Модели планирования действий в системах искусственного интеллекта

Планированием называется процесс выработки последовательности действий, позволяющих достичь цели.

Будем рассматривать среду классического планирования, которая является полностью наблюдаемой, детерминированной, конечной, статической (изменения происходят только в результате действий агента) и дискретной (по времени, действиям, объектам и результатам).

Задача планирования

Рассмотрим использование обычного агента, решающего задачи с помощью стандартных алгоритмов поиска. Такой агент при решении задачи планирования сталкивается со следующими проблемами.

1. Наличие огромного количества действий, не относящихся к делу. Например, при решении задачи покупки одного экземпляра книги с 10-цифровым номером ISBN, агент-покупатель, осуществляющий одно действие на каждый возможной номер ISBN, должен исследовать результаты 10 миллиардов действий. С другой стороны, если агент имеет информацию о том, что его цель `Have(1234567890)` и о том, что действие `Buy(x)` приводит к результату `Have(x)`, то агенту-планировщику достаточно единственного шага унификации, чтобы определить, что правильным действием является `Buy(1234567890)`.

2. Определение хорошей эвристической функции.

Рассмотрим задачу планирования покупки 4-х разных книг. Общее количество планов будет составлять величину 10^{40}. Цель для данной задачи можно представить как конъюнкцию `Have(A)&Have(B)&Have(C)&Have(D)`. Здесь имеет смысл использовать в качестве эвристической функции количество невыполненных коньюнктов.

Тогда состояние, содержащее выражение `Have(A)&Have(C)` будет иметь стоимость 2.

3. Возможность декомпозиции задачи.

Например, в наихудшем случае время, необходимое для поиска наилучшего плана доставки n пакетов экспресс-почты адресатам, разбросанным по всей России составит величину O(n!). Если же использовать декомпозицию задачи на независимые подзадачи в соответствии с ближайшим к каждому адресату аэропортом, то время уже составит величину O((n/k)!*k).

Язык описания состояний и действий

Способ представления задач планирования (состояний, действий и целей) должен обеспечивать возможность для алгоритмов планирования воспользоваться логической структурой задачи. Возможности языка среды CLIPS достаточны для этих целей.

Рассмотрим описание задачи планирования на примере одной из наиболее широко известных проблемных областей планирования, известной под названием **мира блоков**.

Эта проблемная область состоит из множества блоков кубической формы, находящихся на столе. Блоки можно укладывать в столбик, но на верхней поверхности одного блока может быть непосредственно размещен только один блок. Робот может брать манипулятором только один блок и перемещать его либо на стол, либо на верхнюю поверхность другого блока. Цель заключается в построении одного или нескольких столбиков из блоков. Например, задача может состоять в том, чтобы поставить блок `B` на блок `C` и блок `A` на блок `B`.

Представление **состояний**. В планировщиках применяется декомпозиция мира на логические условия, а состояние представляется в виде конъюнкции положительных литералов или, на языке CLIPS, в виде

совокупности выражений с синтаксисом представления фактов. Например, для указания на то, что блок B находится на блоке x, где x – либо другой блок, либо стол, используется конструкция on (up B)(down ?x), построенная на основе и шаблона on для двух неупорядоченных фактов (up B) и (down ?x), где переменная ?x может принимать значение либо Table, либо имя другого блока.

Представление **действий**. Любая схема совокупности действий задается на CLIPS в терминах правила, состоящего из антецедента - совокупности предусловий - условных элементов (УЭ), которые совместно должны соблюдаться до того, как совокупность действий может быть выполнена, и консеквента – собственно совокупности действий, переводящих задачу в новое состояние. Например, схему совокупности действий, состоящих в переносе блока B с блока A на блок C можно записать следующим образом:

```
(defrule move
    ?onBA <- (on (up B)(down A))
=>
    (assert (on (up B)(down C)))
     (retract ?onBA)
)
```

Представление **целей**. Цель – это частично заданное состояние, представленное в виде конъюнкции положительных базовых литералов. Состояние s удовлетворяет цели g, если s содержит все атомы из g (и, возможно, другие атомы). Например, цель построения столбика из трех блоков, в котором блок A стоит на блоке B, а блок B на блоке C, находящемся на столе, может быть представлена совокупностью двух правил:

```
(defrule init-system
    (initial-fact)
=>
    (assert (goal(current-task find)))
)
```

```
(defrule finish-process
      (on (up A)(down B))
      (on (up B)(down C))
      (on (up C)(down Table))
      ?goal-ptr <- (goal (current-task find))
=>
      (retract ?goal-ptr)
)
```

Под **решением** для задачи планирования будем понимать последовательность действий, которая будучи выполненной в начальном состоянии (`initial-fact`), приводит к состоянию, удовлетворяющему цели.

Схема совокупности действий, позволяющих переместить блок `b` с блока `x` на блок `y`, если свободны верхние поверхности и блока `b`, и блока `y`, представлена правилом `move`. После выполнения этой совокупности верхняя поверхность блока `x` свободна, а блока `y` – нет.

```
(defrule move
      ?onbx <- (on (up ?b)(down ?x))
      (clear (name-clear ?b))
      ?cleary <- (clear (name-clear ?y))
      (goal (current-task find))
=>
      (assert (on (up ?b)(down ?y)))
      (assert (clear (name-clear ?x)))
      (retract ?onbx)
      (retract ?cleary)
)
```

Здесь используется новый шаблон `clear` для представления на его основе состояний свободной верхней поверхности блока `b (clear (name-clear ?b))` и блока `y (clear (name-clear ?y))`, входящих в

состояние-предусловие совокупности, и действия добавления в базу знаний факта свободной верхней поверхности блока x (assert (clear (name-clear ?x))). Если ?x=Table, то результатом этого действия будет факт clear (name-clear Table)), но поверхность стола не должна становиться свободной (поскольку она и так всегда свободна), а если ?y=Table, то совокупность действий имеет УЭ (clear (name-clear Table)), но вся поверхность стола не обязана быть свободной для того, чтобы на нее можно было поместить блок (поскольку на столе всегда и без того достаточно места). Для исправления такого положения необходимо предусмотреть два изменения. Во-первых, введем еще одно правило для перемещения блока b с блока x на стол:

```
(defrule movetotable
    ?onbx <- (on (up ?b)(down ?x))
    (clear (name-clear ?b))
    (goal (current-task find))
=>
    (assert (on (up ?b)(down Table)))
    (assert (clear (name-clear ?x)))
    (retract ?onbx)
)
```

Во-вторых, примем интерпретацию условного элемента (clear (name-clear ?b)) как означающую, что “на b есть достаточно места, чтобы поместился один блок”. При этой интерпретации УЭ (clear (name-clear Table)) всегда будет истинным. Для реализации этой интерпретации включим факт (clear (name-clear Table)) в начальный список фактов.

Единственная проблема состоит в том, что ничто не будет препятствовать планировщику использовать правило move с ?y=Table вместо правила movetotable. Можно смириться с этой проблемой (она приводит к созданию пространства поиска с размерами больше

необходимого, но не становится причиной получения неправильных результатов) или ввести в антецедент `move` условные элементы на основе шаблона `block`:

```
    (block (name-block ?b))
    (block (name-block ?y))
```

Наконец, существует проблема фиктивных действий, таких как правило `move` при `?x=?y`, которые должны представлять собой пустую операцию, но имеют противоречивые результаты. Обычно принято игнорировать подобные проблемы, поскольку они редко вызывают выработку неверных планов. Правильный подход состоит в добавлении УЭ-проверок в антецеденты правил, как показано в окончательном описании задачи планирования в мире из 3-х блоков:

```
(deftemplate on
    (slot up (type SYMBOL))
    (slot down (type SYMBOL))
)
(deftemplate block
    (slot name-block (type SYMBOL))
)
(deftemplate clear
    (slot name-clear (type SYMBOL))
)
(deftemplate goal
    (slot current-task (type SYMBOL))
)
(deffacts init
    (on (up A)(down Table))
    (on (up B)(down Table))
    (on (up C)(down A))
    (block (name-block A))
```

```
    (block (name-block B))
    (block (name-block C))
    (clear (name-clear B))
    (clear (name-clear C))
    (clear (name-clear Table))
)
(defrule init-system
    (initial-fact)
=>
    (assert (goal(current-task find)))
)
(defrule move
    ?onbx <- (on (up ?b)(down ?x))
    (clear (name-clear ?b))
    ?cleary <- (clear (name-clear ?y))
    (block (name-block ?b))
    (block (name-block ?y))
    (test (not (eq ?b ?x)))
    (test (not (eq ?b ?y)))
    (test (not (eq ?x ?y)))
    (goal (current-task find))
=>
    (assert (on (up ?b)(down ?y)))
    (assert (clear (name-clear ?x)))
    (retract ?onbx)
    (retract ?cleary)
)
(defrule movetotable
    ?onbx <- (on (up ?b)(down ?x))
    (clear (name-clear ?b))
```

```
    (block (name-block ?b))
    (block (name-block ?x))
    (test (not (eq ?b ?x)))
    (goal (current-task find))
=>
    (assert (on (up ?b)(down Table)))
    (assert (clear (name-clear ?x)))
    (retract ?onbx)
)
(defrule finish-process
  (declare (salience 1000))
    (clear (name-clear A))
    (on (up A)(down B))
    (on (up B)(down C))
    (on (up C)(down Table))
    ?goal-ptr <- (goal (current-task find))
=>
    (retract ?goal-ptr)
    (printout t "Solution found" crlf)
)
```

При использовании стратегии “вширь” (breadth) в среде CLIPS решением данной задачи является следующий план, который в среде CLIPS можно отследить также средствами трассировки выполнения программы (Execution/Watch, Execution /Step, Window/Facts):

```
movetotable(on(C A)&clear(C)
=>assert(on(C Table))&assert(clear(A))&retract(on(C A)))
move(on(C Table)&clear(B)&clear(C)
     =>assert(on(B C))&retract(on(B Table))
     &retract(clear(C)))
move(on(B C)&clear(A)&clear(B)
```

```
    =>assert(on(A B))&retract(on(A Table))
    &retract(clear(B))).
```

Рассмотрим описание задачи планирования еще на одном примере – задаче организации перевозок с помощью воздушного грузового транспорта, в которой предусматривается загрузка и разгрузка грузов в самолеты и из самолетов, а также перелет из одного места в другое. Эта задача может быть определена с помощью трех правил: `load`, `unload`, `fly`. Их совокупности действий влияют на значения двух предусловий - условных элементов: УЭ `(in (c ?c)(p ?p))` означает, что груз `?c` находится в самолете `?p`, а УЭ `(at (x ?x)(a ?a))` означает, что объект `?x` (самолет или груз) находится в аэропорту `?a`.

Ниже приведена полная реализация задачи организации перевозок на языке CLIPS:

```
(deftemplate in
    (slot c (type SYMBOL))
    (slot p (type SYMBOL))
)
(deftemplate at
    (slot x (type SYMBOL))
    (slot a (type SYMBOL))
)
(deftemplate cargo
    (slot name-cargo (type SYMBOL))
)
(deftemplate plane
    (slot name-plane (type SYMBOL))
)
(deftemplate airport
    (slot name-airport (type SYMBOL))
```

```
)
(deftemplate goal
    (slot current-task (type SYMBOL))
)
(deffacts init
    (at (x C1)(a SFO))
    (at (x C2)(a JFK))
    (at (x P1)(a SFO))
    (at (x P2)(a JFK))
    (cargo (name-cargo C1))
    (cargo (name-cargo C2))
    (plane (name-plane P1))
    (plane (name-plane P2))
    (airport (name-airport JFK))
    (airport (name-airport SFO))
)
(defrule init-system
    (initial-fact)
=>
    (assert (goal(current-task find)))
)
(defrule load
    ?atca <- (at (x ?c)(a ?a))
    (at (x ?p)(a ?a))
    (cargo (name-cargo ?c))
    (plane (name-plane ?p))
    (airport (name-airport ?a))
    (goal (current-task find))
=>
    (assert (in (c ?c)(p ?p)))
```

```
    (retract ?atca)
)
(defrule unload
    ?incp <- (in (c ?c)(p ?p))
    (at (x ?p)(a ?a))
    (cargo (name-cargo ?c))
    (plane (name-plane ?p))
    (airport (name-airport ?a))
    (goal (current-task find))
=>
    (assert (at (x ?c)(a ?a)))
    (retract ?incp)
)
(defrule fly
    ?atpfrom <- (at (x ?p)(a ?from))
    (in (c ?c)(p ?p))
    (plane (name-plane ?p))
    (cargo (name-cargo ?c))
    (airport (name-airport ?from))
    (airport (name-airport ?to))
    (test (not (eq ?from ?to)))
    (goal (current-task find))
=>
    (assert (at (x ?p)(a ?to)))
    (retract ?atpfrom)
)
(defrule finish-process
  (declare (salience 1000))
    (at (x C1)(a JFK))
    (at (x C2)(a SFO))
```

```
    ?goal-ptr <- (goal (current-task find))
=>
    (retract ?goal-ptr)
    (printout t "Solution found" crlf)
)
```

Следующий план, полученный в среде CLIPS при использовании стратегии “вширь” (breadth), является решением данной задачи:

```
load(at(C1 SFO)&at(P1 SFO)
    =>assert(in(C1 P1))&retract(at(C1 SFO)))
load(at(C2 JFK)&at(P2 JFK)
    =>assert(in(C2 P2))&retract(at(C2 JFK)))
fly(at(P1 SFO)&in(C1 P1)
    =>assert(at(P1 JFK))&retract(at(P1 SFO)))
fly(at(P2 JFK)&in(C2 P2)
    =>assert(at(P2 SFO))&retract(at(P2 JFK)))
unload(in(C1 P1)&at(P1 JFK)
    =>assert(at(C1 JFK))&retract(in(C1 P1)))
unload(in(C2 P2)&at(P2 SFO)
    =>assert(at(C2 SFO))&retract(in(C2 P2)))
```

Планирование на основе поиска в пространстве состояний

Наиболее простой алгоритм планирования – поиск в пространстве состояний. Поскольку описания действий в задаче планирования определяют и антецеденты правил, и их консеквенты, существует возможность организовать поиск в обоих направлениях, либо в прямом, от начального состояния, либо в обратном, от цели. Кроме того, явные представления действий и целей могут использоваться для автоматического вывода эффективных эвристик.

Первый подход, поиск в прямом направлении, часто называют **прогрессивным планированием**. Планировщик начинает с начального состояния задачи и рассматривает последовательности действий до тех пор, пока не находит последовательность, позволяющую достичь целевого состояния.

Действиями, применимыми в некотором состоянии, являются такие совокупности действий, для которых их УЭ, составляющие антецеденты соответствующих правил, удовлетворяются. Состояние-преемник, являющееся результатом выполнения совокупности действий, вырабатывается путем добавления положительных литералов результата (фактов, добавляемых к базе знаний оператором `assert`) и удаления отрицательных литералов результата (фактов, удаляемых из базы знаний оператором `retract`) с возможным применением унификатора из антецедентов. Такая функция определения состояния-преемника является единственной. После выработки каждого нового состояния осуществляется проверка, удовлетворяет ли текущее состояние цели.

При отсутствии функциональных символов пространство состояний задачи планирования является конечным, поэтому алгоритмом планирования может быть любой алгоритм поиска в графе, который является полным.

На эффективность прямого поиска влияют следующие факторы:

− при прямом поиске рассматриваются все действия, применимые из каждого состояния, в том числе и действия, не относящиеся к делу;

− без хорошей эвристики прямой поиск приводит к чрезмерному увеличению объема вычислений. Например, при решении задачи воздушных грузовых перевозок с 10 аэропортами, где в каждом аэропорту имеется 5 самолетов и 20 единиц груза, с целью перемещения всех грузов из аэропорта `A` в аэропорт `B` образуется дерево поиска, имеющее около 1000^{41} узлов.

Основным преимуществом обратного поиска (**регрессивного планирования**) является то, что он позволяет рассматривать только релевантные действия, то есть действия, которые достигают хотя бы одного из конъюнктов цели. Например, целью задачи по перевозке грузов с 10 аэропортами была доставка 20 единиц груза в аэропорт `B`, а именно:

```
at(C1 B)&at(C2 B)&…&at(C20 B)
```

Рассмотрим конъюнкт `at(C1 B)`. Двигаясь в обратном направлении можно найти только одно действие (правило), имеющее в консеквенте такой конъюнкт в операторе `assert`:

```
unload(in(C1 ?p)&at(?p B)=>assert(at(C1 B))&retract(in(C1 ?p)))
```

Имеется много нерелевантных действий, приводящих к целевому состоянию. Если решение существует, то оно должно быть найдено с помощью обратного поиска, допускающего только релевантные действия. Данное ограничение обеспечивает для обратного поиска гораздо более низкий коэффициент ветвления по сравнению с прямым поиском. Например, в данной задаче с грузовыми перевозками количество действий в прямом направлении около 1000, а в обратном только 20.

Кроме того, искомое действие не должно отменять какой-либо конъюнкт цели (быть совместимым). Например, действие `load(…=>assert(in(C2 ?p)&retract(at(C2 B)))` несовместимо с текущей целью, поскольку оно отменяет конъюнкт `at(C2 B)` цели.

Опишем общий процесс формирования состояний-преемников для обратного поиска. Пусть для цели `G` имеется релевантное и совместимое действие `A`. Состояние-преемник определяется следующим образом:

– любые положительные (`assert`) результаты действия `A`, которые имеются в цели `G`, удаляются;

– добавляется каждый условный элемент из `A`, если он еще не присутствует в состоянии-преемнике, с возможным применением подстановки переменных.

Завершение работы происходит после выработки такого описания преемника, которое соответствует начальному состоянию задачи планирования. Например, для рассматриваемой задачи преемник после подстановки может выглядеть следующим образом:

```
in(C1 P12)&at(P12 B)&at(C2 B)&…&at(C20 B).
```

Ни прямой, ни обратный поиск не могут быть эффективными без хорошей эвристической функции.