichkeitsverteilung genutzt. Entsprechend dieser werden dann die Individuen mit höherer Fitness mit höherer Wahrscheinlichkeit für die Folgepopulation gewählt.

Zielfunktion

Die Zielfunktion stellt eine zentrale Rolle bei den genetischen Algorithmen dar. Aus ihr wird direkt die Fitnessfunktion zur Berechnung der Fitness der Individuen bestimmt. Die Erstellung der Zielfunktion ist oft ein schwieriges Problem, so auch in unserem Fall, gerade, da die Bedingungen, wann eine Patterneinteilung als 'gut' einzuschätzen ist, nahezu unbekannt sind.

Wie bereits erwähnt wird bei der Erstellung der Pattern Data Bases eine rückwärtige Breitensuche durchgeführt. Je höher die hierbei erreichten heuristischen Werte sind, als desto besser sind sie zu beurteilen. Dies gilt, da eine Suche auf dem vollständigen Suchraum (welcher ja durch die Suche auf dem durch die Pattern Data Base induzierten Suchraum verkleinert wird) obere Schranken für die heuristischen Werte von jedem Zustand liefert.

Wir bilden daher die heuristischen Werte und berechnen den Durchschnitt dieser für jede Pattern. Der Durchschnitt dieser Werte für eine vollständige Patterneinteilung gibt uns den zugehörigen Zielfunktionswert.

Implementierungsansatz: GALib

In einem ersten Implementierungsansatz verfolgten wir eine vollständig eigenständige Implementierung. Diesen Ansatz verwarfen wir schließlich zu Gunsten der Nutzung einer Library, da wir uns hiervon eine höhere Effizienz von möglicherweise notwendigen Experimenten mit unterschiedlichen Operatoren erhoffen.

Die Entscheidung ist auf die GAlib[4] [39] gefallen. Besonders hervorzuheben ist bei dieser erst einmal der technische Vorteil, daß sie ohne Änderungen unter verschiedenen Betriebssystemen läuft. Angenehm ist auch, daß bereits eine Datenstruktur zur Verfügung steht, welche genau unseren Ansprüchen genügt. So ist es ausreichend, eine Zielfunktion zu implementieren. Eine Fitnessfunktion wird automatisch, wenn gewünscht, mit dem Verfahren des Fitness Scalings errechnet. Die Operatoren Rekombination, Mutation und Selektion sind auch nur noch entsprechend zu konfigurieren. Bemerkenswert ist, daß diese Konfigurationen auch über eine externe Datei möglich sind und nicht zwingend direkt im Quellcode stattfinden muß.

Notwendig für spätere Experimente ist aber nicht nur die Flexibiliät des Systems, sowie gewisse mitgebrachte Operatoren, sondern auch die einfache Erweiterbarkeit.

[4] `http://lancet.mit.edu/galib-2.4/`

5.5. Vergleiche und Analysen

5.5.1. HSP

Zur Analyse unserer Implementierungen vergleichen wir das bestehende, nicht optimale FF System mit unserer nicht zulässigen HSP-Add-Atom-Heuristik sowie unseren zulässigen HSP-Max-Atom- und HSP-Max-Pair-Heuristiken. Hierbei wird jeweils der Algorithmus A* benutzt. Die Ergebnisse sind in Abbildung 5.11 dargestellt.

Wichtig herauszustellen ist hier der Trade-Off zwischen Optimalität und Geschwindigkeit. Besonders in den schwierigeren Problemen der jeweiligen Domäne (Logistics 7-0; Blocks 8-0, 9-1; Depot 1935,6512; Driverlog 2-2-4; Satellite 2,3,9) ist die Diskrepanz der Laufzeiten wie auch der Expansionen zwischen optimalem und suboptimalem Planer sehr deutlich. Hierbei möchten wir bemerken, dass die von uns implementierte HSP-Add-Atom-Heuristik durchaus mit FF konkurrieren kann. Die Laufzeiten bewegen sich in der selben Größenordnung, die Expansionzahl ist in unseren Analysen im Allgemeinen nicht größer, sondern oft niedriger als bei FF (Logistics 7-0; Blocks 8-0; Depot 1935). Eine Ausnahme bildet hier die Domäne Satellite, bei der HSP-Add-Atom deutlich schlechter abschneidet.

Auch wenn die suboptimale Lösung nur wenige Planschritte länger ist als die optimale Lösung, muss das Finden des kürzesten Plans teuer erkauft werden. In einigen Fällen ist die Verbesserung tatsächlich minimal (Logistics 4-1; Blocks 9-1; Zeno Travel 2-6).

Manche Situationen verlangen allerdings aufgrund von sehr teuren Planungsschritten nach der kürzesten Lösung. Bei der Blocks-Domäne werden diese optimalen Lösungen tatsächlich auch in akzeptabler Zeit gefunden. In anderen Bereichen wie zum Beispiel Depot, finden wir einen kürzeren Plan, allerdings auch in deutlich höherer Zeit (Depot 1935). Weiterhin möchten wir bemerken, dass HSP-Max-Pair deutlich informativere heuristische Werte liefert als HSP-Max-Atom, wodurch die Expansionszahl wesentlich niedriger ausfällt (Zeno Travel 2-4; Logistics 4-1), allerdings die Zeit für die Berechnung dieser Werte übermäßig hoch liegt.

Wir müssen also feststellen, dass die Wahl des Planungssystems starkt von der Anwendung abhängt.

5.5.2. Pattern Data Bases

In dem Test der Pattern Data Bases unterscheiden wir zwischen der Implementierung mit Aufbau der Patterns durch ein Greedy Bin-Packing und der optimierenden Variante mit unterschiedlichen Parametern. Aufgrund des Einsatzes von genetischen Algorithmen ist die Findung der heuristischen Werte nicht deterministisch, so daß wir bei diesen Tests jeweils mehrere Durchläufe des Planungssystems heranziehen und die Durchschnittswerte zum Vergleich heranziehen. Selbstverständlich ist hierbei zu bemerken, daß es bei vielen Problemen die Möglichkeit gibt, durch ungeschickt gewählte Parameter die Planfindung zu verhindern. Dieser Fehler ist uns, wie man im folgenden sieht, auch unterlaufen.

Da die Tests durch die wiederholte Durchführung recht aufwändig sind stellen wir hier die Durchschnittswerte besonders aussagekräftiger und interessanter Domänen in Diagrammform vor.

Die Beschriftung der horizontalen Achse beschreibt die Parameterwahl in folgender Weise:

			FF			HSP-Add-Atom			HSP-Max-Atom			HSP-Max-Pair		
	Domain	Problem	TT	Exp	PL	TT	Exp	PL	TT	Exp	PL	TT	Exp	PL
AIPS-2000	Logistics	4 - 1	0.05	316	9	0.05	43	9	6.16	38281	9	116.47	9129	9
		6 - 0	0.08	689	9	0.05	46	10	31.94	234643	9	—	—	—
		7 - 0	2.28	13405	13	0.14	57	14	—	—	—	—	—	—
	Blocks	5 - 0	0.03	64	12	0.04	39	12	0.04	170	12	0.53	51	12
		5 - 2	0.03	72	16	0.04	38	16	0.08	361	16	1.14	102	16
		6 - 1	0.05	210	12	0.05	44	10	0.27	1024	10	5.01	194	10
		7 - 1	0.24	3296	22	0.10	121	24	7.40	36567	22	393.20	16218	22
		8 - 0	0.46	5307	22	0.17	199	22	38.32	120719	18	—	—	—
		9 - 1	0.23	1558	28	0.88	962	34	572.85	1519483	28	—	—	—
ICAPS02	Depot	7512	0.09	119	12	0.12	15	12	2.36	3840	12	60.84	764	12
		1935	0.43	1422	20	0.51	98	22	1176	1231700	17	—	—	—
		6512	16.77	59190	23	41.08	12740	25	—	—	—	—	—	—
	Zeno Travel	2 - 4	0.08	17	6	0.10	6	5	1.07	806	5	2.97	16	5
		2 - 6	0.19	309	8	0.58	178	8	220.45	275663	8	—	—	—
	Driverlog	2-2-2	0.04	8	7	0.04	8	7	0.04	11	7	0.21	8	7
		2-2-4	0.03	42	7	0.06	27	8	2.69	6392	7	25.46	1182	7
	Satellite	1	0.03	11	8	0.03	9	8	0.04	123	8	0.09	29	8
		2	0.04	17	12	0.03	13	12	33.95	106964	12	—	—	—
		3	—	—	—	—	—	—	—	—	—	—	—	—
		9	3.94	343	15	97.51	1736	16	—	—	—	—	—	—

Abbildung 5.11.: Vergleiche der Planungssysteme in unterschiedlichen Domänen (—: Kein Ergebnis innerhalb von 600 Sekunden); TT: Total Time, Exp: Expansionen, PL: Optimierte Planlänge

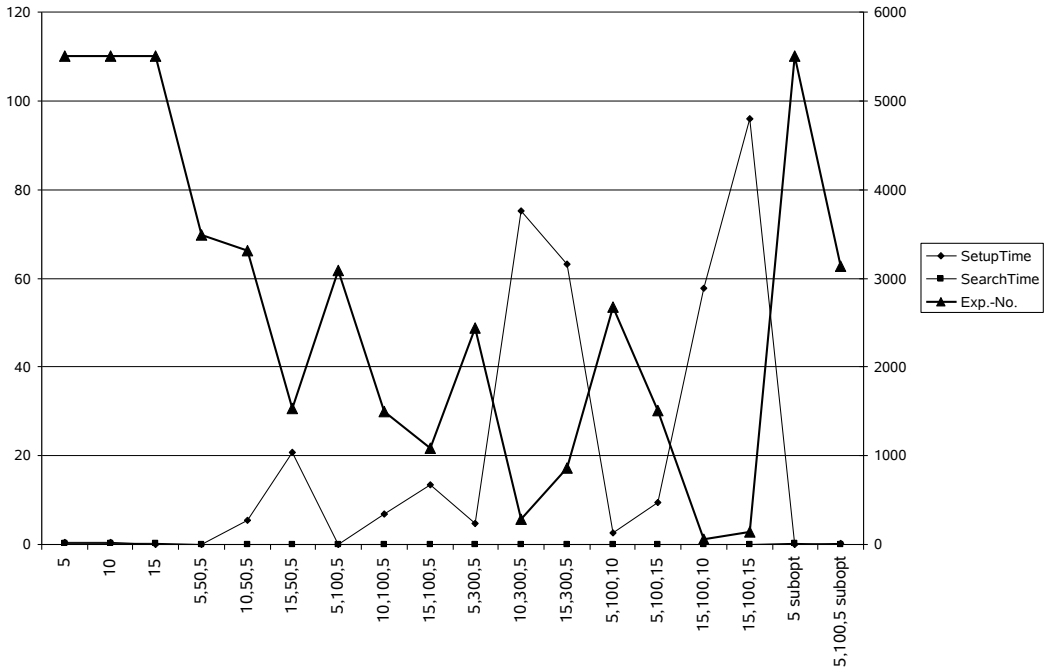


Abbildung 5.12.: Test von Logistics 4-1

Beschriftung	Bedeutung
5	BinPacking: log Maximale Patterngröße 5
5,50,5	Genetisch: log Max. Patterngr. 5, Generationen 50, Population 5
...	

5.6. Aufruf und Handhabung

Das System wird durch den Aufruf `./mips` unter unixoiden Betriebssystemen gestartet. Unter Windows ist der Aufruf entsprechend `mips`. Mögliche Optionen lassen sich durch `./mips -h` anzeigen. Sie sind in Abbildung 5.16 aufgelistet. Entsprechend der Erläuterungen dieser Ausgabe lassen sich mit den Parametern die Suchstrategie wie auch die Heuristik einstellen. Weiterhin werden Problem- und Domaindatei übergeben.

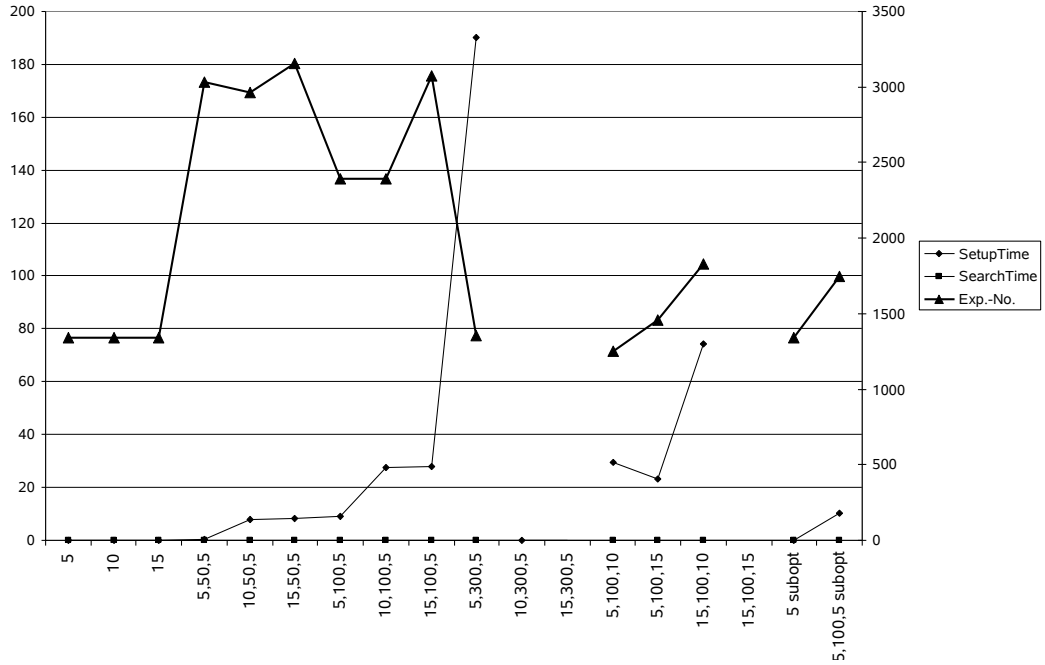


Abbildung 5.13.: Test von Depot 7512

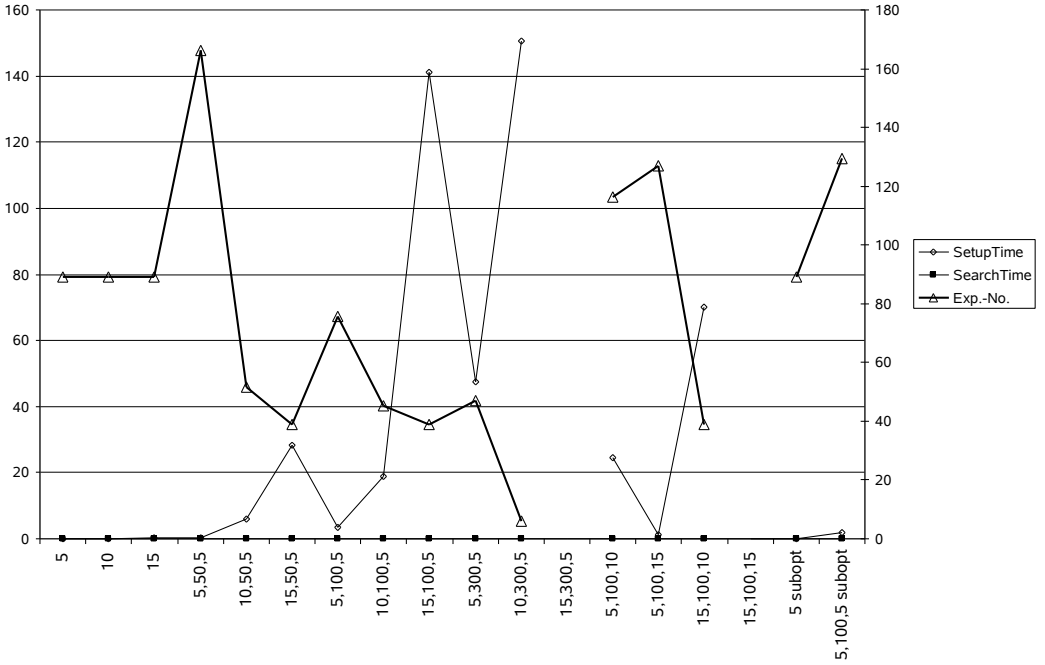


Abbildung 5.14.: Test von Zeno-Travel 2-4

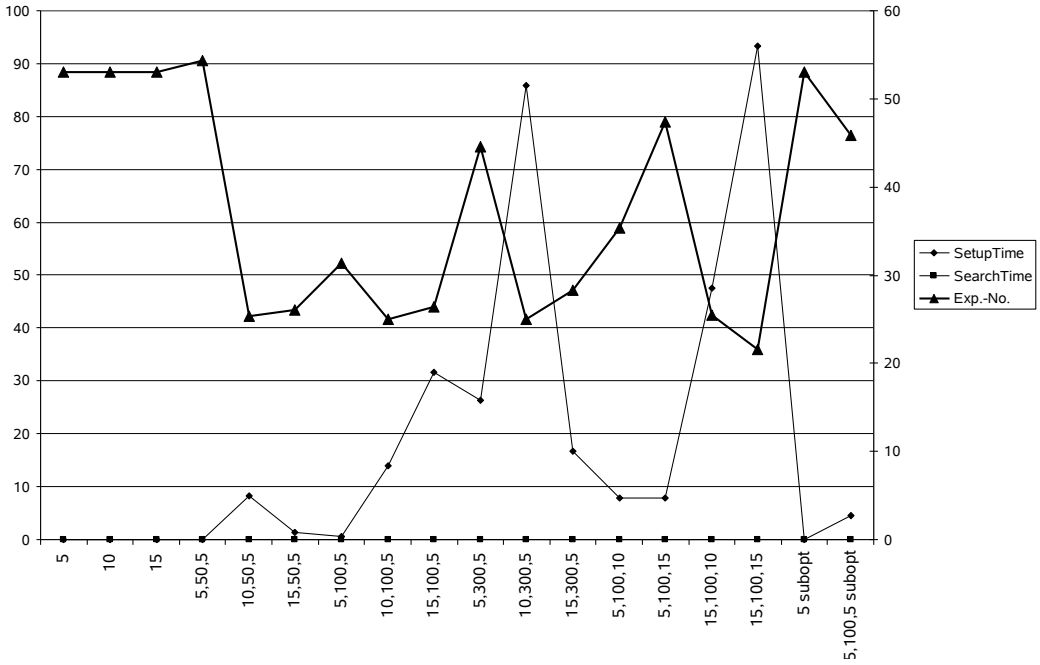


Abbildung 5.15.: Test von Driverlog 2-2-2

```
user@computer:~/> ./mips -h
ModPlan-Version (Roman Klinger, Kenneth Kahl)

MIPS 2.2 - the Model Checking Integrated Planning System
developed by Stefan Edelkamp

ModPlan elements by Roman Klinger and Kenneth Kahl:

usage: mips [<option> ...]
       mips [<option> ...] <problem file>
       mips [<option> ...] <domain file> <problem file>

If no domain file is specified, domain.pddl serves as a default.
If no problem file is specified, problem.pddl serves as a default.

Options:

Exploration algorithm:
  -A 0 = Explicit A*
  -A 1 = Explicit Hill-Climbing (default)
  -A 2 = Explicit IDA*
Heuristic Functions:
  -F:   Relaxed planning heuristic (default)
  -P x: Pattern database heuristic, log number of explicit patterns
  -G x: Pattern database heuristic, optimized with genetic algorithms,
        log number of explicit patterns in initialisation
  -K 0: HSP Add-Atom-Heuristik
  -K 1: HSP Max-Atom-Heuristik
  -K 2: HSP Max-Pair-Heuristik
  -K 3: HSP Max-Pair-Heuristik with Preprocessing (BUGGY)
Weightening of heuristic function
  -W x: 1 + weight in x%   (default 100)

Various:
  -O x: Optimizing quality by scheduling relaxed plans (default 0)
  -i  : VEPA vizualization information
  -p  : flushing intermediate representation
  -v  : verbose information
  -d  : debug information
  -a  : Turn off additivity of the patterns for Pattern Data Bases
  -u  : break for user editing patterns
 The Pattern-XML-File is named: ".transfer.xml" in the same
 directory as mips, if you want to edit it, enter the name or the
 path of your editor in "lib/edit.cfg"
```

Abbildung 5.16.: Mögliche Optionen für den Aufruf von mips

Kapitel 6.

Durative-FF

Mohammed Nazih, Abdelaziz Elalaoui, Khalid Lahiane

6.1. Einführung

Metric-FF ist eine Erweiterung des FF-Planers um nummerische Zustandsvariablen. Das in C geschriebene System hat beim dritten internationalen Planungswettbewerb gut abgeschnitten. Der zeitliche Ablauf zur Durchführung eines Plans ist offensichtlich sehr wichtig. Bei Metric-FF werden die Ausführungszeiten einzelner Aktionen nicht berücksichtigt. Das Ziel unserer Gruppe bestand darin, Metric-FF zu Durative-FF zu erweitern, so dass Problemlösungen unter Berücksichtigung von Aktionszeitdauern und zeitlichen Literalen ermöglicht wird.

Der Inhalt dieses Kapitels gliedert sich in fünf Abschnitte. Der erste Abschnitt bietet einen Überblick über den Metric-FF Planer. Der zweite Abschnitt bildet eine solide Grundlage für das temporale Planen. Danach wird ausführlich auf die Erweiterung des Metric-FF zu Durative-FF Planers eingegangen. Der dritte Abschnitt beschäftigt sich mit der Einführung der zeitlichen Literalen in das temporale Planen. Im vierten Abschnitt werden die Visualisierung und die Manipulation der Abhängigkeiten zwischen der Operatoren dargestellt. Der letzte Abschnitt bewertet die Leistung des Durative-FF Planers anhand von Vergleichen (in Form von Diagrammen) mit konkurierenden Planern.

6.2. Metric-FF Planer

In diesem Abschnitt beschreiben wir die Funktionsweise des Metric-FF-Planers, seine verwendete relaxierte Planungsheuristik und schlisslich seinen Parser-Aufbau. Metric-FF ist eine Erweiterung von FF-Planer (ein klassisches Handlungsplanungssystem, das von Jörg Hoffmann entwickelt wurde). FF löst beliebige Planungsprobleme, die in der standardisierten Eingabesprache PDDL (Planning Domain Definition Language) formuliert sind und kommt dabei mit klassischen Problemen (STRIPS) ebenso zurecht wie mit komplexen Domänen mit Quantoren und bedingten Effekten (ADL). Der Metric-FF-Planer, eine Erweiterung zu FF, kann zusätzlich mit Zahlen (nummerischen Zustandsvariablen) umgehen. Beide Systeme sind in C implementiert [23].

6.2.1. Struktur des Metric-FF Planers

Die Funktionsweise des Metric-FF Planers lässt sich in drei Phasen unterteilen. In der ersten Phase erhält Metric-FF als Eingabe ein Problem und eine Domäne. Wenn die Eingaben korrekt definiert sind, werden sie geparst. Der Metric-FF-Parser (wird im nächsten Unterabschnitt erläutert) fängt damit an, die Domäne zu analysieren, denn es ist für die PDDL-Sprache wichtig, die Domäne vor dem Einlesen des Problems zu definieren. Danach wird getestet, ob es sich um ein ADL-Problem handelt, denn beim Metric-FF können nur ADL-Probleme bearbeitet werden. Bei der zweiten Phase handelt es sich um die erste Vorverarbeitung (preprocessing). Es wird die Domäne durch Ganzzahlen kodiert. Danach werden die wichtigsten Informationen gesammelt und der initiale Zustand wird in drei Arrays zerlegt: ein Array für individuelle Prädikate, ein Array für alle statischen Relationen und ein Array für nicht statische Relationen. Es werden dann alle prädikatenlogische Formeln in der Domänebeschreibung normalisiert und die negativen Vorbedingungen übersetzt. In der dritten Phase, werden die Informationen von der *Normoperator*- und der *Pseudoaction*-Struktur zur

Action-Struktur weitergeleitet. Danach werden die relevanten Fakten gesammelt und die finale Problem- und Domänedarstellung erstellt. Anschließend werden die Informationen von der *Action*-Struktur bis zur höchsten Ebene weitergeleitet. Abschließend wird die Suche nach der Lösung durchgeführt (siehe Abbildung 6.1).

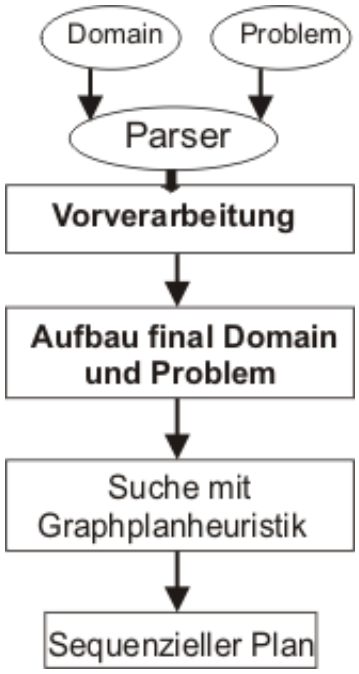


Abbildung 6.1.: Metric-FF Planer

6.2.2. Die verwendete relaxierte Planungsheuristik

Bei Metric-FF wird das Planen als Suche aufgefasst: Startend beim initialen Zustand werden die möglichen Zustände abgesucht, um schließlich zu einem Zustand zu gelangen, der unserem Ziel entspricht. Die Aktionen, die den Zustandswechseln dienten, bilden dann den Plan. Metric-FF benutzt die FF-Heuristik, sie basiert auf den relaxierten Planinformationen. Für ein gegebenes nummerisches Problem (V, P, A, I, G) und einen Zustand s, ist der FF-Heuristikwert $h(\mathrm{s}) = \infty$, falls die relaxierte Planerzeugungsprozedur keinen relaxierten Plan gefunden hat, andernfalls $h(\mathrm{s}) = \sum_{t \in T} |A_t|$, wobei $T = \{1, 2, ..., \mathit{finallayer}\}$ und A_t die Operatormenge ist, die in der Schicht t von der relaxierten Planextraktionsprozedur des Metric-FF ausgewählt wurde.

6.3. Erweiterung des Metric-FF zum Durative-FF Planer

In diesem Abschnitt betrachten wir Aktionen, die mit Ausführungszeiten belegt sind. Dies führt zu sequentiellen oder parallelen temporalen Plänen. Danach wird auf die Erweiterung der Parser für temporalen Domänen näher eingegangen. Abschließend wird der Durative-FF Planer detailliert beschrieben und anhand eines Beispiels ein paralleler Plan berechnet.[9]

6.3.1. Parallelisierung sequentieller Pläne

Wir beschreiben kurz zuerst, was ein temporales Modell ist. Danach beschäftigen wir uns mit der Abhängigkeiten von Operatoren und der resultierenden Vorgängerbeziehung von Aktionen. Anschliessend stellen wir einen Ansatz für das temporale Planen dar, bei dem sequentiell erzeugte Pläne parallelisiert werden. Darauf folgt eine Einführung der Schedulingheuristik, die auf dem Pert-Ansatz basiert.

Temporales Modell

Die PDDL 2.1 Level 3 Beschreibungen enthalten die temporalen Annotationen **at start**, **over all** und **at end**. In Abbildung zeigen wir zwei verschiedene Optionen diese Information in einfache Operatoren zu zerlegen, um eine Semantik für sequentielle Pläne zu gewinnen. Im ersten Fall Abbildung 6.2 (oben rechts), wird der Operator in drei Einzeloperatoren zerlegt. Dieses ist die vorgeschlagene Semantik von Fox und Long. Für die Parallelisierung sequentieller Pläne bietet sich die alternative Operatorrepresentation in Abbildung 6.2 (unten rechts) an.

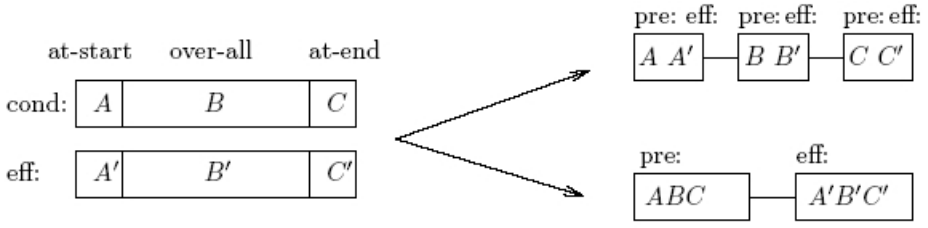


Abbildung 6.2.: Kompilierung von temporalen Operatoren

Abhängigkeit der Operatoren

Die Definition der Abhängigkeit von Operatoren ermöglicht die Berechnung von optimalen Anordnungen (Schedules) von sequentiellen Plänen gemäß der generierten Aktionsfolge und der implizierten kausalen Struktur der Operatoren untereinander. Wenn alle Operatoren paarweise abhängig sind, liefert in diesem Fall keine Anordnung eine Verbesserung des Schedules.

Propositionaler Konflikt: Zwei Operatoren sind abhängig, falls der folgende propositionale Konflikt vorliegt, d.h. die propositionalen Vorbedingungen eines Operators haben einen nicht-leeren Schnitt mit dem Effekten des anderen Operators.

Nummerischer Konflikt: Man unterscheidet zwischen zwei Konflikten: Direkter und indirekter nummerischer Konflikt. Direkter nummerischer Konflikt, d.h. der Kopf eines nummerischen Zuweisungsoperators einer Aktion ist enthalten in einer Bedingung der anderen Aktion. Indirkter nummerischer Konflikt, d.h. der Kopf einer nummerischen Zuweisung in einer Aktion tauchet innerhalb des Restes einer Zuweisung in einer anderen Aktion auf.

Beispiel: Direkter nummerischer Konflikt.

```
(reful plan-a city-a)        (fly plane city-a city-c)
```

```
nummerischer Bedingung:
(fuel ?p) < (capacity ?p)          (fuel p) >= (distance a c)*(stow-burn p)
nummerische Effekten:
(fuel p) += (capacity ?p) (total-fuel-used) += (distance a c)*(stow-burn p)
                                    (fuel p) -= (distance a c)*(stow-burn p)
```

Im Beispiel haben die Operatoren (reful plan-a city-a) und (fly plane city a city c) einen direkten nummerischen Konflikt. Indirekte nummerische Konflikte tauchen im Beispielproblem nicht auf. Die Abhängigkeiten der Operatoren werden genutzt, um ein optimales nebenläufiges Arrangement der Operatoren in einem sequentiellen Plan zu finden. Falls O_2 von O_1 abhängt und O_1 vor O_2 in dem sequentiellen Plan liegt, dann muss O_1 vor O_2 ausgeführt werden.

Pert Scheduling

Procedure Critical-Path
Eingabe: Sequenz von Aktionen $O_1, ..., O_k$,
Ordnung $\leq_d$
Output: Makespan

for all i $\in \{1, \ldots, k\}$
  e(O_i) $\leftarrow$ d(O_i)
  for all j $\in \{ 1, \ldots,$ i-1$\}$
    if($O_j \leq_d O_i$)
      if e(O_i) < e(O_j)+d(O_i)
      e(O_i) $\leftarrow$ e(O_j) + d(O_i)
return $max_{1 \leq i \leq k}$ e(O_i)

Algorithmus zur Berechnung der kritischen Pfadkosten.

Der kritsche Pfad ist eine Sequenz von Aktionen, so dass die Gesamtzeit dieser Aktionen größer oder gleich zu allen anderen Operatorpfaden ist.

Der PERT-Scheduling-Algorithmus nimmt als Eingabe eine Folge von Operatoren und gibt als Ausgabe die Planlänge eines optimalen parallelen Plans.

Metric-FF wurde auf einem nummerischen ZENO-TRAVEL-Problem zum Laufen gebracht und hat folgenden sequentiellen Plan erzeugt:

```
step0 : BOARD SCOTT PLANE CITY-A
step1 : FLY PLANE CITY-A CITY-C
step2 : BOARD ERNIE PLANE CITY-C
step3 : BOARD DAN PLANE CITY-C
step4 : FLY PLANE CITY-C CITY-D
step5 : DEBARK ERNIE PLANE CITY-D
step6 : DEBARK SCOTT PLANE CITY-D
step7 : REFUEL PLANE CITY-Dthe
```

```
step8 : FLY PLANE CITY-D CITY-C
step9 : FLY PLANE CITY-C CITY-A
step10: DEBARK DAN PLANE CITY-A
```

Man hat hier einfach die Aktionszeitdauer einen konstanten Wert zugewiesen, und zwar 1.

Nachdem wir den PERT-Algorithmus implementiert und dem Metric-FF hinzugefügt haben, haben wir ihn auf der ZENO-TRAVEL-Domäne wieder zum Laufen gebracht. Er hat folgenden parallelen Plan erzeuget:

```
step1: BOARD SCOTT PLANE CITY-A
step2: FLY PLANE CITY-A CITY-C
step3: BOARD ERNIE PLANE CITY-C
       BOARD DAN PLANE CITY-C
step4: FLY PLANE CITY-C CITY-D
step5: DEBARK ERNIE PLANE CITY-D
       DEBARK SCOTT PLANE CITY-D
       REFUEL PLANE CITY-D
step6: FLY PLANE CITY-D CITY-C
step7: FLY PLANE CITY-C CITY-A
step8: DEBARK DAN PLANE CITY-A
```

Der kritsche Pfad-Wert beträgt: 8

Schedulingheuristik

Die FF-Schedulingheuristik ist eine Erweiterung der relaxierten FF-Heuristik. Sie wird beim Durchsuchen des Zustandsraums von dem Planer als eine Distanz-Schätzung vom aktuellen Zustand zum Zielzustand verwendet.

Definition Gegeben sei ein durative-nummerisches Problem (V, P, A, I, G) und ein Zustand s. Der FF-Schedulingheuristikwert ist $sh(\mathrm{s}) = \infty$, falls die relaxierte Planerzeugungsprozedur keinen relaxierten Plan gefunden hat, andernfalls ist $sh(\mathrm{s}) = pert(\cup_{t \in T} A_t)$, wobei $T = \{1, 2, ..., \mathit{finallayer}\}$ und A_t die Operatormenge bildet, die in der Schicht t von der relaxierten Planextraktionsprozedur ausgewählt wurde.

Anwendung der Schedulingheuristik Beim Expandieren des aktuellen Zustandsknotens berechnet man die relaxierte Planheuristik und speichert die Operatormenge in einem Array, deren Größe der relaxierter Planheuristikwert bildet, und wendet darauf die *pert*-Funktion an, die wiederum die *depend*-Funktion aufruft, die die Abhängigkeiten zwischen den Operatoren an die *pert* zurückgibt. Anhand dieser Abhängigkeiten berechnet die *pert*-Funktion den parallelen Plan, und gibt die Länge des längsten Pfades zurück. Wir leiten diesen Wert an die $search\text{-}for\text{-}better\text{-}state$-Funktion weiter, die ihn mit dem relaxierten Planheuristikwert des aktuellen Zustandes vergleicht. Falls der neue Wert kleiner als der Alte ist, wird der alten Wert durch den Neuen ersetzt. Dieser Schritt wird wiederholt, bis der relaxierte Planheuristikwert den Wert 0 erreicht, was bedeutet, dass der Zielknoten erreicht ist, anderenfalls ist

das Problem unlösbar, weil der Zielzustand nicht von diesem Zustand aus erreicht werden kann. Falls der neue Wert größer als der Alte ist, wird die relaxierte Graphplanheuristik erneut berechnet und die *pert*-Funktion auf die neue Operatormenge angewendet. Dies wird wiederholt, bis ein kleinerer Wert gefunden wird, oder der aktuelle Zustand eine Sackgasse ist, also der Zielzustand nicht von diesem Zustand aus erreicht werden kann.

Vergleich zwischen FF-Schedulingheuristik und relaxierter Graphplanheuristik Wir haben einen Vergleich zwischen FF-Schedulingsheuristik und relaxierter Graphplanheuristik durchgeführt und sind zum Entschluss gekommen, dass die FF-Schedulingheuristik keine verbesserte Heuristik gegenüber der Graphplanheuristik ist. Manchmal liefern sie gleiche Ergebnisse, meistens liefert Graphplanheuristik jedoch bessere Ergebnisse gegenüber der FF-Schedulingheuristik.

6.3.2. Parsen der temporalen Domänen

In diesem Unterabschnitt werden wir uns mit der Erweiterung des Parsers beschäftigen. Die temporale Domäne unterscheidet sich von der vorherigen Domäne, indem die erste Domäne *durative-action* anstatt normale *action* enthält. Deshalb werden wir uns nur auf *durative-action* beschränken.

Gegeben sei diese PDDL-Domänenbeschreibung aus ZenotravelTandN-Time:

```
(:durative-action   fly
   :parameters (?a - aircraft  ?c1  ?c2 - city)
   :duration (= ?duration (/ (distance ?c1 ?c2) (slow-speed ?a)))
   :condition (and (at start (at ?a ?c1))}
                  (at start ( > = (fuel ?a)
                            (* (distance ?c1 ?c2) (slow-burn ?a)))))
   :effect (and (at start (not (at ?a ?c1)))
                (at end (at ?a ?c2))
                (at end (increase total-fuel-used
                     (* (distance ?c1 ?c2) (slow-burn ?a))))
                (at end (decrease (fuel ?a)
                     (* (distance ?c1 ?c2) (slow-burn ?a))))))
```

Syntaxanalyse einer Aktion

Wir stellen die Syntaxanalyse der Aktion mit Hilfe der Backus-Naur-Form (BNF) dar. Sie wird oft auch kontextfreie Grammatik genannt.

Wie es in Abbildung 6.3 dargestellt ist: Die durative-Aktion besteht aus du_param_def, die die Parameter dieser Aktion beschreibt, und der Körper der Aktion, der aus den Effekten, Vorbedingungen und die Aktionszeitdauern bestehen kann. Diese kontextfreie Grammatik hat vier komponenten:

1. Die Terminalsymbole sind in diesem Fall die Token, die in der Lex-Datei definiert sind.
2. Eine Menge von nicht Terminalsymbolen, die Überbegriffe für Konstruktionen sind, die sich aus terminalen und /oder nicht terminalen Symbolen zusammensetzen.

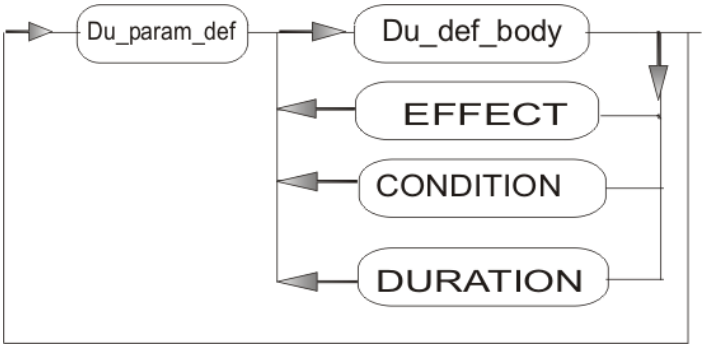


Abbildung 6.3.: Syntax-Diagramm für eine Aktion

3. Eine Menge von Produktionen, die die linke Seite mit der rechten Seite verbinden.

4. Startsymbol, das nicht als Terminalsymbol sein soll.

Metric-FF Lexer

Im ersten Schritt des Einlesevorgangs wird die PDDL-Datei vom Lexer eingelesen. Dieser liest jedes Zeichen einzeln ein und gibt die erkannten Token anschliessend an den Parser weiter. Das Verhalten des Lexers ist in der Datei *lex-fct_pddl.l* für das Problem bzw. (*lex-ops_pddl.l* für die Domäne) spezifiziert, dies wurde von der Metric-FF Gruppe erweitert und angepasst. Aus dieser wird der Lexer (*lex-ops_pddl.c* für die Domäne bzw. *lex-fct_pddl.c* für das Problem) mit Flex generiert. Dabei wird bei der Parsergenerierung die Datei *scan-ops_pddl.tab.c* für die Domäne und die Datei *scan-fct_pddl.tab.c* für das Problem benötigt. Nun folgen einige Zeilen aus der Spezifikation des Lexers der Domäne, die für die lexikalische Analyse der Pddl-Datei notwendig sind:

```
{d}{e}{f}{i}{n}{e}  {  return(DEFINE_TOK); }
{d}{o}{m}{a}{i}{n}  {  return(DOMAIN_TOK); }
":"{r}{e}{q}{u}{i}{r}{e}{m}{e}{n}{t}{s}  {
                        return(REQUIREMENTS_TOK); }}
":"{t}{y}{p}{e}{s}  {  return(TYPES_TOK); }
":"{c}{o}{n}{s}{t}{a}{n}{t}{s}  {  return(CONSTANTS_TOK); }
":"{d}{u}{r}{a}{t}{i}{v}{e}{a}{c}{t}{i}{o}{n} {
                        return(DURATIVE_ACTION_TOK);}
"(" {  return(OPEN_PAREN); }
")" {  return(CLOSE_PAREN);  }
":"{c}{o}{n}{d}{i}{t}{i}{o}{n}   {  return (CONDITION_TOK); }
":"{d}{u}{r}{a}{t}{i}{o}{n}   { return (DURATION_TOK);   }
{a}{t}  {   return (AT_TOK);  }
{s}{t}{a}{r}{t} {  return (START_TOK);   }
{e}{n}{d}    {    return (END_TOK);    }
 {o}{v}{e}{r}   {   return (OVER_TOK);   }
{a}{l}{l}   {    return (ALL_TOK);    }
```

Metric-FF Parser

Nach dem Einlesen der Datei durch den Lexer, erfolgt die Weiterverarbeitung der erkannten Token durch den Parser. Dieser muss nun prüfen, ob er Produktionen findet, so dass die erkannten Token zu der vom Parser akzeptierten Grammatik passen. Ist dies nicht der Fall, gibt der Parser eine Fehlermeldung aus. Der Parser selbst ist in der Datei *scan-ops_pddl.y* für die Domäne und *scan-fct_pddl.y* für das Problem spezifiziert, welche ebenfalls von der Metric-FF Gruppe erweitert wurde. Die Spezifikation des Parsers muss gründlich verändert bzw. angepasst werden. Mit Hilfe des bereits im Konzept genannten Tools Flex und Bison wird die Datei *scan-ops_pddl.tab.c* für die Domäne bzw. (*scan-fct_pddl.tab.c* für das Problem) generiert, die für die Kommunikation zwischen Lexer und Parser notwendig sind. Nun folgt der für unser Beispiel benötigte Auszug aus der Spezifikation des Parsers.

```
du_def_body:
    /* empty */
    |
    EFFECT_TOK du_adl_effect
        {
            durative_op->effects = $2;
        }
    du_def_body
    |
    CONDITION_TOK du_adl_goal_description
        {
            durative_op->preconds = $2;
        }
    du_def_body
    |
    DURATION_TOK du_atom_for
        {
            durative_op->duration = $2;
        }
    du_def_body
    ;
```

Metric-FF-Baum

Der Lexer bekommt als Eingabe eine PDDL-Datei, dann fängt er mit der Analyse der Datei an. Verstößt das Dokument gegen die Struktur der Grammatik (zum Beispiel falsche Reihenfolge oder falsche Schachtelung von Elementen, Verwendung unbekannter Elemente oder Attribute) oder tritt ein Syntax-Fehler auf die Zeile auf, liefert der Parser entsprechende

Fehlermeldungen. Falls alle Elemente im Dokument korrekt geschachtelt sind, baut der Parser bei der Analyse einen Parserbaum auf, dessen Knoten die Bestandteile des Dokuments repräsentieren.

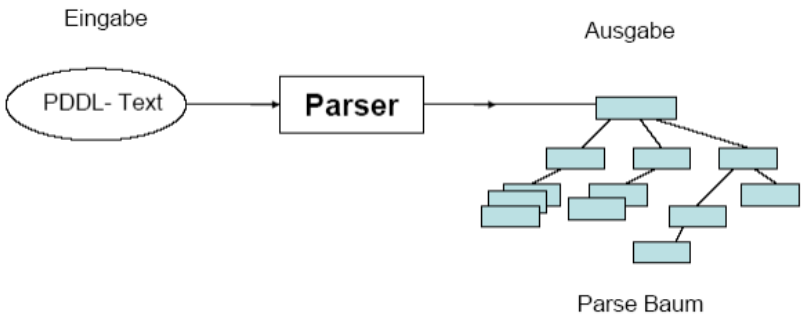


Abbildung 6.4.: Parsen eines PDDL-Dokuments

Nach einem erfolgreichen Parservorgang wird der Parserbaum weiterverarbeitet. Die Aktionen werden in einer verketteten Liste gespeichert:

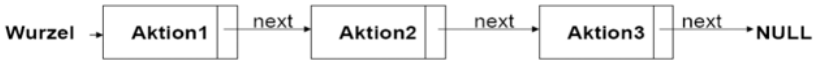


Abbildung 6.5.: Verkettete Liste der Aktionen

Jede Aktion, die das Wurzelelement eines Teilbaums darstellt, enthält Kinder (Unterknoten), die auch Informationen über Aktionszeitdauern, Vorbedingungen und Effekte enthalten.

Der Parser liest die PDDL-Datei und interpretiert sie als Baum:

Nachdem wir den Parser erweitert und die Daten in einem Baum gespeichert haben, mussten wir den Baum in einen anderen Baum ohne temporale Informationen umwandeln, damit Metric-FF den sequenziellen Plan findet. Dafür haben wir die Methode *extract_plops*() implementiert, um den klassischen Metric-FF Baum zu erzeugen.

6.3.3. Der Durative-FF Planer

Struktur des Durative-FF Planers

Damit Metric-FF die Ausführungszeiten einzelner Aktionen berücksichtigen kann, mussten wir die temporalen Informationen zu jeder Aktion hinzufügen. Abbildung 6.7 stellt die Strukturebenen des Metric-FF dar, als auch die beiden wichtigen Programmkomponenten Suche mit Schedulingheuristik und *durative-pert*. Was die Strukturen betrifft, handelt es sich um *Pl-Operator*, *Operator*, *Normoperator*, *Action*, in der die Operatoren instanziiert werden und schließlich die höchste Ebene *Opconn*.

Die temporalen Informationen mussten also quer durch alle Strukturebenen durchgezogen werden. Wie in Abbildung 6.8 dargestellt, werden in der Ebene *Pl-Operator* die zusätzlichen temporalen Informationen zu den Vorbedingungen und zu den Effekten aller Atomen

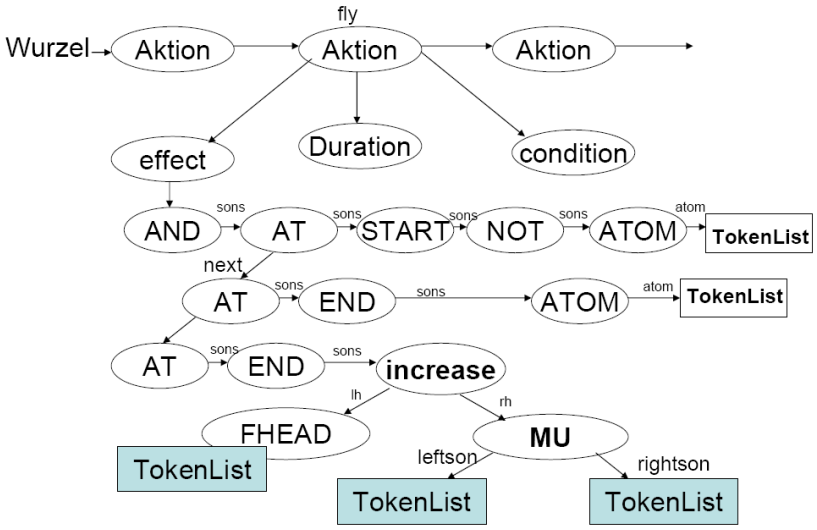


Abbildung 6.6.: Beziehung Elternkinds im Baum der Durative-Aktion *fly* aus vorheriger PDDL-Datei

hinzugefügt. Dafür wurden die zusätzlichen temporalen Informationen in einem Array gespeichert, das uns darüber informiert, mit welcher zeitlichen Modalität ein Atom belegt ist, d.h. *atstart*, *overall* oder *atend*. Enthält ein Atom *atstart* bzw. *overall* oder *atend*, so wird das Array mit 0. bzw. 1 oder 2 belegt.

Abbildung 6.9 zeigt das Durchziehen der Informationen von der ersten Ebene bis zur zweiten Ebene. Dafür wurde die Aktionszeitdauer von der *ParseExpNode*-Struktur in der ersten Ebene zur *ExpNode*-Struktur in der zweiten Ebene weitergeleitet. Erst in der Ebene *Operator* werden die propositionalen und nummerischen Effekte aufgesplittet. Effekte werden hier (siehe oben) als Literale (für propositionale Effekte) bzw. $Nummericeffect$-Struktur (für nummerische Effekte) represäntiert.

In Abbildung 6.9 bezeichnen die ganzzahligen Werte $fact\_t$ und $fluent\_t$ die zusätzlichen temporalen Informationen, die wir in dieser Ebene hinzugefügt haben. Die Vorbedingungen werden in dieser Ebene als $WffNode$-Struktur weitergeleitet. Abbildung 6.10 beschreibt das Durchziehen der temporalen Informationen von Ebene *Operator* bis zur Ebene $NormOperator$. In der Ebene $NormOperator$ werden die Effekte als $Normeffects$-Struktur dargestellt. Die temporalen propositionalen Effekte werden in dieser Ebene als $predicat\_t$ hinzugefügt während die temporalen nummerieschen Effekte als $function\_t$ hinzugefügt werden. Erst in der Ebene $NormOperator$ werden die Vorbedingungen in propositionalen und nummerischen Vorbedingungen aufgesplittet.

In dieser Ebene werden auch die zusätzlichen propositionalen Vorbedingungen als $fact\_t$ und die nummerischen Vorbedingungen als $nummeric\_preconds\_lh\_t$ hinzugefügt. Abbildung 6.11 zeigt das Durchziehen der temporalen Informationen von Ebene $Normoperator$ bis zur Ebene *Action*.

In der Ebene *Action* wird die Aktionszeitdauer instanziiert und wenn sie nicht vom Zu-

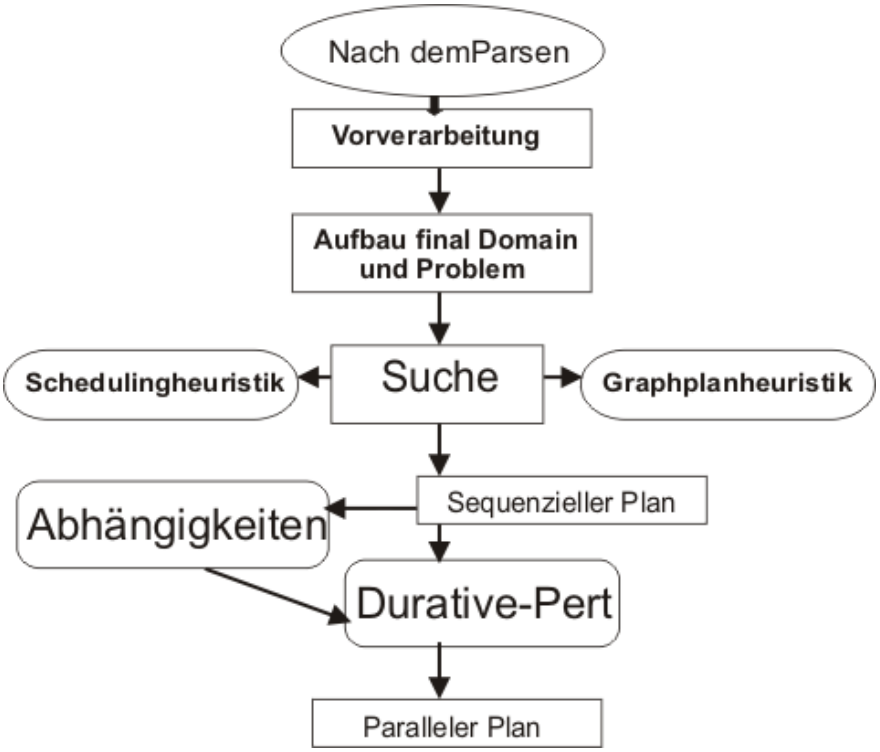


Abbildung 6.7.: Der Durative-FF Planer

stand abhängt, wird sie berechnet. Die Effekte werden an dieser Stelle als *ActionEffects*-Struktur dargestellt. Die propositionalen und nummerischen Effekte und die propositionalen Vorbedingungen werden in der Ebene *Action* als ganzzahliger Wert dargestellt. In dieser Ebene werden die zusätzlichen temporalen Informationen *adds_t* und *dels_t* als temporale propositionale Effekte, *numeric_effect_fl_t* als temporale nummerische Effekte, *preconds_t* als temporale propositionale Vorbedingungen und *numeric_preconds_lh_t* als temporale nummerische Vorbedingungen dargestellt.

Abbildung 6.12 zeigt das Durchziehen der temporalen Informationen von Ebene *Action* zu Ebene *OpConn*. In der Ebene *OpConn* werden die Effekte als *EfConn*-Struktur dargestellt. In dieser Ebene werden alle Attribute in Form ganzzahliger Werte dargestellt. Erst in der Ebene *Action* werden die nummerischen Effekte in nummerische Zuweisungsoperatoren übersetzt. Die temporalen nummerischen Zuweisungsoperatoren werden in dieser Ebene als *IN_fl_t* und *As_fl_t* dargestellt. Die temporalen propositionalen Vorbedingungen werden als *PC_t* dargestellt während die temporalen nummerischen Vorbedingungen als *f_PC_fl_t* dargestellt.
Direkt nach der Ebene *Opconn* wird die Schedulingheuristik (die wir vorher erklärt haben) gestartet. Es wird ein sequenzieller Plan erzeugt, der dann an den Durativ-Pert-Algorithmus übergeben wird, und schließlich ein paralleler Plan erstellt wird.

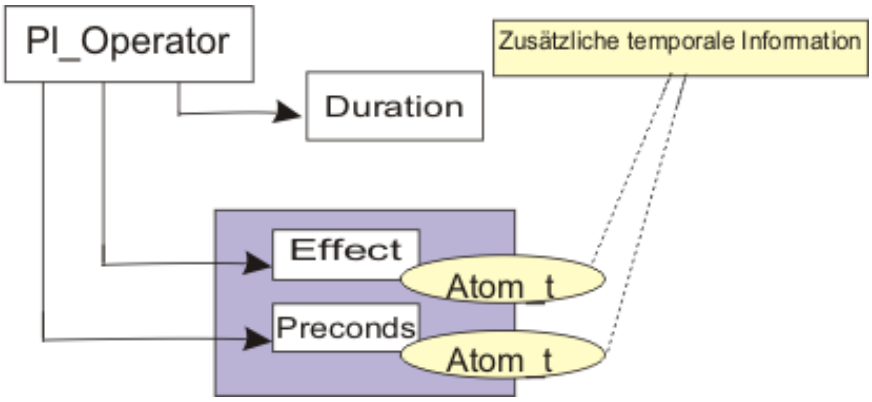


Abbildung 6.8.: Hinzufügen der temporale Informationen

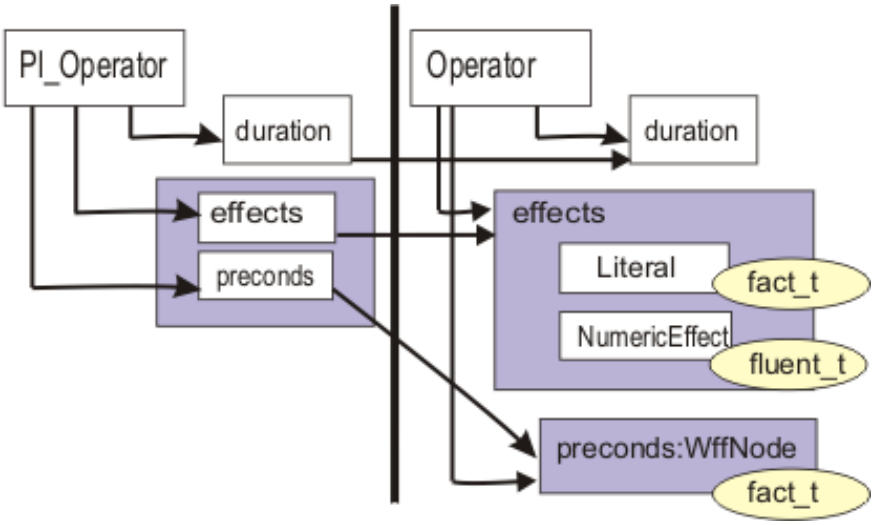


Abbildung 6.9.: Weiterleitung der temporalen Informationen von Ebene 1 zu Ebene 2

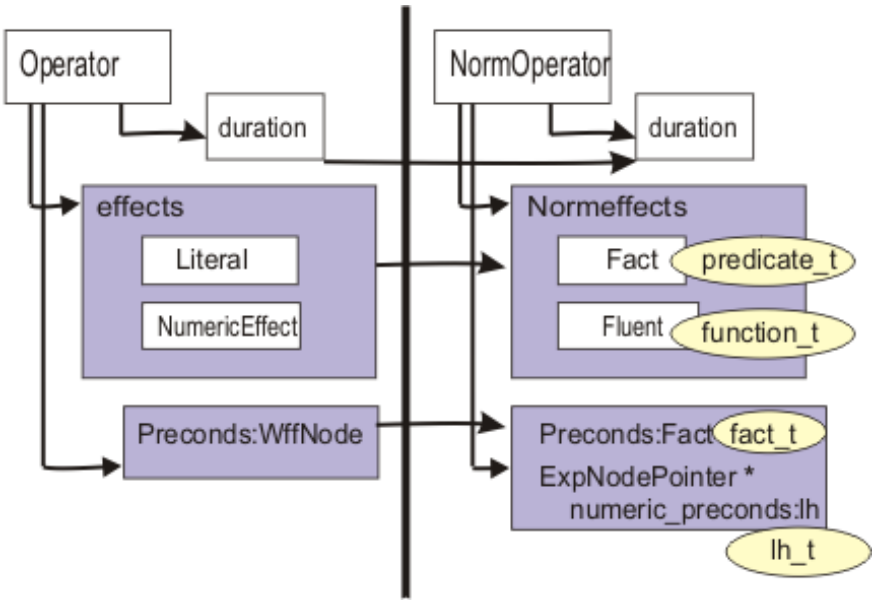


Abbildung 6.10.: Weiterleitung von Ebene 2 zur Ebene 3

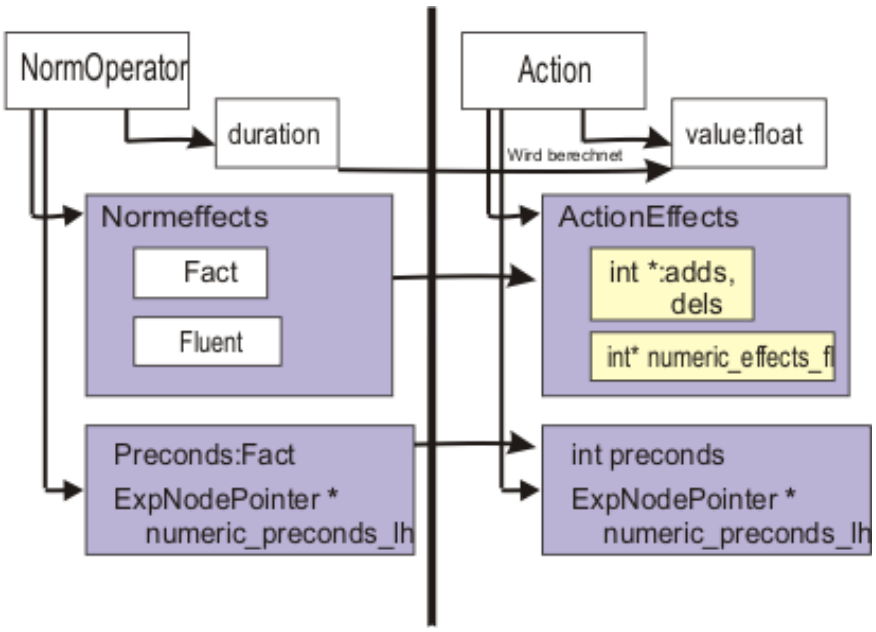


Abbildung 6.11.: Weiterleitung von Ebene 3 zur Ebene 4

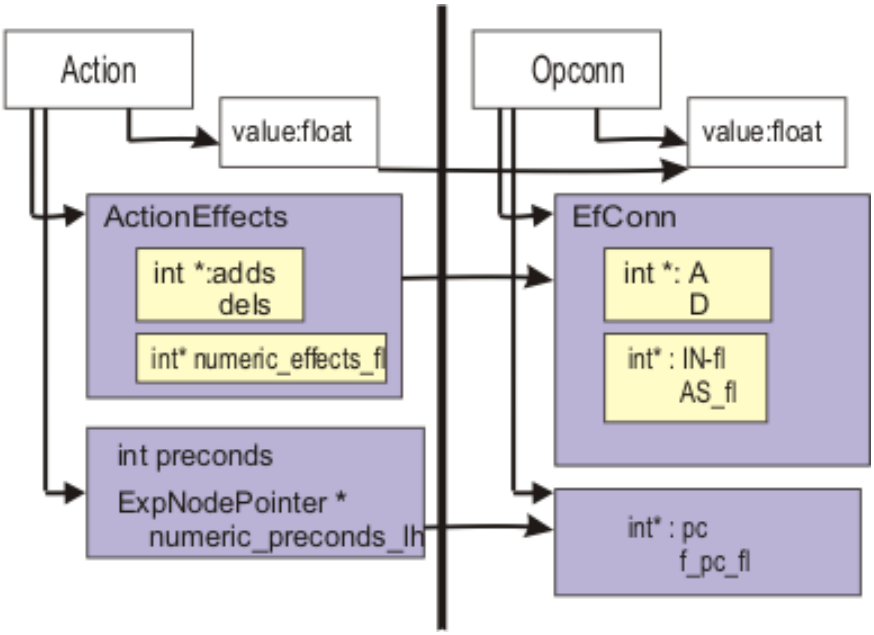


Abbildung 6.12.: Weiterleitung von Ebene 4 zur Ebene 5

Berechnung paralleler Pläne

Nachdem der sequentielle Plan berechnet ist, greift der Durative-FF Planer auf die Folge der Operatoren über der *durative-pert*-Funktion zu und erzeugt daraus, einen parallelen Plan gemäss der Abhängigkeiten, die von der *depend*-Funktion gefunden sind.

Wir haben den Durative-FF Planer auf ein Problem aus der Domäne DepotTime zum Laufen bringen lassen. Es hat sich folgender parallelen Plan ergeben:

```
; MakeSpan 53.35
0.00 : (LIFT HOIST2 CRATE2 CRATE1 DISTRIBUTOR1 ) [1.00]
0.00 : (DRIVE TRUCK1 DEPOT0 DISTRIBUTOR1 ) [10.00]
0.00 : (LIFT HOIST0 CRATE0 PALLET0 DEPOT0 ) [1.00]
0.00 : (LOAD HOIST0 CRATE0 TRUCK0 DEPOT0 ) [3.56]
3.56 : (DRIVE TRUCK0 DEPOT0 DISTRIBUTOR1 ) [3.33]
10.00: (LOAD HOIST2 CRATE2 TRUCK1 DISTRIBUTOR1 ) [29.67]
39.67: (DRIVE TRUCK1 DISTRIBUTOR1 DISTRIBUTOR0 ) [2.00]
39.67: (DRIVE TRUCK1 DISTRIBUTOR0 DEPOT0 ) [2.00]
39.68: (LIFT HOIST2 CRATE1 PALLET2 DISTRIBUTOR1 ) [1.00]
39.68: (LOAD HOIST2 CRATE1 TRUCK0 DISTRIBUTOR1 ) [1.33]
41.02: (UNLOAD HOIST2 CRATE0 TRUCK0 DISTRIBUTOR1 )[10.67]
41.02: (DROP HOIST2 CRATE0 PALLET2 DISTRIBUTOR1 ) [1.00]
41.67: (UNLOAD HOIST0 CRATE2 TRUCK1 DEPOT0 ) [9.89]
41.67: (DROP HOIST0 CRATE2 PALLET0 DEPOT0 ) [1.00]
51.69: (DRIVE TRUCK0 DISTRIBUTOR1 DISTRIBUTOR0 ) [0.67]
52.35: (UNLOAD HOIST1 CRATE1 TRUCK0 DISTRIBUTOR0 )[1.00]
```

```
52.35: (DROP HOIST1 CRATE1 CRATE3 DISTRIBUTOR0 ) [1.00]
```

Die Operatoren, die die gleiche Anfangzeit haben, werden parallel ausgeführt.

6.4. Durative-FF Planer mit zeitlichen Literalen

Um zeitliche Literalen in der Sprachbeschreibung PDDL2.2 zu behandeln, wurde das PERT Scheduling aus dem vorangegangenen Abschnitt erweitert.

Wir setzen in diesem Abschnitt mit der Erweiterung der Parser fort, so dass auch Planen mit zeitlichen Literalen ermöglicht wird. Danach werden einige Vorverarbeitungen vor dem Planen vorgestellt. Abschliessend wird die Berechnung von parallelen Plänen erläutert [9].

6.4.1. Erweiterung der Parser für zeitliche Literalen

Syntaxanalyse des PDDL-Problems

Wir zeigen die Syntaxanalyse des PDDL-Problems mit Hilfe der Backus-Naur-Form(BNF):

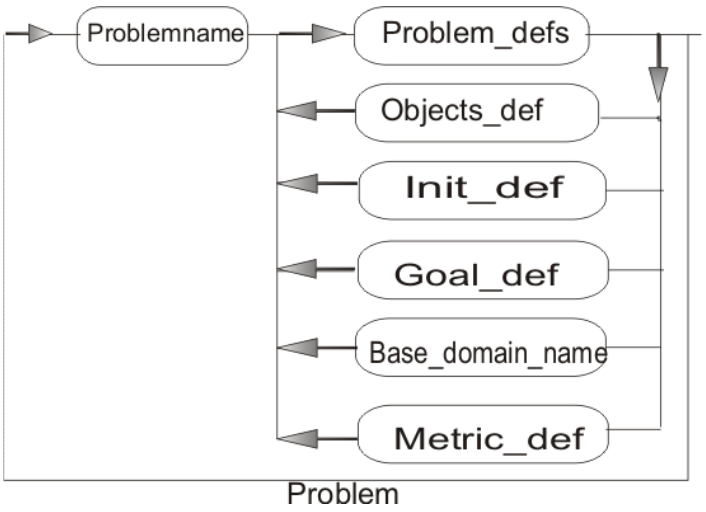


Abbildung 6.13.: Sysntaxanalyse des PDDL-Problems.

Abbildung 6.13 zeigt die Syntaxanalyse des PDDL-Problems. Sie ist in der PDDL-Sprache durch einen Namen und $Problem\_defs$ definiert. $Problem\_defs$ besteht aus vier Komponenten:

1. *Objetct_def*, die die Objekte definiert.
2. *init_def*, die den Initialzustand beschreibt.
3. *goad_def*, die die Definition des Ziels enthält.
4. *metric_def*, die die geforderte Optimierung des Makespans beschreibt.

Zeitliche Literalen

Zeitliche Literalen gehören zu der Spezifikation der PDDL-Beschreibung des Problems. Sie stellen zusätzliche Informationen der Aktionen dar.

Mit der Einführung der zeitlichen Literalen in der PDDL-Sprache wird die Gültigkeit eines Operators durch ein Zeitintervall bestimmt. Dies gibt den Bereich an, innerhalb dessen der Operator existiert, also seine Vorbedingungen wahr sind. Die Operatoren, die von einem zeitlichen Literal abhängen, werden als abstrakte Operatoren bezeichnet und müssen in einer bestimmte Bearbeitungsebene instanziiert werden. Die Gültigkeit einer Aktion kann durch eine Disjunktion von Intervallen definiert werden.

Ein Beispiel aus der Domäne Satellite mit zeitlichen Literale:

```
(at 139.00 (      visible antenna0 satellite0)),
(at 219.04 (not ( visible antenna0 satellite0)))
```

In diesem Problem wird das Zeitfenster [139.00,219.04] für den Fakt *visible antenna*0 *satellite*0 festgelegt. Diese Informationen werden bei der Instanziierung der Operatoren gebraucht, um das Intervall als Gültigkeitsbereich der Aktion, die das Atom *visible antenna*0 *satellite*0 enthält, als Vorbedingungen zu betrachten.

6.4.2. Vorverarbeitung

Extrahieren und Ordnen der zeitlichen Literalen

Nach dem Parsen des Problems und der Domäne haben wir die Methode *extract-Timed-Initial-Literals* implementiert. Ihre Aufgabe besteht darin, den gesamten Baum des initialen Zustands *init-def* zu durchlaufen. Falls sie einen Fakt findet, der mit **at** anfängt und von einer Zahl gefolgt wird, soll dieser aus dem Baum gelöscht und an einer anderen Liste angehängt werden, die später die zeitlichen Literalen darstellt. Vor dem Einfügen des Objekts in die Liste soll zuerst untersucht werden, ob dieser Fakt schon in der Liste vorhanden ist. Wenn ja, soll getestet werden, ob die Anzahl der gelesenen Grenzwerte von Intervallen gerade ist. Wenn ja, (d.h. die beiden Grenzwerten der Intervalle sind komplett belegt) wird ein neues Intervall angelegt. Ist die Anzahl ungerade (d.h. nur ein Grenzwert der Intervall ist belegt) soll das Fakt nicht hinzugefüt. Danach wird die erste Zeichenkette nach der Zahl gelesen werden. Ist sie "not", soll die Zahl als maxtime, sonst als mintime gesetzt werden.

Aufteilung der Vorbedingungen

Unser Ziel ist die Ersetzung der zeitlichen Literalen in den instanziierten Operatoren durch den Schnitt ihrer Zeitintervalle zu realisieren. Wir haben uns dafür entschieden, die Vorbedingungen von der ersten Ebene bis zur Instanziierung in zwei Vorbedingungen aufzuteilen. Der erste Teil enthält alle Vorbedingungen außer denen, die ein zeitliches Literal enthalten. Der zweite Teil enthält die Vorbedingungen, die zeitliche Literalen enthalten. Wir haben dafür eine Routine implementiert, die nach dem Parsen, d.h. in der ersten Ebene, den Vorbedingungsbaum jedes Operators durchläuft und daraus zwei Bäume erstellt, einen Baum für die klassischen Vorbedingungen und einen Baum für die zeitlichen Literal-Vorbedingungen. Anschließend haben wir diese zweigeteilten Vorbedingungen durch die Ebene *Operator* und

Normoperator durchgezogen. In der Ebene *Action*, wo die Instanziierung der Operatoren stattfindet, haben wir die zeitlichen Literal-Vorbedingungen instanziiert.

Berechnung der Aktionsausführungsfenstern

Bei der Ersetzung der zeitlichen Literal-Vorbedingungen wird zuerst geprüft, ob sie in der zeitlichen Literal-Liste enthalten sind. Wenn ja, dann speichert man ihre Zeitintervalle in einer Liste. Jede Intervallliste entpricht einer temporalen Vorbedingung. Wir haben eine weitere Routine implementiert, die aus diesen Intervalllisten eine Disjunktion von Intervallen berechnet und an die Aktion als Ausführungsfenster übergibt.

Pert Scheduling mit Aktionsausführungsfenstern

Procedure Refined-Critical-Path-With-Time-Windows
Eingabe: Sequenz von Aktionen $O_1, ..., O_k$,
Ordnung $\leq_d$
Zeitfenster $[t_{min}(O_i), t_{max}(O_i)]$, i $\in \{1, ..., k\}$
Konflikte $\leq_l^m$ und Offsets Δ_l^m , m, l $\in$ M,
Ausgabe: Makespan

for all i $\in$ {1, . . . , k}
e(O_i) $\leftarrow$ d(O_i) + $t_{min}(O_i)$
for all j $\in$ { 1, . . . , i-1}
if($O_j \leq_d O_i$)
e(O_i) $\leftarrow$ max$\{e(O_i), max\{\Delta_l^m(O_j, O_i) | O_j \leq_l^m O_i, m, l \in M\}\}$
if (e(O_i) > $t_{max}(O_i)$) return ∞
return $max_{1 \leq i \leq k}$ e(O_i)

Der Algorithmus Pert Scheduling mit Aktionsausführungsfenstern in verfeinerter Semantik.

$\Delta_{\vdash}^{\vdash}(O_j, O_i)$	$e(O_j) - d(O_j) + d(O_i) + \epsilon$
$\Delta_{\leftrightarrow}^{\vdash}(O_j, O_i)$	$e(O_j) - d(O_j) + d(O_i)$
$\Delta_{\dashv}^{\vdash}(O_j, O_i)$	$e(O_j) - d(O_j) + \epsilon$
$\Delta_{\vdash}^{\leftrightarrow}(O_j, O_i)$	$e(O_j) + d(O_i)$
$\Delta_{\leftrightarrow}^{\leftrightarrow}(O_j, O_i)$	-
$\Delta_{\dashv}^{\leftrightarrow}(O_j, O_i)$	$e(O_j)$
$\Delta_{\vdash}^{\dashv}(O_j, O_i)$	$e(O_j) + d(O_i) + \epsilon$
$\Delta_{\leftrightarrow}^{\dashv}(O_j, O_i)$	$e(O_j) + d(O_i)$
$\Delta_{\dashv}^{\dashv}(O_j, O_i)$	$e(O_j) + \epsilon$

Abbildung 6.14.: Verschiedene Konflikte in der Fox Long Semantik.

Der Algorithmus Pert Scheduling mit Aktionsausführungsfenstern bekommt als Eingabe eine Folge von Operatoren $O_1, ..., O_k$ mit einer Vorgängerrelation $\leq_d$, eine Menge von

temporalen Konflikten $O_i < O_j$ mit Zeit Offset $\Delta_l^m(O_j, O_i)$ (siehe Abbildung 6.14) und Ausführungsfenster $[t_{min}, t_{max}]$, und gibt einen optimalen parallelen Plan aus, der die gegebene temporale Vorbedingung einhält, nämlich $[t_i, t_i + d(O_i)] \subset [t_{min}(O_i), t_{max}(O_i)]$.

Um Pert Scheduling mit Aktionsausführungsfenstern zu implementieren, haben wir ein paar Routinen entwickelt, die dafür vorgesehenen temporale Konflikte zwischen den Operatoren zu berechnen.

Berechnung der temporalen Konflikte Die *startend*-Funktion überprüft ob ein propositionaler oder nummerischer Konflikt zwischen den Operatoren wie *depend*-Funktion vorliegt. Wenn ja, dann prüft man weiter, ob die temporale Information für den Ausdruck (propositional oder nummerisch) des ersten Operators mit *atstart* bzw. den Ausdruck des zweiten Operators mit *atend* übereinstimmt. Die *endend*-Funktion unterscheidet sich von der *startend*-Funktion, indem sie prüft, ob die temporale Information für den Ausdruck des ersten Operators mit *atend* bzw. den Ausdruck des zweiten Operators mit *atend* übereinstimmt. Die *startstart*-Funktion unterscheidet sich von den beiden Funktionen, indem sie prüft, ob die temporale Information für den Ausdruck des ersten Operators mit *atstart* und bzw. den Ausdruck des zweiten Operators mit *atstart* übereinstimmt. Die *startover*-Funktion prüft weiter, ob die temporale Information für den Ausdruck des ersten Operators mit *atstart* bzw. den Ausdruck des zweiten Operators mit *overall* übereinstimmt. Die *endover*-Funktion prüft weiter, ob die temporale Information für den Ausdruck des ersten Operators mit *atend* bzw. den Ausdruck des zweiten Operators mit *overall* übereinstimmt. Diese Routinen werden benutzt, um aus dem sequentiellen Plan, einen gültigen parallelen Plan zu berechnen.

Als nächstes erläutern wir die Berechnung von Aktionsausführungsfenstern an einem Beispiel aus der Domäne Satellite: Im Gegensatz zu der 2002-Satellite Version hat man in den erweiterten 2004er zeitlichen Literalen eingebracht, in denen die Bilder an den Empfänger gesendet werden können.

```
(:durative-action send_image
        :parameters (?s - satellite ?a - antenna ?d - direction ?m - mode)
        :duration (= ?duration (send_time ?d ?m))
        :condition (and (at start (have_image ?d ?m))
                        (at start (available ?a))
                        (over all (visible ?a ?s)))
        :effect (and (at end (sent_image ?d ?m))
                     (at start (not (available ?a)))
                     (at end (available ?a))) )
```

Die zeitlichen Literalen, die bei der Instanziierung dieses Operators ersetzt werden sollen, sind:

(at 123.00 (visible antenna0 satellite0))
(at 203.04 (not (visible antenna0 satellite0)))
Das bedeutet: antenna0 kann Bilder, die von satellite0 gesendet werden, innerhalb [123.00, 203.04] empfangen.

(at 160.00 (visible antenna1 satellite0))
(at 240.04 (not (visible antenna1 satellite0)))
Das bedeutet: antenna1 kann Bilder, die von satellite0 gesendet werden, innerhalb [160.00, 240.04] empfangen.

Bei der Instanziierung der Operatoren wird jedem Operator ein Zeitintervall zugewiesen, innerhalb dessen der Operator aktiv ist.
Der Operator send_image satellite0 antenna0 erhält als Intervall: [123.00, 203.04].
Der Operator send_image satellite0 antenna1 erhält als Intervall: [160.00, 240.04].
Die anderen Operatoren, die keine temporale Vorbedingung beinhalten, erhalten ein Standardintervall: [0.00, 10000.00].

6.4.3. Berechnung paralleler Pläne

Der Durative-FF Planer erzeugt einen parallelen Plan gemäß der normalen Abhängikeiten, die von der *depend*-Funktion gefunden sind, und der temporalen Konflikte zwischen der Operatoren. Nachdem der Planer den sequentiellen Plan berechnet hat, startet er über die *durative_pert*-Funktion die Berechnung vom parallelen Plan. Zuerst wird die früheste Endezeit des Operators O_i mit $d(O_i) + t_{min}(O_j)$ initialisiert und nachdem der Wert e(O_i) gesetzt worden ist, wird überprüft, ob $e(O_i) < t_{max}(O_i)$ ist. Wenn diese Bedingung erfüllt ist, wird mit der Berechnung von e(O_{i+1}) fortgefahren. Wenn nicht, wird die Berechnung beendet, da es in diesem Fall keinen parallelen Plan gibt, der die gegebene temporale Vorbedingung erfüllt.

Berechnung von e(O_{i+1})

Um die frühesten Endezeit des Operators O_{i+1} zu berechnen, wird zuerst geprüft, ob normale Abhängigkeit zwischen O_{i+1} und ein O_j startend von O_1 bis zu O_i vorliegt. Wenn ja, wird zum zweiten Mal geprüft, ob $e(O_j) + d(O_i)$ grösser als $e(O_i)$ ist, wenn ja, wird anhand der Tabelle (siehe Abbildung 6.14) und der vorliegenden temporalen Konflikten der Wert von $e(O_{i+1})$ bestimmt. Wenn mehr als einen Konflikt zwischen ein Operatorpaar gibt, dann berechnen wir den Maximumwert aus den einzelnen Konflikte.

Wenn die frühesten Endzeiten für alle Operatoren des sequentiellen Plans berechnet sind, dann ziehen wir von denen die $d(O_i)$ ab. Dann erhalten wir die frühesten Startzeiten für alle Operatoren. Abschliessend sortieren wir diese Startzeiten anhand des *Bubblesorts*. Dann ist der parallele Plan berechnet, so dass wir einfach die Operatoren sortiert ausgeben. Wenn der Operator O_i mehrere Intervalle als temporale Vorbedingung hat, dann werden aus jedem Operator mehrere Kopien erzeugt, deren Anzahl mit der Anzahl der Intervalle jedes Operators übereinstimmen. Jeder Kopie wird ein Intervall als Ausführungzeit zugewiesen. Danach werden aus dem originalen sequentiellen Plan mehrere sequentielle Pläne erzeugt, indem jede Kopie eines Operators mit allen anderen Kopien der anderen Operatoren kombiniert wird. In diesem Fall handelt es sich nicht nur darum einen parallelen Plan zu finden, sondern den besseren parallelen Plan unter allen sequentiellen Plänen herauszufinden, wenn es einen gibt. Wir haben dafür zwei Routinen implementiert. Als ersten Schritt haben wir den se-

quentiellen Plan in einer Liste gespeichert, da wir die Operatoren des generierten Plans in einer rekursiven Routine aufrufen lassen wollen. Als zweiten Schritt haben wir eine rekursive Routine die sich selbst aufruft und als Argument die Liste der Plan-Operatoren hat. Die ganze Liste wird durchlaufen. In jedem Aufruf wird die ganze Intervallliste des Operators durchlaufen, auf den die Liste gerade zeigt. Hier wird einfach jedes Intervall der Intervalllliste an den Operator übergeben und indirekt als Kopie betrachtet. Wenn das Ende der Liste erreicht ist, dann ist ein kompletter sequentieller Plan (aus Kopien) aufgebaut. Gegenüber dem originalen sequentiellen Plan bleiben die Operatoren gleich, nur die Intervalle ändern sich. Danach berechnen wir den kritischen Pfad. Wenn dieser kleiner als der Alte ist, dann wird der kritische Pfad aktualisiert und der gefundene Plan gespeichert, usw. bis alle möglichen Konbinationen betrachtet werden. Am Ende erhalten wir den besseren kritischen Pfad und den besseren parallelen Plan, wenn es überhaupt einen gibt. Wir haben den Durative-FF Planer auf ein Problem aus der Domäne Satellite mit zeitlichen Literalen zum Laufen bringen lassen. Es hat sich folgenden parallelen Plan ergeben:

```
; MakeSpan 234.69
0.00  :(SWITCH_ON INSTRUMENT1 SATELLITE0) [2.00]
0.00  :(TURN_TO SATELLITE0 GROUNDSTATION2 PLANET4) [58.98]
58.99 :(CALIBRATE SATELLITE0 INSTRUMENT1 GROUNDSTATION2) [5.54]
59.00 :(TURN_TO SATELLITE0 PLANET3 GROUNDSTATION2) [17.26]
76.26 :(TAKE_IMAGE SATELLITE0 PLANET3 INSTRUMENT1 INFRARED0) [7.00]
83.26 :(TURN_TO SATELLITE0 PLANET4 PLANET3) [30.65]
113.91:(TAKE_IMAGE SATELLITE0 PLANET4 INSTRUMENT1 INFRARED0) [7.00]
120.91:(TURN_TO SATELLITE0 PHENOMENON5 PLANET4) [15.48]
136.39:(TAKE_IMAGE SATELLITE0 PHENOMENON5 INSTRUMENT1 IMAGE2) [7.00]
143.39:(TURN_TO SATELLITE0 PHENOMENON6 PHENOMENON5) [36.47]
160.00:(SEND_IMAGE SATELLITE0 ANTENNA1 PLANET3 INFRARED0) [5.14]
160.00:(SEND_IMAGE SATELLITE0 ANTENNA1 PLANET4 INFRARED0) [8.38]
160.00:(SEND_IMAGE SATELLITE0 ANTENNA1 PHENOMENON5 IMAGE2) [16.52]
179.86:(TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT1 INFRARED0) [7.00]
179.86:(SEND_IMAGE SATELLITE0 ANTENNA1 PHENOMENON6 INFRARED0) [16.25]
186.86:(TURN_TO SATELLITE0 STAR7 PHENOMENON6) [29.32]
216.18:(TAKE_IMAGE SATELLITE0 STAR7 INSTRUMENT1 INFRARED0) [7.00]
216.18:(SEND_IMAGE SATELLITE0 ANTENNA1 STAR7 INFRARED0) [18.51]
```

6.5. Visualisierung und Manipulation der Abhängigkeiten zwischen den Operatoren

Wir haben uns für eine in der Markup-Sprache XML formulierte Spezifikation entschieden, denn die Verwendung von XML als Quellformat bietet uns die Möglichkeit, die Abhängigkeiten der Operatoren graphisch darzustellen. Es wird nach der Bestimmung der Abhängigkeiten zwischen den Operatoren eine XML-Datei generiert.

6.5.1. Entwurf und Generierung der XML-Datei

Um unser DTD für die XML-Datei für die Abhängigkeiten zwischen den Operatoren zu entwerfen, haben wir die Elemente in der Verschachtelung übernommen, genau so wie sie in der Dokumenttyp-Definition stehen. Dies wird im folgenden Beispiel veranschaulicht:

```
<?xml version= "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE   Durative-FF-Depend    SYSTEM      "Depend.dtd" >
<Durative-FF-Depend>
       <Operator>
            <Name> (CT A1 M1 L1)  </Name>
                <Depend>
                    <On\_Operator>
                       <Name> (TRM A1 M1 L1) </Name>
                       <depend_art>   endstart    </depend_art>
                    </On_Operator>
                </Depend>
        </Operator>
        <Operator>
            <Name>(AM A1 M1 L1)</Name>
                 <Depend>
                    <On_Operator>
                       <Name>(CT A1 M1 L1) </Name>
                       <depend_art> endstart </depend_art>
                    </On_Operator>
                 </Depend>
        </Operator>
</Durative-FF-Depend>
```

Die obige wohlgeformte XML-Instanz soll eine konkretes Beispiel der Abhängigkeit zwischen den Operatoren für Durative-FF sein. Wenn man von diesem Beispiel ausgeht, erhält man unmittelbar die folgende DTD:

```
<!ELEMENT   OPERATOR  ( NAME , DEPEND* ) >
<!ELEMENT   NAME  ( # PCDATA ) >
<!ELEMENT   DEPEND( ON_OPERATOR)*>
<!ELEMENT ON_OPERATOR ( NAME , DEPEND_ART+ )* >
<!ELEMENT DEPEND_ART ( #PCDATA ) >
```

Nach dem Aufruf des Editors "Pollo"kann man sich die Abhängigkeiten der Operatoren ansehen. Ausserdem darf man die Abhängigkeitsart manipulieren, d.h. neue Abhängigkeiten einfügen, eine bereits vorhandene löschen. Nachdem der Benutzer die neue Konfiguration gespeichert und den Pollo-Editor geschlossen hat, wird der XML-Parser aufgerufen, der die neuen Abhängigkeiten zwischen den Operatoren im Dokument "Depend.xml" liest, und die enthaltenen Informationen in Form eines Baums zur Verfügung stellt.

6.5.2. XML-Parser

Die Aufgabe des Parsers besteht darin, das XML-Markup durchzulesen und Daten in einer Datenstruktur zu Verfügung zu stellen. Dies passiert mit Hilfe von sogenannten Tokens. Ein Token kann Start-Tag eines Elements, Ende-Tag eines Elements, Zeichenkette oder irgendeine Auszeichnung sein, die einen Wechsel zwischen den Bereichen des Dokuments kennzeichnet. Es wurde einen XML-Parser mit Hilfe des Tools Flex und Bison implementiert. Dafür haben wir eine Datei für Lexer und eine Datei für Yacc implementiert. Nun folgen einige Zeilen aus der Spezifikation des Lexers der XML, die für die lexikalische Analyse der XML-Datei notwendig sind:

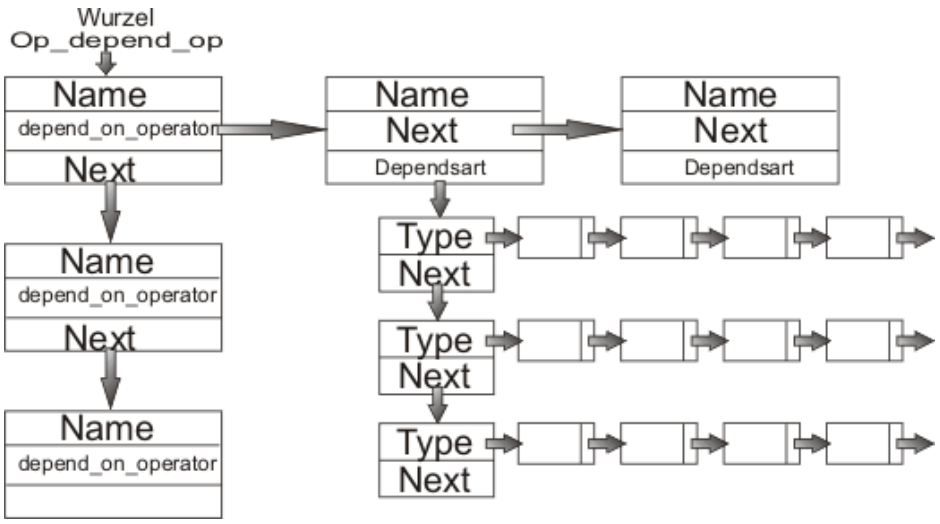
```
 {d}{e}{f}{i}{n}{e} { return(DEFINE_TOK); }
 "<"{d}{u}{r}{a}{t}{i}{v}{e}"-"{f}{f}"-"{d}{e}{p}{e}{n}{d}">"
          { return(OPEN_DURATIVE_FF_DEPEND_TAG); }
 "<""/"{d}{u}{r}{a}{t}{i}{v}{e}"-"{f}{f}  "-"{d}{e}{p}{e}{n}{d}">"
          {return(CLOSE_DURATIVE_FF_DEPEND_TAG);}
"<"{o}{p}{e}{r}{a}{t}{o}{r}">"
          { return(OPEN_OPERATOR_TAG); }
 "<""/"{o}{p}{e}{r}{a}{t}{o}{r}">"
          { return(CLOSE_OPERATOR_TAG); }
 "<"{d}{e}{p}{e}{n}{d}">"  { return(OPEN_DEPEND_TAG); }
"<""/"{d}{e}{p}{e}{n}{d}">"  { return(CLOSE_DEPEND_TAG ); }
"<"{o}{n}"_"{o}{p}{e}{r}{a}{t}{o}{r}">"
          { return (OPEN_ON_OPERATOR_TAG); }
"<""/"{o}{n}"_"{o}{p}{e}{r}{a}{t}{o}{r}">"
          { return (CLOSE_ON_OPERATOR_TAG); }
"<"{d}{e}{p}{e}{n}{d}"_char34{a}{r}{t}">"
          { return (OPEN_DEPEND_ART_TAG) ; }
 "<""/"{d}{e}{p}{e}{n}{d}"_"{a}{r}{t}">"
          { return (CLOSE_DEPEND_ART_TAG); }
```

Die Yacc-Datei für den Scanner der XML-Datei sieht wie folgt aus:

```
... Definitionen ...
%%
... Regeln ...
%%
... Funktionen ...
```

Unter Definitionen haben wir die Vereinbarungen definiert, die Bison bei der Programmerzeugung benutzt, z.b. die Definition von Variablen, Deklarationen der Token. Unter Regeln stehen Grammatikregeln in BNF-Notation, wobei den Regeln Aktionen zugeordnet sein können. Es wurde eine Funktion $load\_xml\_file$ implementiert, die die Eingaben von yacc festlegt. Beim Übersetzen des Programms wird die Datei *scan-xml.tab.c* automatisch durch

Bison generiert. Diese Datei ist für die Kommunikation des lex und yacc notwendig. Bison ist ein Parsergenerator, der eine Eingabe einliest und daraus ein C-Quellcode-Programm erzeugt. Dieses C-Programm muss dann von einem C-Compiler in ausführbaren Code übersetzt werden. Das Programm bearbeitet die Eingabe gemäß der Grammatik in BNF-Notation und der zugeordneten Aktionen in der Bison-Eingabedatei.



Wenn die Grammatik vom Parser erkannt wird, dann erzeugt das Programm von der Struktur des XML-Dokuments eine Baumstruktur, die für Durative-FF zur Verfügung gestellt wird. In Abbildung 6.5.2 steht auf der linken Seite die Liste der Operatoren. Jeder Operator enthält einen Zeiger auf einer Liste der Operatoren, von denen er abhängig ist, falls solche Abhängigkeiten vorhanden sind. Wenn ein Operator von einem anderen Operator abhängig ist, können wir mit Hilfe der Liste "Dependsart", die Art der Abhängigkeiten herausfinden. Falls diese Liste leer ist, ist die Abhängigkeit normal.

6.5.3. Berechnung des neuen parallelen Plans nach den manuellen Änderungen

In disem Abschnitt beschreiben wir den Ablauf der Aktualisierung der neuen Abhängigkeiten der Operatoren nach den manuellen Änderungen dieser durch den Benutzer. Um dies zu realisieren, haben wir uns dafür entschieden, mit zwei neuen Strukturen zu arbeiten. Eine Struktur für XML-Operatoren, d.h. die Operatoren, die in der XML-Datei dargestellt sind und eine Struktur für die Operatoren mit ihrer Aktionszeitdauern und ihren Zeitintervallen, die im sequentiellen Plan generiert sind. Ausserdem haben wir zwei Listen erstellt, eine mit der ersten Struktur und die andere mit der zweiten Struktur. Weiterhin haben wir eine Such-Methode zur Verfügung gestellt, die später in der *user-pert*-Funktion verwendet wird. Diese Methode nimmt als Argument einen XML-Operator und sucht mit seinem Namen in der Liste der generierten Operatoren nach dem Operator, der den gleichen Namen hat. Durch diesen Namen werden die Informationen über die Aktionszeitdauer und das Zeitintervall geholt. Um die Informationen über die neuen Abhängigkeiten zwischen den Operatoren zu aktualisieren, schrieben wir zwei weitere Routinen. Die erste Routine prüft, ob zwei vorhandene

XML-Operatoren voneinander abhängig sind. Wenn ja, wird nach der Art der Abhängigkeit zwischen den beiden Operatoren gefragt. Um dies zu realisieren, schrieben wir eine Methode, die die Art der Abhängigkeiten zwischen den beiden Operatoren liefert. An dieser Stelle brauchten wir eine weitere Routine, die überprüft, ob eine bestimmte Abhängigkeitsart in der Liste der vorhandenen Abhängigkeitsarten enthalten ist. Nachdem wir alle Informationen über die neuen Abhängigkeiten der Operatoren zur Verfügung gestellt hatten, haben wir den sequenziellen Plan an die *user-pert*-Funktion übergeben und damit einen neuen parallelen Plan erzeugt.

6.6. Bewertung der Leistung des Durative-FF Planers

In diesem Abschnitt beschreiben wir die Leistung des Durativ-FF Planer im Vergleich mit anderen modernen Planern. Wir haben fünf Diagramme erstellt. In der x-Achse stehen die Probleme. Sie werden durch ihre Nummern angegeben. Für die vier ersten Diagramme steht der Makespan in der y-Achse. Für das letzte Diagramm steht die Planerzeugungszeit in Milliseconds in der y-Achse.

Abbildung 6.15 veranschaulicht einen Vergleich zwischen Durativ-FF, LPG_TD, und TILSAPA Planern auf der Domäne UMTS mit zeitlichen Literalen. Der Durative-FF Planer liefert im Vergleich mit anderen Planern immer einen guten Makespan.

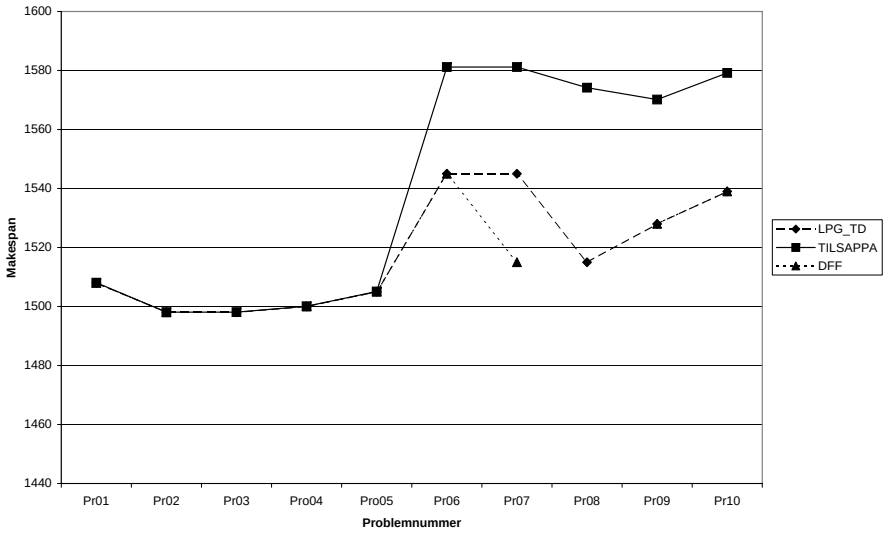


Abbildung 6.15.: Umts mit zeitlichen Literalen: Makespan

Abbildung 6.16 zeigt einen Vergleich zwischen Durativ-FF, LPG_TD, und SGPLAN Pla-

nern auf der Domäne Satellite mit zeitlichen Literalen. Der Durative-FF Planer liefert im Vergleich mit anderen Planern keinen guten Makespan. Aufgrund der Überschreitung der gegebenen Zeitintervalle hat Durative-FF nicht alle Probleme erfolgreich gelöst.

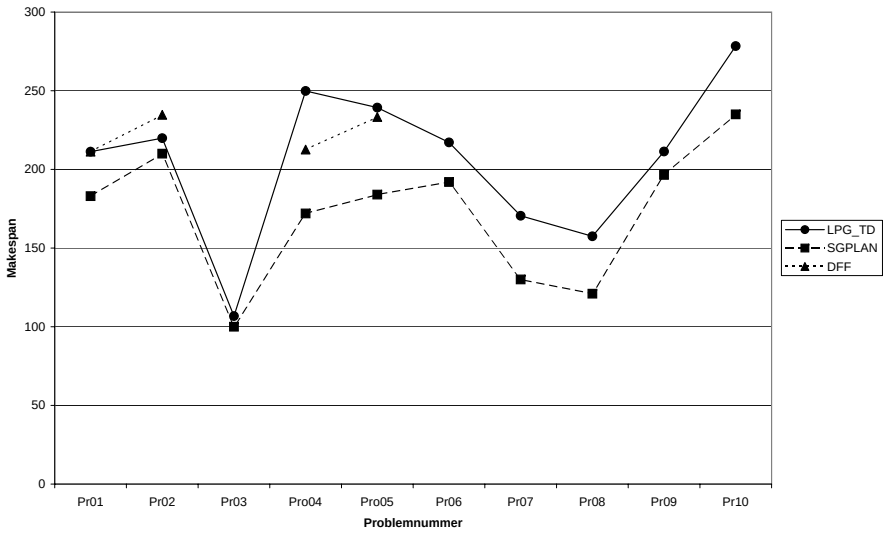


Abbildung 6.16.: Satellite mit zeitlichen Literalen: Makespan

Abbildung 6.17 zeigt einen Vergleich zwischen Durativ-FF, LPG_TD, und CRIKEY Planern auf der Domäne temporal Airport. Der Durative-FF Planer leistet im Vergleich mit den anderen Planern eine gute Arbeit. Er hat alle Probleme erfolgreich gelöst und einen guten Makespan geliefert.

Abbildung 6.18 stellt einen Vergleich zwischen Durativ-FF, LPG_TD und SGPLAN Planern auf der Domäne Airport mit zeitlichen Literalen dar. Der Duarative-FF hat auch hier alle Probleme erfolgreich gelöst und einen guten Makespan geliefert.

Abbildung 6.19 stellt einen Vergleich für die Planerzeugungszeit zwischen Durativ-FF, LPG_TD und SGPLAN Planern auf der Domäne Airport mit zeitlichen Literalen dar. Der Durative-FF Planer zeigt hier gegenüber anderen modernen Planern einen guten Wettkampf, was die Zeit der Planerzeugung betrifft.

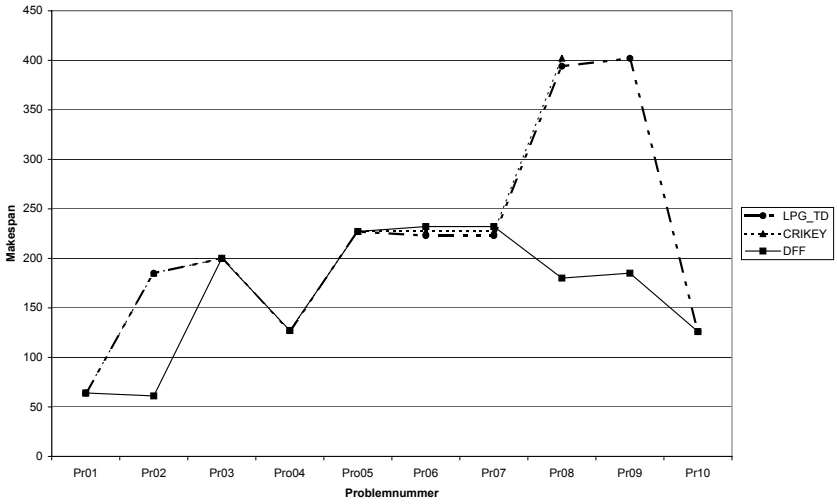


Abbildung 6.17.: Airport temporal: Makespan

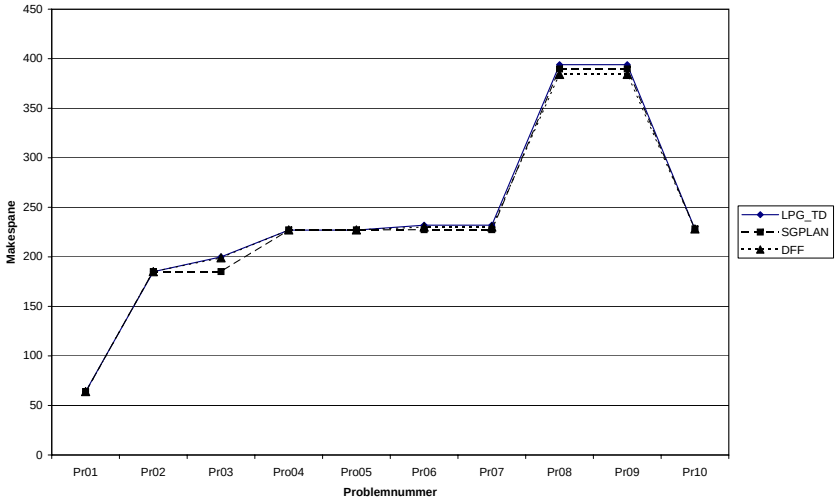


Abbildung 6.18.: Airport time windows: Makespan

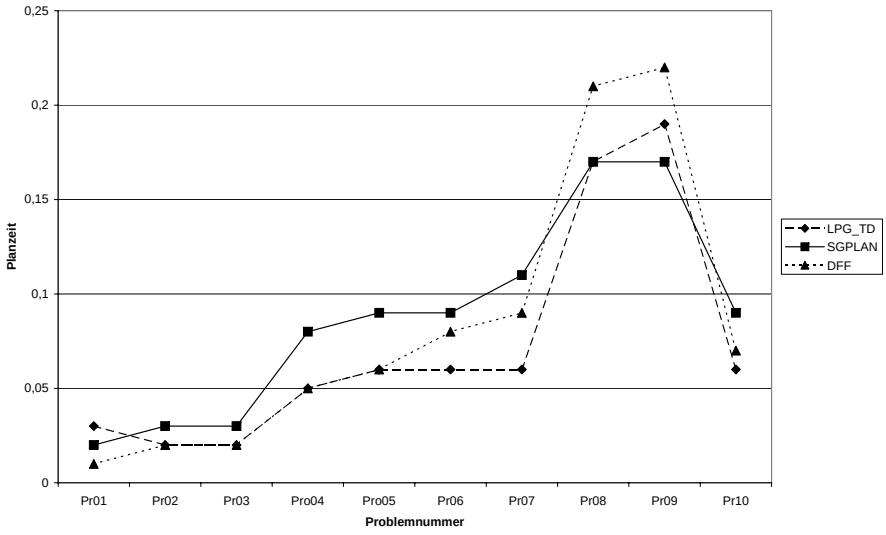


Abbildung 6.19.: Airport time windows: Planerzeugungszeit

Anhang A.

Handbuch der ModPlan Workbench

Arne Wiggers, Rachid Karmouni

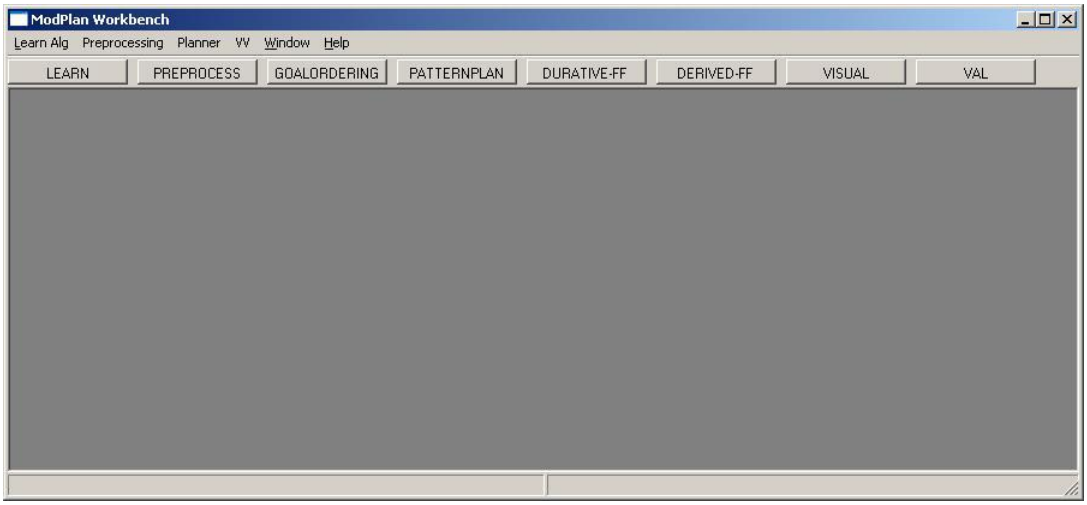


Abbildung A.1.: ModPlan gesamtes Fenster

A.1. Installation

Bevor Sie die ModPlan-Workbench installieren (oder auf Ihren Rechner kopieren), sollten Sie Java und Python schon auf Ihrem Rechner installiert haben und deren Pfade in der Umgebungsvariable gesetzt haben. Falls Sie es noch nicht haben, dann finden Sie die Software in der ModPlan-CD.
Kopieren Sie das ModPlan-Verzeichnis auf Ihren Rechner (Zum Beispiel auf der Desktop).

A.2. Die Verwendung von ModPlan

Das Programm wird gestartet, indem Sie ModPlan.exe in dem ModPlan-verzeichnis starten, und dann bekommen Sie die GUI der ModPlan-Workbench (Abbildung A.1). Da sammeln sich die ganze Teilprogramme der Projektgruppe. Für der Anfang, und damit Sie mit ModPlan arbeiten können, brauchen Sie zwei Dateien: problem.pddl und domain.pddl. Es gibt zwei Möglichkeiten diese Dateien zu erhalten: Lernalgorithmus anwenden und solche Dateien erstellen oder einfach laden, falls die Dateien schon vorhanden sind. Danach können Sie diese Dateien groundieren und zusätzliche Dateien (nutzlich für die Planer) erzeugen. Dies geschieht in der Preprocessingphase. Auf groundierten- oder auch normalen PDDL-Datein können Sie die zur Verfügung stehenden Planer anwenden (Pattern-Plan, Durative-FF, Derived-FF). Nachdem die Planer einen Plan liefern, können Sie diese Pläne validieren und dann in einer benutzerfreundlichen Weise anzeigen lassen (Visualisierung).

A.2.1. LEARN

Beim Laden erscheint einfach ein Dateiauswahlfenster, das die entsprechende PDDL-Datei ladet. Wenn Sie den Learn-Algorithmus anwenden wollen, starten Sie den, indem Sie der LEARN-Knopf drücken. Sie bekommen das LEARN-Fenster in Abbildung A.2 und dann

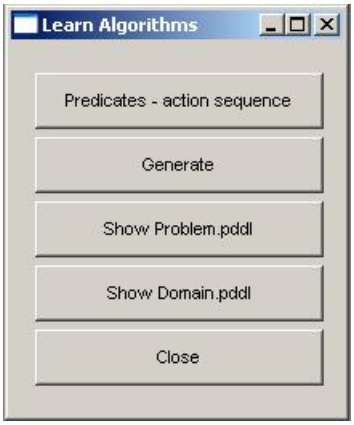


Abbildung A.2.: learn Fenster

Drücken Sie den "predicates - action sequence"-Knopf. Da sollen Sie die Prädikate und eine Aktionsequenz eingeben. In Abbildung A.3 ist ein Beispiel für Prädikate und eine Aktionssequenz (Blocks-World Beispiel).

Von der gegebenen Informationen wird der Algorithmus, mit Hilfe der User die PDDL-Dateien erstellt. Das starten Sie mit einem Knopfdruck auf "Generate" Abbildung A.2. Die Userhilfe besteht darin, zuerst die Objekte zu deren zugehörigen Typen zuzuordenen (Abbildung A.4), die Effekte der Aktionen zu geben (Abbildung A.5), und bei der Generierung der Vorbedingungen zu helfen (Abbildung A.6).

Bei der Effekteingabe und Vorbedingungsgenerierung können Sie eine durative und/oder eine bedingte Aktion wählen (erstellen). Bei durativen Aktionen ist die Duration zu geben, und ein Präfix (at start, at end, over all) für die Prädikate zu wählen. Bei der durativen Aktion erscheint ein Präfixfenster (Abbildung A.7), dieses hilft dem Algorithmus ein Präfix zum entsprechenden Prädikat zu zuweisen.

Nachdem Sie alle gebrauchte Informationen zur Generierung eingegeben haben, erstellt der Algorithmus die PDDL-Dateien (problem und domain). Diese Dateien können Sie , und falls nötig, editieren und ändern.

A.2.2. Preprocessing

In der Preprocessingphase wird das Problem umgeformt ohne den Inhalt zu verändern. Dies vereinfacht für die Planer die Suche nach einem Plan. Dafür gibt es in unserer Workbench drei Technologien, die ungefär dasselbe machen: translate, adl2strips und ground. Translator "translate" erstellt drei Dateien: grounded-domain.pddl, grounded-problem.pddl und problem.sas+. Diese kann man mit den ansprechenden Knöpfen ansehen. (Abbildung A.8). "calculate objects" dient dazu, die Objekte zu berechnen. Sie können die vorhandenen Objekte anschauen und auch löschen unter "delete objects". Die Abbildung A.9 zeigt das dafür verantwortliche Fenster.

Die Symmetrien können Sie berechnen und anzeigen lassen "show symmetry" (Abbildung A.10).

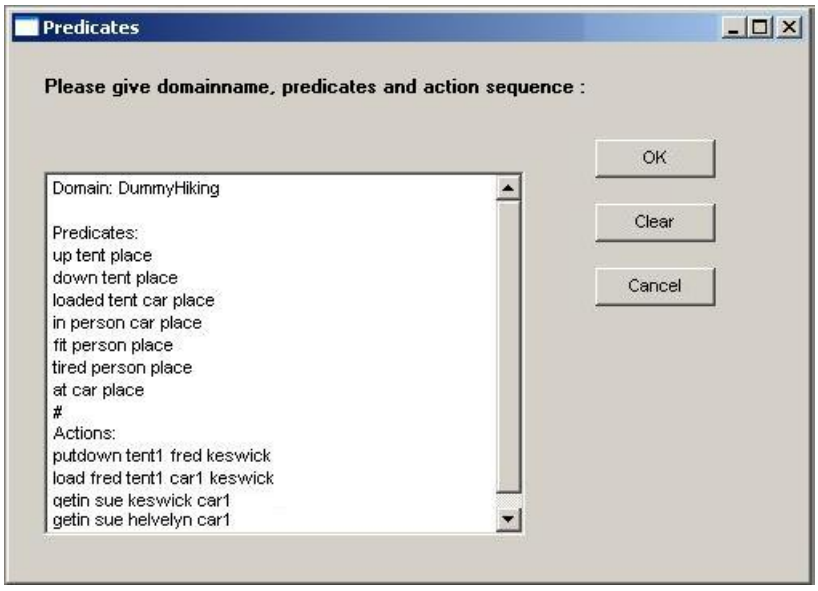


Abbildung A.3.: Prädikaten und Aktionsequenz Eingabe

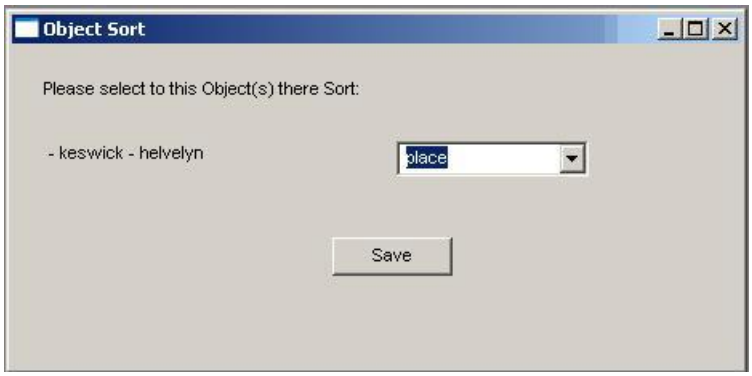


Abbildung A.4.: Zuordnung der Objekte zu Typen

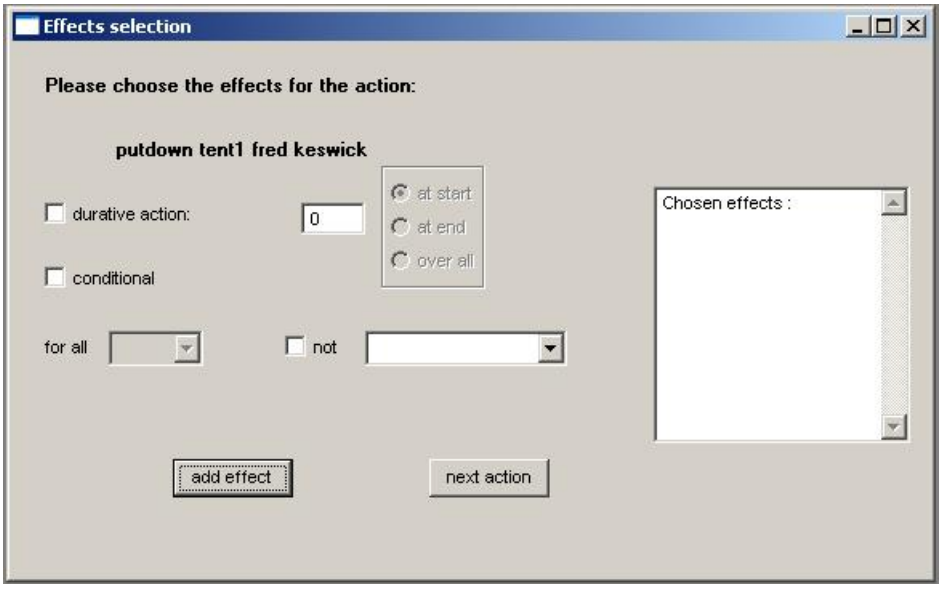


Abbildung A.5.: Effekteingabe

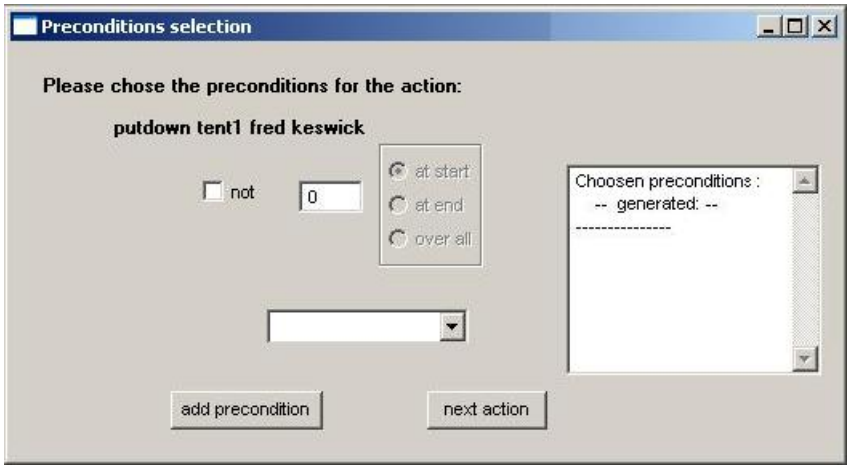


Abbildung A.6.: Bedingungsfenster

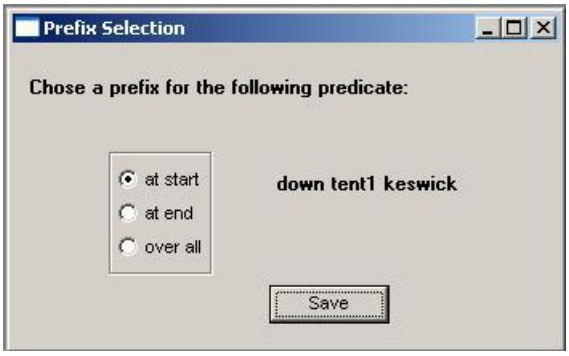


Abbildung A.7.: Präfix Fenster

A.2.3. Planer

Wir sind jetzt soweit, Planer auf unseren Dateien anzuwenden, und Pläne für unsere Probleme zu finden. Dazu haben wir zwei unterschiedliche Planer: Pattern-Plan und Durative-FF (Abbildung A.11).

Pattern-Plan

Pattern-Plan muss nach der Ausführung der Preprocessingphase ausgeführt werden, da er die groundierten Dateien benutzt. Als erstes kommen die unterschiedlichen Optionen. "Explicit Hill-Climbing" und "Relaxed planning heuristic" sind die Standardoptionen. Wenn Sie die Pattern database heuristic wählen, müssen Sie dazu einen Wert für diese Pattern unter "PDB log size" eingeben, sonst wird 0 als Defaultwert genommen. Dazu gehört auch "User interaction", die Ihnen die Möglichkeit gibt, die Pattern Database zu verändern. Diese Pattern Database wird als xml-Datei gespeichert und mit der Option "User interaction" wird der Pollo-XML-Editor gestartet (Abbildung A.12), und das Pattern-Plan der Patternplanner wird pausiert, bis Sie die Änderungen in der XML-Datei speichern und Pollo schliessen.

Durative-FF

Der Durative-FF Planer läuft sowohl auf normalen als auch auf grundierten PDDL-Dateien (Abbildung A.11). Sie können das Problem anhand die groundierte PDDL-Dateien lösen, wenn Sie die Checkbox "Grounded PDDL" wählen. "User interaction" macht das gleiche wie bei Pattern-Plan. Als Heuristik können Sie Scheduling- oder Graphplanheuristik wählen. Der Durative-FF Planer erzeugt eigentlich zwei Lösungen oder Pläne, einen sequentiellen und einen paralellen Plan, die Sie mit einem Knopfdruck ansehen können (Abbildung A.11).

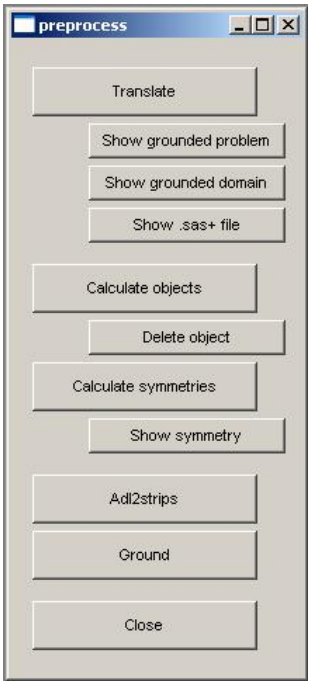


Abbildung A.8.: Preprocessing

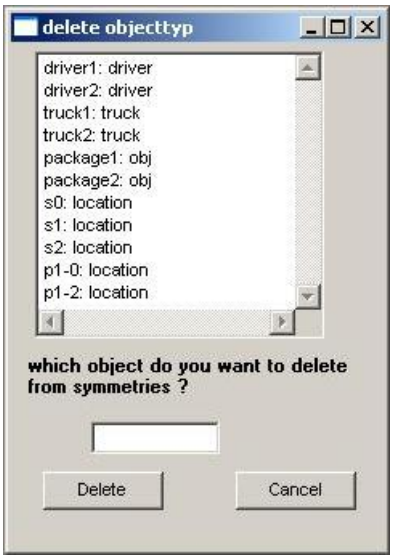


Abbildung A.9.: Objekte ansehen und löschen

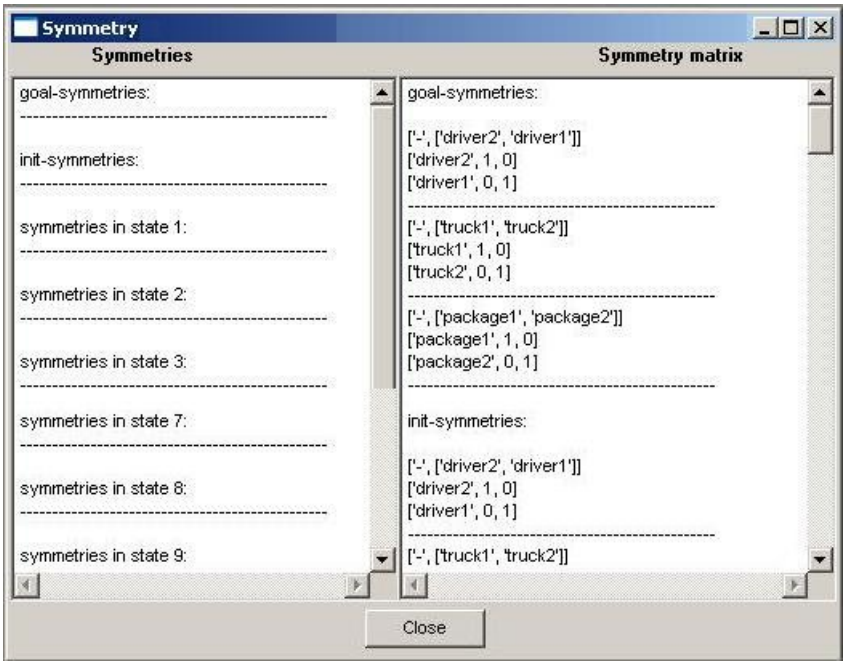


Abbildung A.10.: Symmetrien und Symmetrimatrix

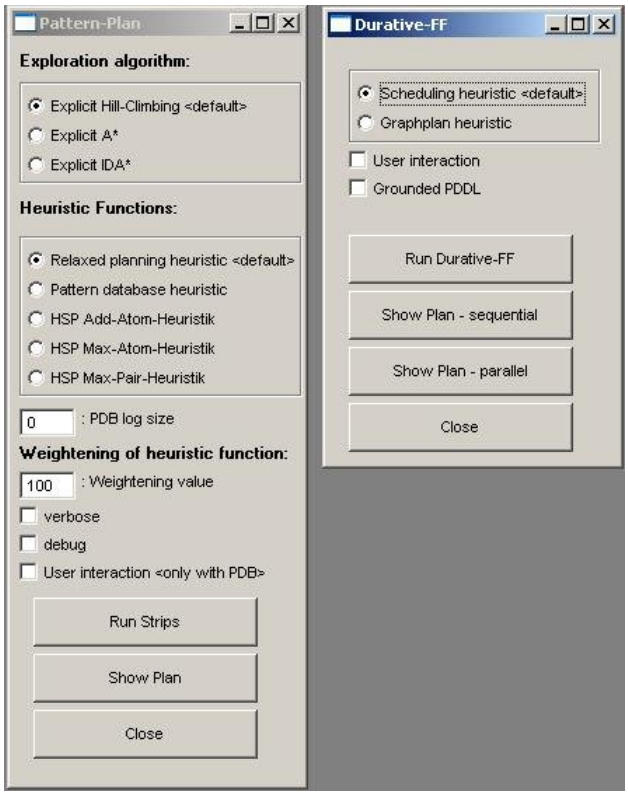


Abbildung A.11.: Fenster der Pattern-Plan und Durative-FF

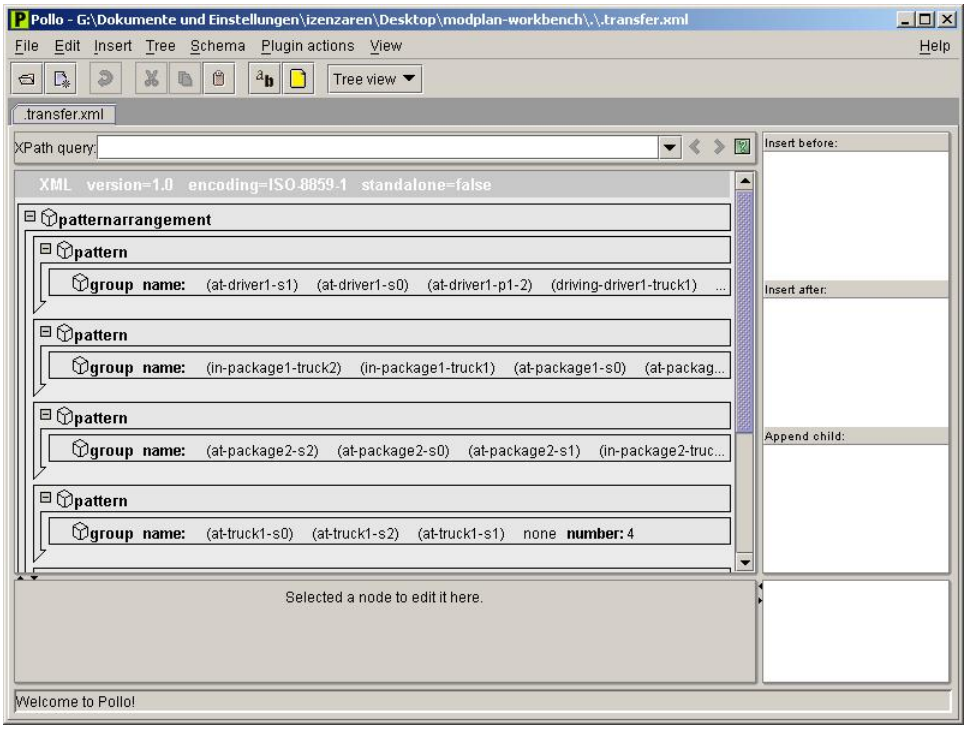


Abbildung A.12.: Der XML Editor pollo

A.2.4. Validierung und Visualisierung

Validierung

Der Validierer validiert einen gegebenen Plan und ergibt als Ergebnis, ob der stimmt oder nicht (Abbildung A.13). Im positiven Fall (Abbildung A.14) erzeugt der Validierer eine Datei (state-sequence), die die durchgelaufenen Zustände enthählt. Diese kann man unter "show state-sequence" anschauen.
Statevis ist auch ein erweiterte Art vom Validierer. Statevis schreibt die Dateien sequence.scene und sequence.run als Balkengraphik. Diese kann man dann unter Vega (Visualisierung) einladen und animiert darstellen.

Visualisierung

VEGA/VEPA veranschaulicht Pläne in der Form eines GANNT Diagramms (Abbildung A.15). Dazu müssen Sie eine Scene-Datei laden. Im Fall der statevis, müessen Sie zuerst die sequence.scene-Datei laden, danach die sequence.run-Datei (die Scene muss vor dem Run geladen werden damit die Animation funktioniert). in Abbildung A.15 ist ein Driverlog-Beispiel zu sehen.

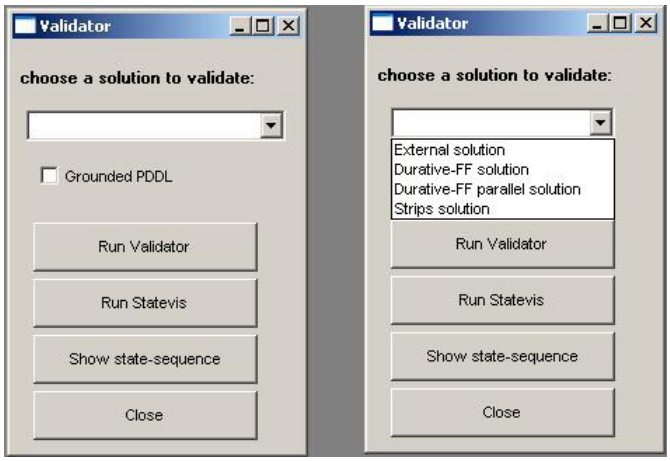


Abbildung A.13.: Das Validatorfenster - mit unterschiedliche Selectionen

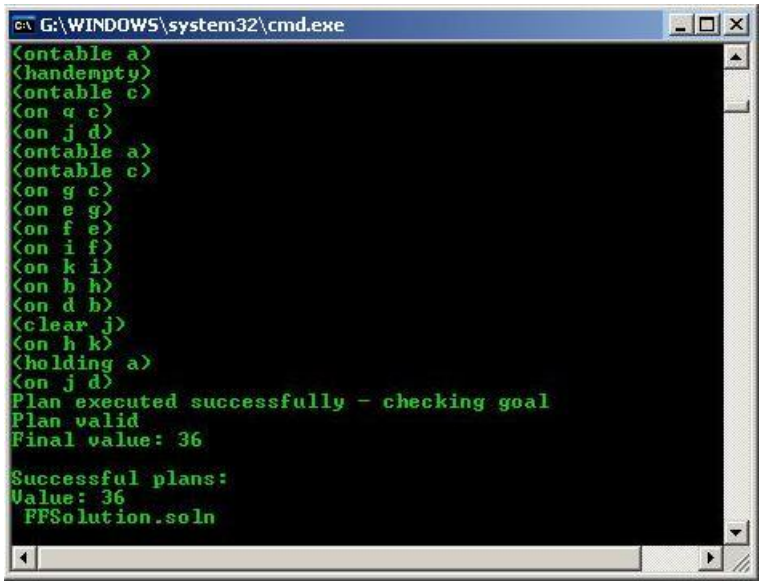


Abbildung A.14.: Die Ausgabe der Validator im positiven Fall

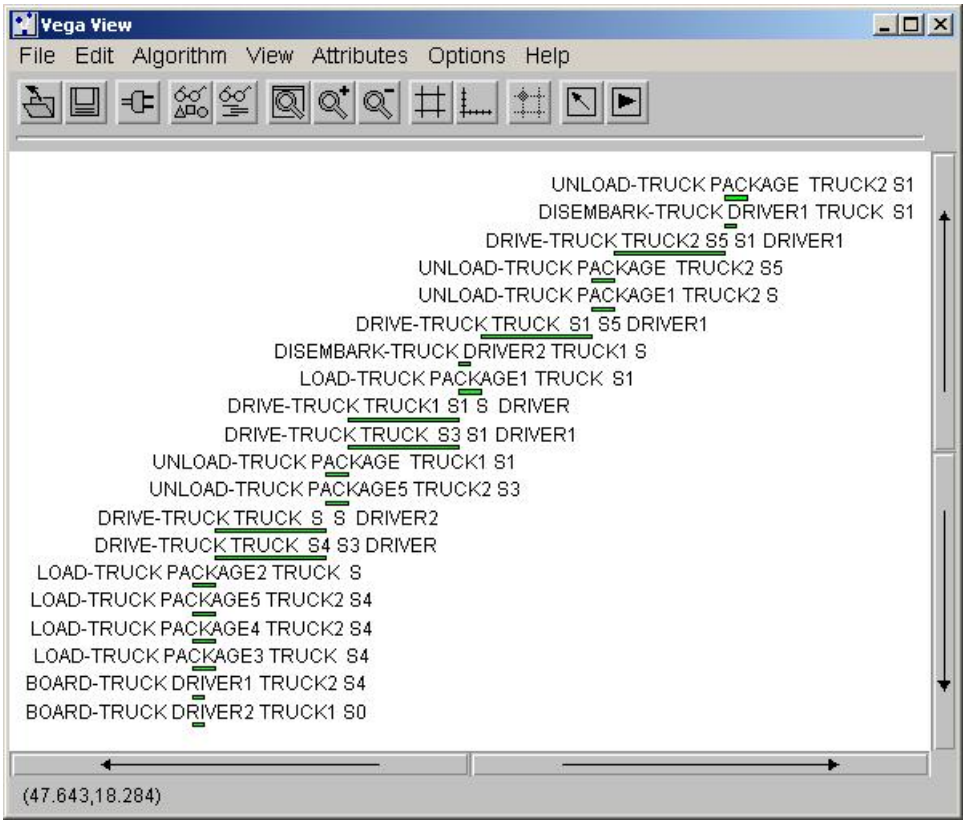


Abbildung A.15.: Das VEGA zur Visualisierung

Abbildung A.16.: Startfenster der GoalOrdering

A.2.5. Goal-Ordering

Das Goalordering Fenster ist in Abbildung A.16 dargestellt. Die GoalOrdering arbeitet auf groundierten PDDL-Dateien, also kommt dies nach der Preprocessingphase. Bei manchen Problemen werden die Abhängigkeiten berechnet und der User wird gefragt ob er diese (einzeln) akzeptiert oder nicht (Abbildung A.17). Dann werden die Zwischenziele berechnet und gelöst (hier wird der durative-FF als Planer benutzt). Die Teilprobleme, Teilpläne sowie der Gesamtplan können unter "partial problems and plans" und "Total Plan" gezeigt werden (Abbildung A.18).

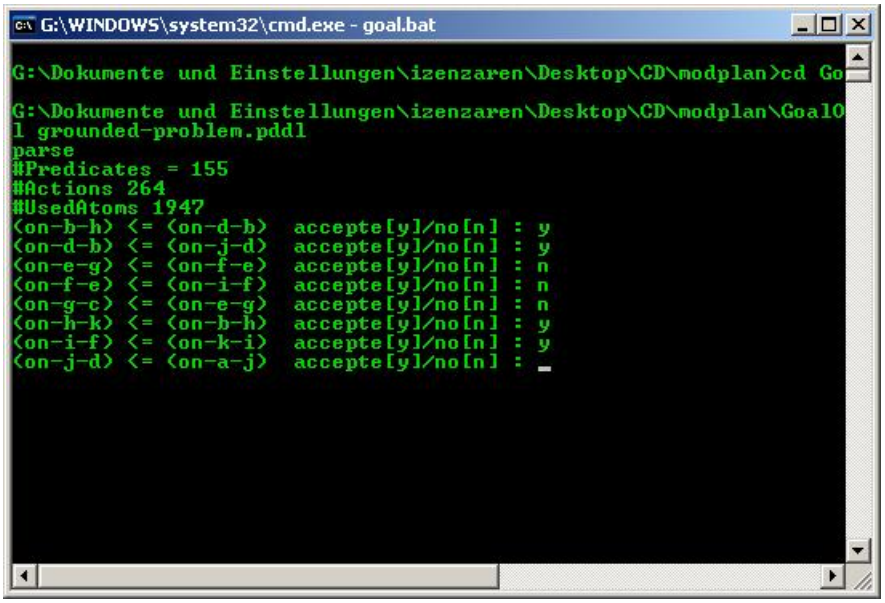


Abbildung A.17.: Beispiel zur Frage nach Abhängigkeiten

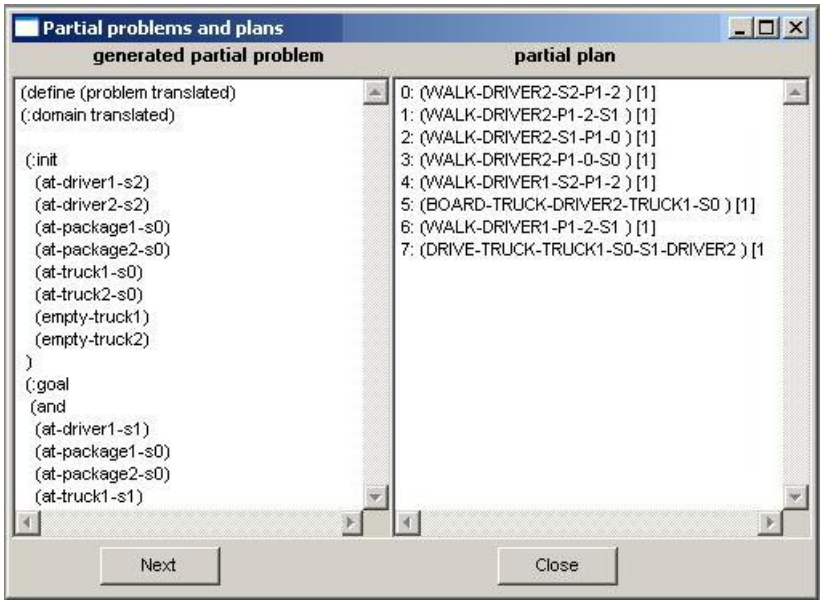


Abbildung A.18.: Anzeigen der Partial Probleme und Pläne

Literaturverzeichnis

[1] Fahiem Bacchus. The AIPS'00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

[2] Christer Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9:99–137, 1998.

[3] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 7(90):281–300, 1997.

[4] Yixin Chen and Benjamin W. Wah. Subgoal partitioning and resolution in planning. In *Proceedings of the International Planning Competition (IPC)*, 2004.

[5] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the European Conference on Planning (ECP)*, pages 13–24, 2001.

[6] Stefan Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research*, 20:195–238, 2003.

[7] Stefan Edelkamp. Extended critical paths in temporal planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 38–45, 2004.

[8] Stefan Edelkamp. Generalizing the relaxed planning heuristic to non-linear tasks. In *Proceedings of the German Conference on Artificial Intelligence (KI)*, 2004. 198-212.

[9] Stefan Edelkamp. *Handlungsplanung – Skript zur Vorlesung.* University of Dortmund, 2004.

[10] Stefan Edelkamp and Malte Helmert. MIPS: The model-checking integrated planning system. *AI Magazine*, 22(3):67–72, 2001.

[11] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, University of Freiburg, 2004.

[12] Stefan Edelkamp, Jörg Hoffmann, Michael Littman, and Hakan Younes, editors. *Proceedings of the International Planning Competition.* JPL, 2004.

[13] Maria Fox and Derek Long. The detection and exploration of symmetry in planning problems. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 956–961, 1999.

[14] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Research (JAIR)*, 20:61–124, 2003.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, München, 1989.

[16] Emmanuel Guéré and Rachid Alami. One action is enough to plan. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 439–444, 2001.

[17] Patrik Haslum, Blai Bonet, and Hector Geffner. Use and value of pattern database heuristics in optimal planning. In *AAAI'05*, 2005.

[18] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 140–149, 2000.

[19] Malte Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 161–170, 2004.

[20] Lars Hildebrand. *Vorlesungfolien zu Vorlesung: Grundlagen und Anwendungen der Computational Intelligence II: Evolutionäre Algorithmen*. University of Dortmund, 2004.

[21] Christoph A. Hipke. *Distributed Visualization of Geometric Algorithms*. PhD thesis, University of Freiburg, 2000.

[22] J. Hoffmann. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[23] Jörg Hoffmann. Extending FF to Numerical State Variables. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 571–575, 2002.

[24] Jörg Hoffmann, Roman Englert, Frederico Liporace, Sylvie Thiebaux, and Sebastian Trüg. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research*, 2005. Submitted.

[25] Jörg Hoffmann and Bernhard Nebel. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[26] Robert.C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 122 – 131, 2004.

[27] Richard Howey and Derek Long. VAL's progress: The automatic validation tool for PDDL2.1 used in the international planning competition. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) – workshop on the Competition*, pages 28–37, 2003.

[28] Xavier Hüe. *Genetic Algorithms for Optimisation – Background and Applications*. The University of Edinburgh, Edinburgh Parallel Computing Centre, 1997.

[29] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI) and the Innovative Applications of Artificial Intelligence Conference (IAAI)*, pages 1194–1201, 1996.

[30] Henry A. Kautz and Joachim P. Walser. State-space planning by integer optimization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI) and the Innovative Applications of Artificial Intelligence Conference (IAAI)*, pages 526–533, 1999.

[31] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 2000. Submitted.

[32] Richard E. Korf and Ariel Felner. *Chips challenging champions: games, computers and Artificial Intelligence*, chapter Disjoint pattern database heuristics, pages 13–26. Elsevier Science Publishers Ltd., 2002.

[33] Derek Long and Maria Fox. The 3rd international planning competition: Overview and results. *Journal of Artificial Intelligence Research*, 20:1–59, 2003. Special issue on the 3rd International Planning Competition.

[34] T. Lee McCluskey, N. Elisabeth Richardson, and Ron M. Simpson. An interactive method for inducing operator descriptions. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 121–130, 2002.

[35] Drew McDermott. The 1998 AI Planning Competition. *AI Magazine*, 21(2):35–55, 2000.

[36] Mohammed Nazih. Planen mit Kausalgraphen. In Miriam Bützken, Andrea Matuszewski, Rachid Karmouni, Michael Nelskamp, Arne Wiggers, Abdelaziz Elalaoui, Khalid Lahiane, Mohammed Nazih, Kenneth Kahl, and Roman Klinger, editors, *Zwischenbericht der PG 463*, 2004.

[37] Ioannis Refanidis, Ioannis Vlahavas, and Lefteri Tsoukalas. On determining and completing incomplete states in strips domains. In *Proceedings of the International Conference on Information, Intelligence and Systems (IEEE)*, pages 289–296, 1999.

[38] Jussi Rintanen. Symmetry reduction for SAT representations of transition systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 32–40, 2003.

[39] Matthew Wall. *GAlib – A C++ Library of Genetic Algorithm Components*. Massachusetts Institute of Technology, 2005.

[40] Ingo Wegener. *Komplexitätstheorie*. Springer, 2003. (in German).

[41] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples with incomplete knowledge. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.