Практическая работа № 4

Реализация поиска в пространстве состояний

Цель работы: Реализация в среде CLIPS задачи поиска в пространстве состояний и анализ ее решения.

Введение

Одной из классических задач ИИ, рассматриваемых при построении и анализе алгоритмов поиска является известная головоломка о фермере, которому необходимо переправить на другой берег реки лису, козу и капусту. Он располагает двуместной лодкой, т.е. может перевозить только по одному объекту. При этом нельзя оставлять на берегу лису с козой и козу с капустой, т. к. в этом случае первая из них съест вторую.

Общие сведения

Как известно, постановка задачи поиска в пространстве состояний в общем случае предполагает описание *исходного состояния*, *множества операторов* перехода в пространстве состояний и множества целевых состояний (процедуры определения целевого состояния). Рассмотрим эти компоненты для данной задачи.

Представление состояний в пространстве состояний и вершин в дереве поиска. Каждое *состояние* в пространстве состояний *определяется нахождением каждого персонажа/объекта* (фермера (farmer), лисы (fox), козы (goat) и капусты (cabbage*)) на одном из двух берегов* (shore-1 или shore-2). Таким образом, состояние можно представить неупорядоченным фактом, содержащим слоты для задания местоположения каждого персонажа (объекта): farmer-location, fox-location, goat-

location и cabbage-location. Эти слоты могут принимать символьные значения shore-1 и shore-2.

Поскольку поиск выполняется по дереву поиска (ДП), при разработке программы необходимо представлять вершины ДП. Каждая вершина ДП, помимо описания некоторого состояния, должна содержать также дополнительную информацию*: ссылку на родительскую вершину*, *глубину вершины* и *последнее перемещение*. Последнее перемещение определяет с кем/чем переправлялся фермер последний раз и может принимать следующие символьные значения: no-move, alone, fox, goat и cabbage.

Таким образом, для представления вершин ДП можно использовать неупорядоченный факт, определяемый следующим шаблоном:

```
(deftemplate status
(slot  farmer-location  (type SYMBOL)  (allowed-symbols shore-1 shore-2))
(slot fox-location  (type SYMBOL)  (allowed-symbols shore-1 shore-2))
(slot goat-location  (type SYMBOL)  (allowed-symbols shore-1 shore-2))
(slot cabbage-location  (type SYMBOL)  (allowed-symbols shore-1 shore-2))
(slot  parent  (type FACT-ADDRESS SYMBOL)  (allowed-symbols no-parent))
(slot  search-depth  (type INTEGER) (range 1 ?VARIABLE))
(slot last-move  (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage)))
```

Исходным является *состояние*, в котором все действующие лица (и лодка) находятся на первом берегу (shore-1). Соответствующая (корневая) вершина в ДП не имеет родительской вершины, имеет глубину 1 и не имеет последнего перемещения (no-move). Таким образом, исходное состояние может быть представлено следующим фактом:

```
(deffacts initial-positions
(status (search-depth 1)
(parent no-parent)
(farmer-location shore-1)
(fox-location shore-1)
(goat-location shore-1)
(cabbage-location shore-1)
(last-move no-move)))
```

Операторы перехода в пространстве состояний. *Множество операторов* перехода для данной задачи включает:

- перемещение с одного берега на другой одного фермера (move-alone);
- перемещение фермера с лисой (move-with-fox);
- перемещение фермера с козой (move-with-goat);
- перемещение фермера с капустой (move-with-cabbage).

При реализации программы в среде CLIPS операторы удобно представлять правилами. При этом *в левой части правил* должны распознаваться условия применимости данного оператора и фиксироваться (связываться) параметры конкретного состояния: указатель (адрес) на текущую вершину, местонахождение действующих лиц, затрагиваемых данным оператором, и глубина поиска.

В правой части правила должна порождаться новая вершина, являющаяся потомком текущей в случае применения данного оператора и устанавливаться ее параметры: глубина, новое местонахождение действующих лиц, ссылка на родительскую вершину и последнее перемещение. Новую вершину удобно порождать путем дублирования текущей с изменением значений некоторых параметров. Пример правила для перемещения фермера с лисой:

```
(defrule move-with-fox                                  ; правило «перемещение с лисой»
 ?node <- (status (search-depth ?num) ; фиксация адреса текущей вершины и ее глубины
          (farmer-location ?fs)                  ; фиксация текущего местонахождения фермера
          (fox-location ?fs))                    ; лиса на том же берегу, что и фермер
 (opposite-of ?fs ?ns)              ; связывание значения противоположного берега
 =>
 (duplicate ?node                                       ; создать новую вершину дублированием
    (search-depth =(+ 1 ?num))  ; установить ее глубину инкрементом текущей
    (parent ?node)              ; установить в качестве родительской вершины текущую
    (farmer-location ?ns)                  ; установить новое местонахождение фермера
    (fox-location ?ns)                     ; установить новое местонахождение лисы
    (last-move fox)))                      ; установить тип последнего перемещения
```

Для фиксации (привязки) текущего берега и связывания переменной ?ns значением противоположного берега в левой части правила используется условный элемент (opposite-of ?fs ?ns). Значение переменной ?ns используется в правой части

правила для установки нового местонахождения персонажей в результате выполнения оператора. Для использования такого элемента необходимо заранее определить отношение opposites-of между берегами с помощью конструкции:

```
(deffacts opposites
(opposite-of shore-1 shore-2)
(opposite-of shore-2 shore-1))
```

Ограничения на возможные состояния. Процесс поиска может приводить в *запрещенные состояния*, в которых лиса ест козу или коза ест капусту. При попадании в запрещенные состояния соответствующие вершины должны удаляться. Например, лиса ест козу, если она находится с ней на одном берегу и на этом берегу нет фермера. Соответствующее правило можно записать так:

```
(defrule fox-eats-goat
?node <- (status (farmer-location ?s1) ; фиксируется адрес вершины и положение фермера
                 (fox-location ?s2&~?s1)          ; лиса и фермер на разных берегах
                 (goat-location ?s2))             ; коза на том же берегу, что и лиса
  =>
  (retract ?node))                                ; удалить вершину
```

Правило, определяющее состояние, в котором коза ест капусту, записывается аналогично.

Необходимо также распознавать *ситуации зацикливания* процесса *поиска*, т.е. повторного попадания в уже пройденное состояние. Для этого новое состояние должно сравниваться с ранее достигнутыми. Если имеется состояние с меньшей глубиной и точно таким же местоположением всех персонажей, то новая вершина должна удаляться. Соответствующее правило представлено ниже:

```
(defrule circular-path
(status (search-depth ?sd1)
        (farmer-location ?fs)
(fox-location ?xs)
        (goat-location ?gs)
        (cabbage-location ?cs))
```

```
?node <- (status (search-depth ?sd2&:(< ?sd1 ?sd2))
    (farmer-location ?fs)
        (fox-location ?xs)
    (goat-location ?gs)
        (cabbage-location ?cs))
=>
(retract ?node))
```

Первая часть антецедента этого правила сопоставляется с некоторой вершиной и фиксирует (в переменной ?sd1) ее глубину, а также местоположение всех персонажей – фермера, лисы, козы и капусты – соответственно в переменных ?fs, ?xs, ?gs и ?cs. *Вторая часть антецедента* сопоставляется с вершиной, имеющей большую глубину и точно такое же состояние (местоположение персонажей). Адрес этой вершины фиксируется в переменной ?node, чтобы в консеквенте правила можно было удалить данную вершину.

Распознавание и вывод решения. *Решением* задачи является *последовательность перемещений* на лодке с берега на берег, переводящая исходное состояние в целевое. В данной задаче целевым является состояние, когда все находятся на втором берегу. При достижении целевого состояния должно быть выведено решение – последовательность перемещений. Однако каждая вершина в ДП (в том числе целевая) явно хранит лишь *последнее перемещение* и указатель на вершину-предка. Поэтому при обнаружении целевого состояния необходимо выполнить обратный проход от целевой вершины к корню ДП (исходному состоянию), чтобы восстановить полную последовательность перемещений. Таким образом, необходимо иметь *правило для распознавания целевого состояния и правило для построения решения* – последовательности операторов (перемещений) переводящих исходное состояние в целевое.

Для представления последовательности перемещений, приводящих в некоторое состояние, удобно использовать факт на основе следующего шаблона:

```
(deftemplate moves
  (slot id (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
  (multislot moves-list (type SYMBOL) (allowed-symbols no-move alone fox goat cabbage))
```

Соответствующий факт содержит два слота:

1. Слот для идентификации вершины-предка. Значением слота является адрес вершины-родителя рассматриваемой вершины, или символьное значение no-parent для корневой вершины (у нее отсутствует родитель).

2. Мультислот moves-list для хранения последовательности перемещений, приводящих в данное состояние (вершину).

Правило распознавания целевого состояния должно активироваться, если имеется вершина, в которой все действующие лица находятся на втором берегу (shore-2). Правая часть правила должна удалять эту вершину и добавлять в базу данных факт, представляющий путь в соответствии с шаблоном moves. В этом факте слот идентификатора вершины должен указывать на вершину-предка целевой вершины, а мультислот moves-list содержать последнее перемещение из этой вершины-предка в целевую вершину. Тогда правило распознавания целевого состояния может быть записано следующим образом:

```
(defrule goal-test
?node <- (status (parent ?parent)
                 (farmer-location shore-2)
                 (fox-location shore-2)
                 (goat-location shore-2)
                 (cabbage-location shore-2)
          (last-move ?move))
=>
(retract ?node)
(assert (moves (id ?parent) (moves-list ?move))))
```

Появление в базе данных факта moves инициирует процесс обратного движения по ДП к корневой вершине (исходному состоянию) с построением пути-решения. *Правило построения решения* при каждом срабатывании реализует переход к родительской вершине, добавляя в мультислот moves-list факта moves соответствующее перемещение. Пример правила построения решения:

```
(defrule build-solution
 ?node <- (status (parent ?parent) ; фиксация адреса некоторой вершины ?node в ДП,
```

(last-move ?move)) ; ее вершины-родителя и последнего перемещения

?mv <- (moves (id ?node) (moves-list $?rest)) ; проверка, *есть ли вершина* moves ; с адресом ?node и, если «да», фиксация адреса ; факта и значения его мультислота moves-list

=>

(modify ?mv (id ?parent) (moves-list ?move ?rest))) ; модификация факта moves путем ; расширения списка перемещений и ; обновления предка

После завершения построения пути-решения, его необходимо отобразить на экране. Соответствующее правило должно сработать, когда обнаружится факт moves, не имеющий родителя (корневая вершина ДП). Правило вывода решения на экран может быть задано так:

```
(defrule SOLUTION::print-solution
 ?mv <- (moves (id no-parent) (moves-list no-move $?m)) ; для факта moves, не имеющего
                                                        ; предка фиксируется его адрес ?mv и значение ?m
                                                        ; мультислота moves-list – список перемещений
=>
(retract ?mv)                          ; факт ?mv удаляется
 (printout t t  "Solution found: " t t)  ; Печать сообщения «Решение найдено:»
 (bind ?length (length ?m)); ?length = длина списка перемещений ( переменной ?m)
 (bind ?i 1)                           ; ?i = 1
 (bind ?shore shore-2)                 ; ?shore = shore-2
 (while (<= ?i ?length)                ; Пока ?i <= ?length
   (bind ?thing (nth ?i ?m)); ?thing = значение i-го слота мультислота ?m (тип перемещения)
   (if (eq ?thing alone)               ; Если ?thing = alone
     then (printout t "Farmer moves alone to " ?shore "." t)
     else (printout t "Farmer moves with " ?thing " to " ?shore "." t))
   (if (eq ?shore shore-1)   ; Если ?shore = shore-1
```

```
    then (bind ?shore shore-2)   ;?shore = shore-2
    else (bind ?shore shore-1))  ;?shore = shore-1
  (bind ?i (+ 1 ?i))))                    ; ?i = ?i + 1
```

Порядок выполнения работы

1. Построить полное дерево поиска для данной задачи.

2. Разработать, используя среду CLIPS, программу решения данной головоломки. Программа должна быть построена по модульному принципу и состоять из трех модулей:

- основного (MAIN);
- контроля ограничений (CONSTRAINTS);
- вывода решения (SOLUTION).

Для объявления модуля используется конструкция defmodule, в которой указываются экпортируемые в другие модули или экспортируемые из других модулей конструкции. Например модуль MAIN экспортирует шаблон status:

```
(defmodule MAIN
  (export deftemplate status))
```

2.1. Модуль MAIN должен содержать:

- объявление шаблона состояния status;
- определение факта исходного состояния – initial-positions;
- определение факта отношения между берегами – opposites;
- определение правил генерации пути, соответствующих четырем операторам в пространстве состояний.

Имена всех конструкций модуля MAIN должны начинаться с префикса MAIN::. Например:

```
(deftemplate MAIN::status
. . .
)
```

2.2. Модуль контроля ограничений CONSTRAINTS должен импортировать из модуля MAIN шаблон status:

```
(defmodule CONSTRAINTS
        (import MAIN deftemplate status))
```

и содержать:

- два правила для распознавания запрещенных ситуаций fox-eats-goat и goat-eats-cabbage;
- правило для распознавания зацикливания пути – circular-path;

Имена всех конструкций модуля CONSTRAINTS должны начинаться с префикса CONSTRAINTS::. Например:

```
(defrule CONSTRAINTS::goat-eats-cabbage
. . .
)
```

У всех правил модуля CONSTRAINTS должно быть установлено свойство автофокусировки. Это делается так:

```
(defrule CONSTRAINTS::fox-eats-goat
(declare (auto-focus TRUE))
. . .
```

Если свойство автофокусировки правила установлено, то всякий раз при активации правила автоматически выполняется команда фокусировки на модуле, в котором определено данное правило.

2.3. Модуль вывода решения SOLUTION также должен импортировать из модуля MAIN шаблон status:

```
(defmodule SOLUTION
        (import MAIN deftemplate status))
```

и содержать:

- объявление шаблона факта-решения moves;
- правило распознавания целевого состояния goal-test;
- правило построения пути-решения – build-solution;
- правило вывода решения на экран – print-solution.

Имена всех конструкций модуля SOLUTION должны начинаться с префикса SOLUTION::. Например:

```
(defrule SOLUTION::print-solution
. . .
)
```

У правила распознавания целевого состояния должно быть установлено свойство автофокусировки:

```
(defrule SOLUTION:: goal-test
        (declare (auto-focus TRUE))
    . . .
```

3. Выполните программу в пошаговом режиме, проанализируйте и объясните ход поиска решения. В отчете необходимо привести трассу поиска решения.