KI-Praktikum

M. Helmert, G. Röger, P. Eyerich, R. Mattmüller
Sommersemester 2009

Universität Freiburg
Institut für Informatik

Projekt 4: Suche und Planen

Abgabe: 19. Juli 2009

Präsenzaufgaben

Aufgabe 1

In dieser Aufgabe sollen Sie verschiedene Such-Verfahren und Heuristiken miteinander vergleichen. Betrachten Sie dazu zunächst das Modul `transport.py`, das eine A*-Implementierung zur Lösung des *Transportproblems* implementiert. Dabei ist das Transportproblem folgendermaßen definiert:

Es sollen mithilfe eines Fahrzeugs Pakete zwischen verschiedenen Orten transportiert werden. Jedes Paket hat einen Start- und einen Zielort. Das Fahrzeug kann jeweils mit Kosten 1 an einen beliebigen anderen Ort fahren, ein Paket an dessen Startort einladen oder ein geladenes Paket am Zielort ausladen. Das Fahrzeug hat unendliche Kapazität, kann also beliebig viele Pakete gleichzeitig transportieren. Das Ziel ist es, alle Pakete an ihren Zielorten abzuliefern.

Anmerkung: Das optimale Transportproblem ist NP-schwierig, d. h. es gibt vermutlich keine polynomiellen Lösungsverfahren. Die Schwierigkeit liegt in der Bestimmung der Orte, die mehrfach angefahren werden müssen.

In `transport.py` sind die domänenspezifischen Teile eines Planungssystems für Transportprobleme implementiert. Mit `astar_experiments.py` können Sie dieses Planungssystem für die Beispieleingaben `transport01` bis `transport06` (im Unterverzeichnis `transport-inputs`) testen. Die Beispieleingaben verwenden ein sehr einfaches zahlenbasiertes Format: Die erste Zeile gibt die Zahl der Orte an, die von 0 an durchnumeriert sind (bei Eingabe K gibt es also die Orte $0, 1, 2, \ldots, K-1$), die weiteren Zeilen bis zum Ende der Datei geben, durch Leerzeichen getrennt, jeweils die Start- und Endorte für jedes Paket an. Ein Paket hat niemals denselben Start- und Endort. Das Fahrzeug beginnt immer am Ort 0.
Die Ausgabe des Programms enthält für alle bearbeiteten Probleme die gefundene (optimale) Lösung sowie als weitere Informationen den h-Wert des Startknotens, die Zahl der expandierten Knoten, sowie die optimale Lösungslänge. Die Numerierung der Pakete (0, 1, 2, 3) entspricht dabei deren Reihenfolge in der Eingabedatei.
Achtung: Da das momentane System noch nicht sehr effizient arbeitet, sollten Sie, bevor Sie `astar_experiments.py` aufrufen, ein Speicherlimit setzen, z.B. mittels `ulimit -m 1000000 -v 1000000` für ein Limit von 1 GB.

(a) Lösen Sie die gegebenen Transportprobleme, indem Sie wie oben beschrieben ein sinnvolles Speicherlimit und eventuell ein Zeitlimit setzen und `astar_experiments.py` aufrufen. Tragen Sie für alle Probleme jeweils die Laufzeit und die Anzahl der Knotenexpansionen in die erste Spalte von Tabelle 1 ein. Ist ein Problem nicht innerhalb der gegeben Limits lösbar, so tragen Sie an der entsprechenden Stelle einen Strich ein.

(b) Wandeln Sie den A*-Algorithmus in `search.py` so ab, dass er einen Knoten nicht expandiert, wenn ein Knoten mit demselben Zustand bereits expandiert wurde. (Dies ist eine Optimierung, die nur bei konsistenten Heuristiken verwendet werden kann.) Evaluieren Sie Ihren Algorithmus anhand der Beispiel-Probleme und tragen Sie Ihre Ergebnisse in die zweite Spalte von Tabelle 1 ein.

(c) Die in `transport.py` initial implementierte Heuristik h_1 zählt für einen Zustand, wieviele Ein- und Ausladeaktionen noch benötigt werden, um das Ziel zu erreichen. Diese Heuristik ist zulässig (d. h., dass sie für keinen Zustand die Kosten zum Ziel überschätzt) und konsistent (d. h., dass für alle Knoten n und deren Nachfolger $succ(n)$ gilt, dass die geschätzen Kosten von n kleiner gleich den geschätzten Kosten von $succ(n)$ plus den tatsächlichen Kosten für den Übergang von n nach $succ(n)$ sind).

Verbessern Sie h_1 zu h_2, indem sie h_1 so erweitern, dass zusätzlich gezählt wird, wieviele Orte mindestens noch angefahren werden müssen, ohne dabei mehrfach anzufahrende Orte zu berücksichtigen. h_2 soll ebenfalls zulässig und konsistent sein. Evaluieren Sie ihre Heuristik, indem Sie die Laufzeit und die Anzahl der Knotenexpansionen für die Beispielprobleme in die dritte Spalte von Tabelle 1 eintragen.

(d) Implementieren Sie einen Breitensuch-Algorithmus, indem sie die Methode `search` der Klasse `BreadthFirstSearch` in `search.py` vervollständigen. Vermeiden Sie auch hier die mehrfache Expansion von Knoten mit gleichem Zustand. Verfahren Sie dazu ähnlich wie in Teilaufgabe (c), beachten Sie jedoch, dass bei Breitensuche ein Wiederöffnen eines Knotens nie notwendig ist. Evaluieren Sie ihren Algorithmus, indem Sie die Ergebnisse auf den Beispielproblemen in Tabelle 1 eintragen und mit den Ergebnissen aus den ersten drei Teilaufgaben vergleichen.

Aufgabe 2

A*-Suche mit einer zulässigen Heuristik liefert immer optimale Lösungen. Häufig ist jedoch die Optimalität der A*-Suche eher schädlich, da unter Umständen während der Suche sehr viel Zeit damit verbracht wird, zwischen Pfaden zu unterscheiden, deren Kosten nur geringfügig variieren. In dieser Aufgabe wollen wir der Frage nachgehen, ob es eine angemessenere Definition von $f(n)$ als $f(n) = g(n)+h(n)$ gibt, wenn man mehr Wert auf die Minimierung des Suchaufwandes als auf die Minimierung der Kosten der gefundenen Lösung legt (ohne jedoch die Güte der Lösungen ganz außer Acht lassen zu wollen).

Der *Weighted-A*-Algorithmus* (WA*) ist identisch mit dem A*-Algorithmus, mit dem Unterschied, dass $f(n) = g(n) + wh(n)$ für ein Gewicht $w \geq 1$. Er besitzt die Eigenschaft, immer eine Lösung s zu finden, die höchstens um den Faktor w teurer ist als eine optimale Lösung s^*.

(a) Erweitern Sie die A*-Implementierung in `search.py` zu WA*, indem Sie die Berechnung des f-Wertes eines Zustands in der Vorrangwarteschlange `PriorityQueue` mit einem Gewicht w parametrisieren.

(b) In `sliding_tiles.py` sind die Regeln des Schiebepuzzles[1] kodiert. Zusätzlich enthält die Klasse `SlidingTilesState` eine Heuristik, die jeden Zustand mit der Summe der Manhattan-Distanzen aller Steine zu ihrem jeweiligen Ziel bewertet. In `weighted_astar_experiments.py` befinden sich einige ausgewählte Instanzen (11 Steine auf einem 3×4-Feld). Messen Sie für die Gewichte 2.0, 1.5, 1.1 und 1.0 (A*) jeweils Laufzeiten, Knotenexpansionen und die Güte der gefundenen Lösungen im Verhältnis zu den optimalen Kosten einer Lösung der gegebenen Instanzen und protokollieren Sie diese in Tabelle 2. Überprüfen Sie anhand der vorliegenden Ergebnisse, ob die gefundenen Lösungen s das Gütekriterium $len(s) \leq w \cdot len(s^*)$ erfüllen.

(c) Vergleichen Sie die Ergebnisse in Tabelle 2 mit gieriger Bestensuche, also $f(n) = h(n)$, und ergänzen Sie Tabelle 3 entsprechend. Implementieren Sie dazu in `search.py` gierige Bestensuche, etwa indem Sie von `AStarSearch` ableiten oder in `AStarSearch` und der `PriorityQueue` statt nur eines Gewichts für h noch ein zweites Gewicht für g mitführen und dieses für gierige Bestensuche auf Null setzen.

Aufgabe 3

In dem Verzeichnis `planningsystems` finden Sie das Planungssystem *FF* und im Verzeichnis `pddl` die Beschreibung einer Planungsinstanz, bestehend aus zwei Dateien: Die Domänenbeschreibung `blocks-domain.pddl` spezifiziert, welche Prädikate verwendet werden, und die Aktionen dieser Domäne. Eine konkrete Planungsaufgabe für diese Domäne ist in der Datei `blocks-problem.pddl` angegeben. Dort wird festgelegt, welche Objekte es gibt, wie der Anfangszustand aussieht und welche Zielformel erfüllt werden soll.
Bei dieser Planungsdomäne handelt es sich um *Blocksworld*. Bei Blocksworld liegen unterscheidbare Blöcke zu mehreren Stapeln auf einem Tisch. Man kann nun mit einer Hand Blöcke, die nicht unter einem anderen Block liegen, aufnehmen und entweder auf den Tisch oder ganz oben auf einen bereits bestehenden Stapel legen. Ziel ist es, die Blöcke so umzusortieren, dass eine bestimmte Zielkonfiguration erfüllt ist, wobei man stets höchstens einen Block in der Hand halten kann.

[1] `http://de.wikipedia.org/wiki/15-Puzzle`

Problem	A*, d^-, h_1		A*, d^+, h_1		A*, d^+, h_2		BFS	
	n	δ	n	δ	n	δ	n	δ
`transport_1`								
`transport_2`								
`transport_3`								
`transport_4`								
`transport_5`								
`transport_6`								

Tabelle 1: Expandierte Knoten n und Laufzeiten δ von A* und BFS auf sechs ausgewählten Transport-Problemen. Dabei bedeutet d^- ohne und d^+ mit Duplikatelimination; h_1 und h_2 sind unterschiedliche Heuristiken.

Problem	$w = 2.0$			$w = 1.5$			$w = 1.1$			$w = 1.0$	
	n	c	r	n	c	r	n	c	r	n	c
`puzzle_1`											
`puzzle_2`											
`puzzle_3`											
`puzzle_4`											
`puzzle_5`											

Tabelle 2: Expandierte Knoten n, Lösungskosten c und Lösungsgüte r von WA* für unterschiedliche Gewichte w auf fünf ausgewählten Schiebepuzzle-Problemen.

Problem	Gierige Bestensuche		
	n	c	r
`puzzle_1`			
`puzzle_2`			
`puzzle_3`			
`puzzle_4`			
`puzzle_5`			

Tabelle 3: Expandierte Knoten n, Lösungskosten c und Lösungsgüte r von gieriger Bestensuche auf fünf ausgewählten Schiebepuzzle-Problemen.

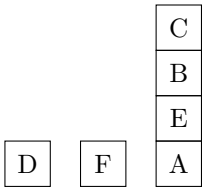
(a) Sehen Sie sich die Problembeschreibung der Planungsaufgabe genauer an. Was ist der Unterschied zwischen den Aktionen `unstack` und `pick-up`? Wie liegen die Blöcke im Anfangszustand?

(b) Lösen Sie mit FF die gegebene Planungsaufgabe, indem Sie das Planungssystem (vom `pddl`-Verzeichnis aus) folgendermaßen aufrufen:

```
../planningsystems/ff -o blocks-domain.pddl -f blocks-problem.pddl
```

Sehen Sie sich den generierten Plan an und beschreiben Sie in einem Satz, was dort gemacht wird.

(c) Erstellen Sie eine zusätzliche Instanz `blocks-problem-new.pddl` für die Blocksworld-Domäne. Der Anfangszustand sollte dabei folgendermaßen aussehen:



Im Ziel soll B auf D, D auf C und E auf F liegen. Findet FF für diese Instanz einen optimalen (d. h. einen kürzesten möglichen) Plan?

(d) Kopieren Sie die Domänenbeschreibung `blocks-domain.pddl` mit `svn cp` nach `blocks-domain-multihands.pddl` und die ursprüngliche Problemdatei nach `blocks-problem-3hands.pddl`. Modifizieren Sie beide Dateien so, dass der Planer statt der einen Hand drei Hände verwenden kann. Bedenken Sie dabei, dass es leicht sein könnte, dass Sie für eine andere Instanz dieser Domäne eventuell auch vier oder fünf Hände verwenden wollen. Die Änderungen in der Domänenbeschreibung sollten daher so sein, dass die genaue Anzahl der Hände erst in der Instanzdatei festgelegt wird.

Lösen Sie die neue Planungsaufgabe mit FF. Werden die zusätzlichen Hände im Plan verwendet?

Hausaufgaben

Aufgabe 4

In dieser Aufgabe geht es darum, eine möglichst gute Heuristik für Blocksworld-Probleme zu finden. Als Grundlage dient hierfür der in `blocks.py` implementierte domänenspezifische Blocksworld-Planer.
Ihre Aufgabe ist es, die Methode `heuristic` der Klasse `BlocksWorldState` so zu erweitern, dass für einen gegebenen Zustand eine möglichst gute Abschätzung der Kosten bis zum nächsten Ziel (also der Anzahl der noch benötigten Aktionen) gegeben wird. Dabei sollte ihre Heuristik zulässig und konsistent (siehe Aufgabe 1) sein. Andere Teile des Systems sollten Sie nicht verändern.
Um ihre Heuristik zu testen, können Sie die Beispieleingaben `blocks02` bis `blocks10` aus dem Unterverzeichnis `blocks-inputs` verwenden oder mit Hilfe des Programmes `bwstates` neue Probleme erstellen. Rufen Sie dazu das Programm folgendermaßen auf:
`./bwstates -n X -s 2 -r Y > Z`,
wobei Sie `X` durch die gewünschte Anzahl an Blöcken, `Y` durch einen zufälligen Wert (der die Positionen der Blöcke eindeutig bestimmt) und `Z` durch den gewünschten Dateinamen ersetzen.
Einen Bonuspunkt erhält die Gruppe, deren Heuristik für eine noch zu bestimmende Menge von Eingabeproblemen insgesamt am wenigsten Knotenexpansionen verursacht (da die Anzahl der von A* auf der letzten Schicht expandierten Knoten willkürlich ist, 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 Kosten des gefundenen Plans bezeichnen).

Aufgabe 5

Rush Hour ist ein Schiebepuzzle, bei dem es darum geht, ein bestimmtes Auto aus einem Verkehrschaos mit anderen Autos und Bussen zu befreien, wobei sich alle Fahrzeuge jeweils nur horizontal oder vertikal auf einem 6×6-Grid verschieben lassen.
Sie können das Spiel zum Beispiel unter

`http://home.hetnet.nl/~fredvonk/rushhour.htm`

online spielen.
In dieser Aufgabe sollen Sie das Spiel in PDDL formalisieren und mit verschiedenen Planungssystemen lösen. Sie haben in den Anwesenheitsaufgaben bereits mit PDDL gearbeitet. Die Planungsinstanzen dort sind im sogenannten STRIPS-Fragment beschrieben, das nur einfache Konstrukte enthält. Sie benötigen für die erste Teilaufgabe nur Konstrukte, die auch in diesem Beispiel enthalten sind, es ist also zunächst nicht unbedingt notwendig, sich darüber hinausgehend mit PDDL zu beschäftigen. Falls Sie dennoch eine genaue Definition der Sprache benötigen, finden Sie im Repository die BNF-Beschreibung `bnf.pdf`

(Sie dürfen für das STRIPS-Fragment alles verwenden, das keine zusätzlichen Requirements benötigt.).

(a) Formalisieren Sie das Spiel im STRIPS-Fragment von PDDL. Mit diesem Fragment ist es nicht einfach zu beschreiben, dass das Bewegen eines Fahrzeuges über mehrere Felder nur ein einzelner Zug ist. Sie können daher in dieser Teilaufgabe davon ausgehen, dass die Fahrzeuge pro Zug nur um ein Feld bewegt werden können.

Erstellen Sie neben der Domänenbeschreibung Problembeschreibungen für verschieden schwere Rush-Hour-Spiele, nämlich für die Levels 1, 11, 21 und 31 des ersten Cardsets und Level 32 des zweiten Cardsets auf der oben angegebenen Webseite.

Hinweis: Sie können davon ausgehen, dass Autos stets 2 Felder groß sind und Busse stets 3 Felder groß sind.

Testen Sie ihre Formalisierung, indem Sie die Planungsaufgaben mit dem FF-System lösen.

(b) Dauerhaft ist es natürlich nicht schön, wenn man seine Ergebnisse nicht mit der optimalen Anzahl von Zügen (die auf der Webseite jeweils mit angegeben ist) vergleichen kann.

Verwenden Sie daher die PDDL-Erweiterung um Aktionskosten, um alle aufeinanderfolgenden Züge mit einem Fahrzeug in eine Richtung nur einmal zu zählen und diesen neuen Wert zu minimieren.

Sie können die Aktionskosten folgendermaßen verwenden:

- Erweitern Sie in der Domänenbeschreibung die Requirements um `:action-costs`.
- Fügen Sie nach dem `:predicates`-Abschnitt einen Abschnitt
  `(:functions (total-cost))`
  hinzu.
- Fügen Sie in der Problembeschreibung nach dem `:init`- und dem `:goal`-Abschnitt einen Abschnitt
  `(:metric minimize (total-cost))`
  hinzu.
- Sie können nun in den Aktionen einen Effekt
  `(increase (total-cost) <zahl>)`
  verwenden, wobei `<zahl>` eine beliebige positive Zahl sein kann.

Ändern Sie also ihre Domainbeschreibung und Ihre Instanzbeschreibungen so, dass `total-cost` am Ende der Anzahl Züge entspricht, die auch im richtigen Spiel gezählt werden.

Tipp: Sie können zum Beispiel gesonderte Startaktionen verwenden, die die Kosten geeignet erhöhen und dafür eine bestimmte Aktion "freischalten". Vergessen Sie nicht, auch geeignete Stopaktionen einzuführen.

Da das FF-Planungssystem älter ist als die Erweiterung um Aktionskosten, können Sie es leider nicht verwenden, um diese neuen Instanzen zu lösen. Wir haben Ihnen daher drei weitere, aktuellere Planungssysteme von der letzten International Planning Competition (IPC) ins Repository gestellt (zu finden in den Verzeichnissen `seq-opt-base` (optimaler Baseline-Planner, im Prinzip Breitensuche), `seq-sat-base` (Baseline-Planner des suboptimalen Tracks, ignoriert Aktionskosten und verwendet dann FF) und `seq-sat-lama` (Gewinner des suboptimalen Tracks, LAMA steht für LandMarks)). Sie können die Planungssysteme kompilieren, indem Sie in das jeweilige Verzeichnis wechseln und dort

```
./build
```

ausführen. LAMA und der optimale Baselineplanner benötigen python2.5. Falls das bei Ihnen nicht installiert ist, müssen Sie in der Datei `build` die entsprechenden zwei Zeilen einkommentieren, um Python mitzukompilieren.

Die Planer können dann mit

```
./plan <domain> <task> <outputfile>
```

gestartet werden. `<outputfile>` gibt an, in welche Datei der Planer die gefundene Lösung schreiben soll (Achtung, diese Datei wird überschrieben!). Beachten Sie, dass es sich bei LAMA um einen sogenannten *Anytime*-Planer handelt. Das heißt, sobald LAMA einen Plan gefunden hat, schreibt er ihn zwar in eine Datei, sucht aber trotzdem weiter nach besseren Lösungen bis er abgebrochen wird. Die Dateien mit den gefundenen Lösungen beginnen alle mit dem in `outputfile` angegebenen Dateinamen und werden aufsteigend nummeriert (der beste Plan befindet sich dann also in der Datei mit der höchsten Zahl).

Vergleichen Sie die drei Planer hinsichtlich der Zahl der gelösten Instanzen und der Planqualität (total-cost)[2]. Verwenden Sie für die Planer eine Zeitbeschränkung von 10 Minuten. Sie können die Plankosten entweder selbst aus dem Plan extrahieren oder den Validator aus dem Repository verwenden (siehe ANLEITUNG-Datei dort).

Aufgabe 6

Das metrische Problem des Handlungsreisenden (metric traveling salesman problem, metric TSP) ist wie folgt definiert: Gegeben sind n Knoten mit paarweisen Distanzen $c(i, j) \in \mathbb{R}$, so dass $c(i, i) = 0$ für alle $i = 1, \ldots, n$, dass $c(i, j) = c(j, i)$ für alle $i, j = 1, \ldots, n$ (Symmetrie) und dass die Dreiecksungleichung $c(i, j) + c(j, k) \geq c(i, k)$ für alle $i, j, k = 1, \ldots, n$ erfüllt ist. Gesucht ist eine möglichst kurze Tour, also eine Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$,

[2]Bei Problemen bei der Ausführung der Planer: Frag Gabi (roeger@informatik.uni-freiburg.de)

so dass die Länge

$$len(\pi) = c(\pi(n), \pi(1)) + \sum_{i=1}^{n-1} c(\pi(i), \pi(i+1))$$

von π minimal wird.
In dieser Aufgabe soll es nicht darum gehen, optimale Lösungen zu finden (das Optimierungsproblem ist NP-äquivalent), sondern lediglich darum, in einer gegebenen Zeit eine möglichst gute Lösung zu finden.
Implementieren Sie mit einem Ansatz Ihrer Wahl einen Anytime-Algorithmus für das metrische TSP, der schon früh eine erste, wenn auch evtl. schlechte, Lösung und dann kontinuierlich bessere Lösungen liefert. Bei der Auswertung werden wir für jede Instanz die beste nach 10 Minuten gefundene Tour berücksichtigen. Um Punkte für diese Aufgabe zu bekommen, sollte Ihr Algorithmus die beiden folgenden Bedingungen erfüllen: Er sollte innerhalb der ersten 10 Minuten zuverlässig Lösungen finden, die höchstens 50% länger sind als die jeweiligen optimalen Lösungen, und er sollte nach endlicher Zeit die optimale Lösung ausgeben.
Einen Bonuspunkt erhält die Gruppe mit dem Algorithmus, der im folgenden Sinne am besten ist: Für jede innerhalb von 10 Minuten gelöste Instanz erhält jede Gruppe $len(\pi^*)/len(\pi)$ Punkte, wenn π die kürzeste von ihr gefundene Tour und π^* die kürzeste Tour überhaupt für das Problem ist. Die Gruppe mit den meisten Punkten gewinnt.
Ihr Algorithmus sollte Distanzmatrizen der folgenden Form (in Python) als Eingabe akzeptieren:

```
cost = [
    [c11, c12, c13, ..., c1n],
    [c21, c22, c23, ..., c2n],
    ...,
    [cn1, cn2, cn3, ..., cnn]
    ]
```

Ein konkretes Beispiel finden Sie im Repository in der Datei `sample_tsp.py`.
Als Ausgabe sollte er in der ersten Zeile die gefundene Tour, beginnend bei Knoten 1, enthalten, etwa `1 5 7 2 8 4 3 6`, und in der zweiten Zeile die Kosten dieser Tour.