Handlungsplanung

Dr. M. Helmert, Prof. Dr. B. Nebel
G. Röger
Sommersemester 2010

Universität Freiburg
Institut für Informatik

Projekt 1: Erster Kontakt mit JavaFF[1]

Abgabe: 11. Juni 2010

JavaFF [1] ist eine Reimplementierung des FF-Planungssystems [3], das Ihnen bereits auf dem ersten Übungsblatt begegnet ist. JavaFF wurde an der University of Strathclyde in Glasgow speziell für die Lehre entwickelt und zeichnet sich dadurch aus, dass es leicht zu erweitern ist und die verschiedenen Aspekte wie Suchalgorithmus oder Heuristik relativ sauber voneinander getrennt sind.

1 Erste Schritte

Der Java-Quelltext von JavaFF (bzw. einer von uns nochmals modifizierte Version) und einige Beispielplanungsprobleme und -domänen sind auf der Kursseite erhältlich:

`http://www.informatik.uni-freiburg.de/~ki/teaching/ss10/aip/JavaFF_p1.tar.gz`

Laden Sie das Archiv (z.B. in Ihr Homeverzeichnis) herunter und entpacken Sie es. Falls Sie mit Linux arbeiten, geht das mit den folgenden Befehlen:

```
cd
wget http://www.informatik.uni-freiburg.de/~ki/teaching/ws0809/aip/JavaFF_p1.tar.gz
tar -zxf JavaFF.tar.gz
```

Das Archiv entpackt in zwei Verzeichnisse: `examples` enthält die Beispielplanungsprobleme und `javaff` den Java-Quelltext und eine vorkompilierte Version. JavaFF kann mit dem folgenden Befehl selbst kompiliert werden (eventuell müssen Sie zuvor den classpath anpassen):

```
javac javaff/JavaFF.java
```

Jetzt können Sie JavaFF bereits auf einem der Beispielprobleme laufen lassen:

```
java javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile01
```

Als Ergebniss sollten Sie einen Plan für das Problem bekommen. Falls Sie das Programm noch auf anderen Domänen und Problemen laufen lassen wollen, stehen Ihnen die folgenden Beispieldomänen zur Verfügung:

- **driverlog** – eine Logistikdomäne, bei der einige Pakete, Lastwagen und Fahrer von ihrer Ausgangsporition zu ihrer angegebenen Zielposition gebracht werden sollen.
- **rovers** – eine Domäne, die die Aktivitäten eines Erkundungsroboters auf einem zu erforschenden Planeten modelliert. Ziel ist es, wissenschaftliche Daten zu sammeln (Bilder, Bodenproben, etc.) und die Daten zur Erde zu übermitteln.
- **depots** – eine Domäne zur Organisation eines Warenlagers, bei der Paletten bewegt werden müssen, so dass sie am Ende wie angegeben gelagert werden.

[1] Die Projektbeschreibung basiert teilweise auf dem JavaFF-Booklet (siehe `http://personal.cis.strath.ac.uk/~ac/JavaFF`)

Sie können JavaFF analog zu obigen Aufruf auf diesen Problemen laufen lassen: Hierzu übergeben Sie die entsprechende `domain.pddl` als erstes Argument und die Problemdatei als zweites. Die Probleme sind in PDDL kodiert. Falls Sie sich die Beispiele also genauer ansehen wollen, können Sie sie einfach mit einem Texteditor öffnen.

Anmerkung: Abhängig davon, wieviel Speicher die Java VM in der Defaulteinstellung verwenden darf, kann es passieren, dass ihr bei der Ausführung von JavaFF der Speicher ausgeht. Verwenden Sie in diesem Fall die Option `-Xmx512m` als ersten Parameter von `java`, um ihr mehr Speicher zuzuteilen, wobei höchstens 512 Megabytes erlaubt sind. Zum Beispiel:

```
java -Xmx512m javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile12
```

Aufgabe 1: Daten sammeln

0.3 Punkte

Einen wichtigen Aspekt wissenschaftlicher Arbeit (und Informatik bildet hier keine Ausnahme) bilden Experimente zur Überprüfung von Hypothesen. Im Bereich der Handlungsplanung ist die Hypothese normalerweise, dass bestimmte Veränderungen an einem Planungssystem dafür sorgen, dass Pläne schneller gefunden werden. Experimente haben daher oft die Form, dass gemessen wird, wie lange es vor der Veränderung dauert, Pläne zu generieren, und wie lange nach der Modifikation. Diese Daten werden dann verglichen, um zu beurteilen, ob diese Änderung eine gute Idee war. Ihre erste Aufgabe ist daher, Daten darüber zu sammeln, wie lange JavaFF in der Originalfassung benötigt, um die Probleme der Beispieldomänen zu lösen. Wir werden diese Ergebnisse bei späteren Aufgaben dann als Benchmark verwenden.

Die beiden Domänen, für die wir uns hier interessieren, sind **rovers** und **driverlog**. Lassen Sie JavaFF auf den ersten zehn Problemdateien (`pfile01` bis `pfile10`) laufen und notieren Sie die Planungszeit, die von JavaFF nach dem Plan ausgegeben wird.

Hinweis: Um sich nur die relevanten Daten auf dem Bildschirm ausgeben zu lassen, können Sie (zumindest als Linux-Nutzer) die Ausgabe des Planers wie folgt durch `grep` leiten:

```
java javaff.JavaFF examples/driverlog/domain.pddl \
     examples/driverlog/pfile01 | grep "Planning Time"
```

Damit wird nur die Zeile mit der Planungszeit ausgeben, da nichts sonst dem regulären Ausdruck “Planning Time”, der `grep` übergeben wird, entspricht.

2 Ein Blick unter die Motorhaube

Da Sie JavaFF nun am Laufen haben und bereits einige Daten gesammelt haben, ist es Zeit für einen ersten Blick auf den Quellcode.

2.1 Die Hauptklasse

Einen guten Einstiegspunkt bildet die Datei `javaff/JavaFF.java`, die den Quellcode der Hauptklasse enthält. Ein Großteil der Datei enthält – für uns uninteressante – Aufrufe von grundlegendem Code, wie das Laden und Parsen des Problems und der Domäne. Der interessante Teil findet sich gegen Ende der Datei: die Methode `performSearch`. Öffnen Sie die Datei in ihrem Lieblingstexteditor und lesen Sie die Methode zunächst einmal vollständig durch, um sich mit dem Inhalt auf einem hohen Abstraktionsniveau vertraut zu machen. Danach sind Sie bereit für die folgenden genaueren Erklärungen.

2.1.1 Aufruf der EHC-Suche

Zunächst versucht `performSearch` eine Enforced-Hill-Climbing-Suche (EHC). Enforced-Hill-Climbing ist eine spezielle lokale Suchmethode, die Sie später in der Vorlesung noch kennenlernen werden. Im Code sieht das folgendermaßen aus:

```
    Heuristic heuristic = new FFHeuristic();

    EnforcedHillClimbingSearch EHCS =
        new EnforcedHillClimbingSearch(initialState, heuristic);

    EHCS.setFilter(new HelpfulFilter(heuristic));

    State goalState = EHCS.search();
```

In der ersten Zeile wird ein Objekt erstellt, das die zu verwendende Heuristik repräsentiert. Die zweite Zeile erstellt ein EHC-Suchalgorithmenobjekt, wobei neben dem Anfangszustand auch die Heuristik übergeben wird.

Während der Suche werden Zustände expandiert, indem Aktionen auf den Zustand angewandt werden. Welche Aktionen hierbei genau in Betracht gezogen werden, wird in JavaFF durch sogenannte Filter festgelegt. FF verwendet normalerweise zunächst nur sogenannte *helpful actions*. Das sind Aktionen, die während der Heuristikberechnung besonders vielversprechend erscheinen (wie das genau gemacht wird, werden wir später noch in der Vorlesung behandeln). Diese Beschränkung auf die helpful actions geschieht im Code in dem dritten Aufruf durch das Setzen des `HelpfulFilter`. Das Argument gibt dabei an, welche Heuristik zur Identifikation der helpful actions verwendet werden soll.

In der letzten Zeile wird die Suche schließlich mit `search()` gestartet.

2.1.2 Verwendung der Bestensuche

Falls EHC keinen Plan findet (wenn `goalState == null`), geht FF zu einer Bestensuche über. Dabei werden nicht nur die helpful actions verwendet, sondern alle in einem Zustand anwendbaren Aktionen. Der Code sieht folgendermaßen aus:

```
    BestFirstSearch BFS =
    new BestFirstSearch(initialState, heuristic);

    BFS.setFilter(new NullFilter());

    goalState = BFS.search();
```

Wie Sie sehen, ist alles sehr ähnlich zum Aufruf von EHC. Ein Unterschied ist, dass nun ein BestFirstSearch-Objekt erstellt wird. Der zweite Unterschied liegt in dem verwendeten Filter. Im Gegensatz zum `HelpfulFilter` werden mit dem `NullFilter` alle anwendbaren Aktionen bei der Expansion verwendet, nicht nur die vielversprechenden.

2.2 Verwendung von Heuristiken

In der Hauptklasse haben Sie bereits gesehen, dass die zu verwendende Heuristik bestimmt wird, indem man ein passendes Heuristikobjekt erzeugt. Die Heuristikklasse muss das Interface `Heuristic` implementieren, das sie in der Datei `javaff/heuristics/Heuristic.java` finden.

```
public interface Heuristic {
    public BigDecimal getHValue(State state);
}
```

Eine Heuristik muss nur eine einzige Methode `getHValue` implementieren. Diese Methode berechnet für einen bestimmten Zustand den Heuristikwert, also eine Schätzung, wieviele Schritte notwendig sind, um von diesem Zustand zu einem Zielzustand zu kommen.

2.3 Kurzes Luftholen...

Nun sollten Sie ein grundlegendes Verständnis für die folgenden Aspekte von JavaFF haben:

- Die Struktur der `performSearch`-Methode (2.1); insbesondere, wie man unterschiedliche Suchverfahren mit verschiedenen Heuristiken kombinieren kann.
- Die Rolle des Interface `Heuristic`.
- Die Rolle der `Filter` bei der Entscheidung, welche Aktionen bei der Expansion eines Zustands in Betracht gezogen werden sollen.
  - Verhalten des `NullFilter`
  - Verhalten des `HelpfulFilter` (eine grobe Idee reicht)

Falls Sie sich in einem der Punkte unsicher sind, lesen Sie den betreffenden Abschnitt nochmal oder fragen Sie einen der Betreuer. Sie müssen den Code nicht aus der Erinnerung rezitieren können, sondern nur mit den Grundkonzepten vertraut sein. Wenn Sie soweit sind, lesen Sie weiter...

3 Erste eigene Änderungen

Im Laufe der Projekte werden Sie einige Änderungen an JavaFF vornehmen. Wir beginnen mit einigen kleinen Änderungen, mit denen wir den Einfluss der Heuristik und des Filters untersuchen wollen.

Eine Möglichkeit, den Einfluss der Heuristik zu testen, wäre, die FFHeuristik durch eine Heuristik zu ersetzen, die der Suche keinerlei hilfreiche Informationen liefert (zum Beispiel, indem sie für alle Zustände den gleichen Wert zurückgibt). Das Ergebnis entspräche dann einer Suche ohne Heuristik. Mit diesem Vorgehen würde man allerdings für die meisten unserer Beispielprobleme überhaupt keine Lösung mehr finden. Aus diesem Grund wollen wir in dieser Übung die FFHeuristik mit einer anderen, recht einfachen Heuristik vergleichen.

3.1 Goal-Count-Heuristik

Die Goal-Count-Heuristik ist eine sehr einfache Heuristik, die bereits 1971 im Planungssystem *Stanford Research Institute Problem Solver* [2] Verwendung fand. Sie setzt voraus, dass die Zielzustände des Problems durch eine Konjunktion von Literalen definiert sind. Die Goal-Count-Heuristik schätzt den Abstand eines Zustands vom Ziel, indem sie zählt, *wieviele Literale der Zielbeschreibungskonjunktion in diesem Zustand falsch* sind. Sollen im Ziel zum Beispiel die Literale $a, \neg c$ und e wahr sein und im aktuellen Zustand s sind die Literale $a, b, c, \neg d, \neg e$ wahr, so ist $h_{GC}(s) = 2$.

In der Datei `javaff/heuristics/GoalCountHeuristic.java` finden Sie bereits einen Platzhalter für diese Heuristik, bei dem allerdings noch alle relevanten Teile fehlen. Ihre Aufgabe wird es in der nächsten Aufgabe unter anderem sein, diese Heuristik zu vervollständigen.

Hinweis: JavaFF kann nur mit Problemen umgehen, deren Zielbeschreibung eine Konjunktion von Atomen ist. Sie können also immer davon ausgehen, dass dies der Fall ist.

Damit sollten Sie genug über die Goal-Count-Heuristik und JavaFF wissen, um sich an Aufgabe 2 wagen zu können.

Aufgabe 2: Relevanz der Heuristik und des Filters

1.2 Punkte

Wie Sie in Abschnitt 2.1 gesehen haben, verwendet FF eine spezielle Heuristik, die die Suche in die richtige Richtung leiten soll. Zudem beschränkt der HelpfulFilter die Suche zunächst auf die helpful actions. Ziel dieser Aufgabe ist es, den Einfluss dieser beiden Komponenten zu untersuchen.

Sie haben bereits gesehen, wie man bestimmen kann, welcher Filter verwendet werden soll. Außerdem wissen Sie, dass in JavaFF der `HelpfulFilter` und der `NullFilter` bereits implementiert sind.

Die einzige bereits fertig implementierte Heuristik ist die `FFHeuristic`. Um einen Vergleich mit einer anderen Heuristik zu haben, werden Sie in dieser Übung zusätzlich die Goal-Count-Heuristik implementieren.

(a) Implementieren Sie die Goal-Count-Heuristik, indem Sie die Datei `GoalCountHeuristic.java` vervollständigen.

Casten Sie hierzu zunächst den an `getHValue` übergebenen `State` in einen `STRIPSState`. Die in einem STRIPSState `sstate` wahren Atome erhalten Sie mittels:

```
Set actualFacts = sstate.getFacts();
```

Die Atome, die im Ziel wahr sein müssen, erhalten Sie mittels:

```
Set goalFacts = sstate.goal.getConditionalPropositions();
```

Testen Sie anschließend, wieviele Ziel-Literale im aktuellen Zustand noch nicht erfüllt sind und geben Sie diese Anzahl als heuristischen Wert zurück.

(b) Führen Sie zu Aufgabe 1 analoge Experimente durch, um den Einfluss der Heuristik und des Filters zu untersuchen. Bestimmen Sie hierzu für die Planungsprobleme aus Aufgabe 1 die Planungszeit, die mit den folgenden Konfigurationen benötigt wird:

- FFHeuristic und HelpfulFilter (hier können Sie die Ergebnisse aus Aufgabe 1 übernehmen)
- FFHeuristic und NullFilter
- GoalCountHeuristic und NullFilter

(c) Werten Sie Ihre Ergebnisse aus. Wie beurteilen Sie den Einfluss des Filters auf die Suche? Wie gut schneidet die Goal-Count-Heuristik im Vergleich zur FFHeuristik ab? Begründen Sie Ihre Antwort differenziert.

Bitte geben Sie die folgenden Ergebnisse (am besten per Mail an einen der Vorlesungsassistenten) als Lösung ab:

- Quellcode der Klasse `GoalCountHeuristic`
- Quellcode der Methode `performSearch` für die Konfiguration FFHeuristik/NullFilter und die Konfiguration GoalCountHeuristik/NullFilter
- gesammelte Laufzeiten in Tabellenform
- Interpretation der Ergebnisse

4 Nachbereitung

Jetzt sollten Sie eine Idee von der groben Struktur von JavaFF und von der Rolle der Heuristik und der Filter haben. Sie sollten auch in der Lage sein, Daten über die Performanz eines Planers zu sammeln und erste Erfahrungen darin haben, die gesammelten Daten zu interpretieren.

Literatur

[1] COLES, ANDREW I., MARIA FOX, DEREK LONG und AMANDA J. SMITH: *Teaching Forward-Chaining Planning with JavaFF*. In: *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence*, Juli 2008.

[2] FIKES, RICHARD E. und NILS J. NILSSON: *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artificial Intelligence, 2:189–208, 1971.

[3] HOFFMANN, JÖRG und BERNHARD NEBEL: *The FF Planning System: Fast Plan Generation Through Heuristic Search*. Journal of Artificial Intelligence Research, 14:253–302, 2001.