Handlungsplanung

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

Universität Freiburg
Institut für Informatik

Projekt 2: Optimales und suboptimales Planen

Abgabe: 9. Juli 2010

Nachdem Sie sich in Projekt 1 bereits ein wenig mit JavaFF [1] vertraut gemacht haben, können Sie nun mit den ersten größeren Änderungen beginnen.

In der originalen Version findet JavaFF nicht notwendigerweise optimale Pläne, da es sich um ein sogenanntes *satisficing* Planungssystem handelt. Ziel dieses Projektes ist es zunächst, aufbauend auf dem bestehenden Code, ein *optimales* Planungssystem zu entwickeln. Eine Möglichkeit für optimales Planen mit heuristischer Suche ist es, den A*-Algorithmus mit einer zulässigen Heuristik zu verwenden. Sie werden daher im Rahmen dieses Projekts die $h_{\max}$-Heuristik als zulässige Heuristik und A*als zusätzliches Suchverfahren implementieren. Zuletzt werden Sie mit Hilfe des Weighted A*-Algorithmus noch einen Zwischenweg zwischen satisficing und optimalen Planen beschreiten.

1 $h_{\max}$-Heuristik

Da die FF-Heuristik nicht zulässig ist, wissen Sie nicht, wie gut die in Projekt 1 gefundenen Pläne sind. Um zum Vergleich optimale Pläne zu erhalten, benötigen Sie eine zulässige Heuristik. Eine Möglichkeit stellt die $h_{\max}$-Heuristik dar, die Sie bereits in der Vorlesung kennengelernt haben. Bei der Implementierung können Sie davon profitieren, dass die FF-Heuristik auch mit Hilfe des relaxierten Planungsgraphen berechnet wird und Sie daher schon einige nützliche Elemente im Code vorfinden.

Zur Repräsentation des relaxierten Planungsgraphen dient die Klasse `RelaxedPlanningGraph`, die von der Klasse `PlanningGraph` erbt, wobei die Klasse `RelaxedPlanningGraph` eine Vereinfachung der Klasse `PlanningGraph` darstellt.

In der Methode `getPlan` der Klasse `PlanningGraph` wird zunächst der Planungsgraph aufgebaut und dann ein Plan extrahiert. Für die Berechnung der $h_{\max}$-Heuristik ist es jedoch ausreichend, nur den Planungsgraphen aufzubauen. Ergänzen Sie die Klasse daher um eine Methode

```
public void buildGraphStructureToFirstGoalLayer(STRIPSState state),
```

die nur den Planungsgraphen aufbaut, bis das Ziel erfüllt ist.

Um daraufhin den Heuristikwert zu erhalten, müssen Sie bestimmen, in welcher Schicht alle Zielpropositionen erfüllt sind. Werfen Sie einen Blick auf die Methode `goalMet()` und finden Sie heraus, wie Sie durch die Zielpropositionen iterieren können und für jede Propsition die erste Schicht erhalten, in der sie wahr ist. Nutzen Sie die gewonnene Information, um eine Methode

```
public BigDecimal getHMaxValue()
```

zu schreiben, die den Heuristikwert der $h_{\max}$-Heuristik bestimmt.

Da Sie im letzten Projekt bereits die `GoalCountHeuristic` implementiert haben, wissen Sie bereits, wie Sie dem Planungssystem eine neue Heuristik hinzufügen können. Schreiben Sie also eine neue Heuristikklasse `MaxHeuristic`, die die $h_{\max}$-Heuristik berechnet. Verwenden Sie hierzu die zuvor implementierten Methoden. Das benötigte `RelaxedPlanningGraph`-Objekt können Sie sich von dem `STRIPSState` geben lassen.

Hinweis: Für die Laufzeit ist es sinnvoll, den Heuristikwert für jeden Zustand höchstens einmal zu berechnen.

Aufgabe 1: Implementierung der $h_{\max}$-Heuristik

2 Punkte

(a) Implementieren Sie die $h_{\max}$-Heuristik wie oben beschrieben.

(b) Verwenden Sie in dieser Aufgabe zunächst die Bestensuche, um Ihre neue Heuristik zu testen. Bestimmen Sie hierzu für alle Blocksworldinstanzen (neu auf der Homepage) sowie für die jeweils ersten zehn Instanzen von rovers und driverlog die Planungszeit und die Länge der generierten Pläne. Wenn das Verfahren zwei aufeinanderfolgende Instanzen einer Domäne nicht gelöst hat, dürfen Sie die übrigen Instanzen dieser Domäne weglassen.

2 A*-Algorithmus

Leider gibt die Bestensuche im Allgemeinen keine Optimalitätsgarantie. Zum Glück haben Sie in der Vorlesung den A^*-Algorithmus als eine Möglichkeit für optimales Planen mit heuristischer Suche kennengelernt. Der zweite Schritt bei der Entwicklung des optimalen Planungssystems wird es daher sein, diesen Algorithmus zu implementieren.

Da $h_{\max}$ konsistent ist, kann man die Betrachtung der *distance* eines Knotens weglassen. Man kann den Algorithmus aus der Vorlesung also folgendermaßen vereinfachen:

A* (with duplicate detection and reopening)

```
open := new min-heap ordered by (σ ↦ g(σ) + h(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ = open.pop-min()
    if state(σ) ∉ closed:
        closed := closed ∪ {state(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ' := make-node(σ, o, s)
            if h(σ') < ∞:
                open.insert(σ')
return unsolvable
```

Die sauberste Möglichkeit, das JavaFF-System um einen neuen Suchalgorithmus zu erweitern, ist es, eine neue Klasse zu schreiben, die von der bestehenden, abstrakten Klasse `Search` erbt. Am einfachsten ist es wahrscheinlich, sich zum Beispiel die Implementierung der `BestFirstSearch` anzusehen und dann analog vorzugehen. Da bei A* die f-Werte der einzelnen Zustände verglichen werden, werden Sie analog zum `HValueComparator` einen `FValueComparator` benötigen.

Zählen Sie während der Suche die Anzahl der expandierten Zustände für jeden f-Wert und geben Sie sie am Ende als Zusammenfassung aus. Interessant ist vor allem die Anzahl der Knoten, die mit einer gegebenen Heuristik in jedem Fall expandiert werden müssen. Da A* auf der letzten Schicht Glück haben kann (oder auch nicht), variiert dort die Anzahl der expandierten Knoten. In jedem Fall muss A* aber $N = 1 + \sum_{f=0}^{n-1} \mathit{expanded}(f)$ Knoten expandieren, wobei $\mathit{expanded}(f)$ die Anzahl der expandierten Knoten für jeden f-Wert und n die Länge des gefundenen Plans bezeichnet.

Hinweis: Achten Sie bei Ihrer Implementierung darauf, geeignete Datenstrukturen zu verwenden.

Aufgabe 2: Implementierung des A*-Algorithmus

2 Punkte

(a) Implementieren Sie den A*-Algorithmus. Die A*-Suche soll analog zu den anderen Suchverfahren folgendermaßen aufgerufen werden können:

```
Heuristic heuristic = new MaxHeuristic();

AStarSearch as =
    new AStarSearch(initialState, heuristic);

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

Da es sich bei A* um ein systematisches, vollständiges Suchverfahren handelt, sollten Sie jeweils alle anwendbaren Aktionen in Betracht ziehen. Sie können daher die Verwendung des NullFilters direkt in die Implementierung des Algorithmus aufnehmen.

(b) Testen Sie Ihre Implementierung mit der $h_{\max}$-Heuristik auf allen Blocksworldinstanzen sowie auf den jeweils ersten zehn Instanzen von rovers und driverlog. Bestimmen Sie jeweils die Planlänge, die Planzeit, die Gesamtzahl der expandierten Knoten und die Zahl N der in jedem Fall expandierten Knoten. Wenn das Verfahren zwei aufeinanderfolgende Instanzen einer Domäne nicht gelöst hat, dürfen Sie wieder die übrigen Instanzen dieser Domäne weglassen. Vergleichen Sie Ihre Ergebnisse mit denen aus Aufgabe 1.

Zur Selbstkontrolle: Falls Sie alles korrekt implementiert haben, sollten Sie folgende N-Werte für die jeweils erste Instanz jeder Domäne erhalten: für Rovers $N = 693$, für Driverlog $N = 10$ und für Blocksworld $N = 18$.

3 Ein kurzes Zwischenfazit

Wenn Sie Ihre Ergebnisse aus Aufgabe 2 mit denen aus Aufgabe 1 des vorherigen Projekts vergleichen, sehen Sie, dass Sie deutlich weniger Planungsaufgaben lösen konnten als mit dem originalen JavaFF-System. Der Hauptgrund hierfür liegt darin, dass FF ein satisficing Planungssystem ist, das keine Garantien für die Länge der Pläne gibt, und optimales Planen sehr viel schwieriger ist als Planen ohne Optimalitätsgarantie.

Man befindet sich also in einem Konflikt: Entweder man kann viele Pläne in kurzer Zeit finden, die dafür sehr schlecht sein können, oder man beschränkt sich auf optimales Planen und kann dafür nur relativ einfache Probleme lösen. Hier wäre es natürlich schön, wenn man auch einen Zwischenweg wählen könnte, der sich nicht gleich auf optimale Pläne beschränkt, aber dennoch eine bestimmte Planqualität garantiert. Daher werden Sie sich im nächsten Kapitel mit diesem Thema beschäftigen.

4 Suboptimales Planen mit Qualitätsgarantie

In der Vorlesung haben Sie neben dem A*-Algorithmus auch noch eine Variante namens *weighted*-A*-Algorithmus (WA*) kennengelernt.

Der WA*-Algorithmus unterscheidet sich von dem „normalen“ A*-Algorithmus nur darin, dass er von einem zusätzlichen Parameter W abhängt, der die Gewichtung der Heuristik bestimmt. Genaugenommen berechnet sich der f-Wert eines Zustands s bei WA* als $f(s) = g(s) + Wh(s)$.

Eine sehr schöne Eigenschaft dieses Algorithmus ist es, dass er mit einer zulässigen Heuristik garantiert, dass die gefundenen Pläne höchstens Länge Wh^* haben, wobei h^* die Länge eines optimalen Plans ist.

Das Ziel der nächsten Aufgabe ist es nun, WA* zu implementieren und den Einfluss des Gewichtungsfaktors auf die Planlänge und die Suchzeit zu untersuchen.

WA* (with duplicate detection and reopening)

```
open := new min-heap ordered by (σ ↦ g(σ) + W h(σ))
open.insert(make-root-node(init()))
closed := ∅
distance := ∅
while not open.empty():
    σ = open.pop-min()
    if state(σ) ∉ closed or g(σ) < distance(state(σ)):
        closed := closed ∪ {state(σ)}
        distance(state(σ)) := g(σ)
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ' := make-node(σ, o, s)
            if h(σ') < ∞:
                open.insert(σ')
return unsolvable
```

Die einfachste Möglichkeit, ihre A*-Implementierung zu einer WA*-Implementierung zu erweitern, ist es, den Comparator so zu verändern, dass er bei der Erstellung das Gewicht W übergeben bekommt, das er dann bei den Vergleichen verwendet. Zusätzlich müssen Sie ihre A*-Implementierung um die *distance*-Behandlung erweitern, die wir zuvor vereinfachend weggelassen haben.

Aufgabe 3: Implementierung und Untersuchung des WA*-Algorithmus

1 Punkt

(a) Implementieren Sie den WA*-Algorithmus.

(b) Testen Sie Ihre Implementierung mit der $h_{\max}$-Heuristik für die Gewichte $W = 10$, $W = 5$ und $W = 2$.

Bestimmen Sie für alle Blocksworldinstanzen sowie die jeweils ersten zehn Instanzen von rovers und driverlog die Planlänge, die Planzeit und die Gesamtzahl der expandierten Knoten.

Wenn das Verfahren für ein Gewicht bei zwei aufeinanderfolgende Instanzen einer Domäne keine Lösung gefunden hat, dürfen Sie für dieses Gewicht die übrigen Instanzen dieser Domäne weglassen.

Interpretieren Sie Ihre Ergebnisse.

5 Nachbereitung

Im Rahmen dieses Projektes sollten Sie zum einen gelernt haben, den A*-Algorithmus selbst zu implementieren. Zum anderen sollten Sie nun ein Bewusstsein für die Unterschiede zwischen optimalen Planen, satisficing Planen und suboptimalen Planen mit Qualitätsgarantie entwickelt haben.

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.