Лабораторная работа №4: Реализация поиска в пространстве состояний

Цель работы

Формирование умения реализации в среде CLIPS задачи поиска в пространстве состояний и освоение способов анализа ее решения.

Основные теоретические положения

Введение

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

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* (крестьянина (`peasant`), волка (`wolf`), 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): `peasant-location`, `wolf-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 крестьянин последний раз и может принимать следующие символьные значения: `no-move`, `alone`, `wolf`, `goat` и `cabbage`.

Таким образом, для представления вершин ДП можно использовать неупорядоченный факт,

определяемый следующим шаблоном:

```
(deftemplate status
  (slot peasant-location (type SYMBOL) (allowed-symbols shore-1 shore-2))
  (slot wolf-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 wolf 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)
          (peasant-location shore-1)
          (wolf-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 крестьянина (`move-alone`);
- перемещение крестьянина с волком (`move-with-wolf`);
- перемещение крестьянина с козой (`move-with-goat`);
- перемещение крестьянина 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 фиксироваваться (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 крестьянина с волком:

```
(defrule move-with-wolf "Правило перемещения с волком"
  ?node <- (status (search-depth ?num)     ; фиксация адреса текущей вершины и
```

```
ее глубины
                    (peasant-location ?fs)  ; фиксация текущего местонахождения
крестьянина
                    (wolf-location ?fs))    ; волк на том же берегу, что и
крестьянина
  (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
    (peasant-location ?ns)      ; установить новое местонахождение крестьянина
    (wolf-location ?ns)         ; установить новое местонахождение волка
    (last-move wolf)))          ; 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 *запрещённые состояния*, в которых волк ест козу или коза ест капусту. При попадании в запрещенные состояния соответствующие вершины должны удаляться. Например, волк ест козу, если он находится с ней на одном берегу и на этом берегу нет крестьянина. Соответствующее правило можно записать так:

```
(defrule wolf-eats-goat
  ?node <- (status (peasant-location ?s1)    ; фиксируется адрес вершины и
положение крестьянин
                   (wolf-location ?s2&~?s1)  ; волк и крестьянин на разных
берегах
                   (goat-location ?s2))      ; коза на том же берегу, что и волк
=>
  (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)
          (peasant-location ?ps)
          (wolf-location ?ws)
          (goat-location ?gs)
          (cabbage-location ?cs))
  ?node <- (status (search-depth ?sd2&:(< ?sd1 ?sd2))
                   (peasant-location ?ps)
                   (wolf-location ?ws)
                   (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 – крестьянина, волка, козы и капусты – соответственно в переменных `?ps`, `?ws`, `?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 wolf
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)
                   (peasant-location shore-2)
                   (wolf-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) do         ; 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 "Peasant moves alone to " ?shore "." t)
       else (printout t "Peasant 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
```

Постановка задачи

Необходимо построить полное дерево поиска для задачи о крестьянине, которому необходимо переправить на другой берег реки волка, козу и капусту, разработать на продукционном языке CLIPS модульную программу решения данной задачи и проанализировать ход поиска решения, выполнив программу в пошаговом режиме.

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:
    1. основного (`MAIN`);
    2. контроля ограничений (`CONSTRAINTS`);
    3. 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))
    ```

3. Модуль MAIN должен содержать:
    1. объявление шаблона состояния `status`;
    2. определение факта исходного состояния – `initial-positions`;
    3. определение факта отношения между берегами – `opposites`;
    4. определение правил генерации пути, соответствующих четырем операторам в пространстве состояний.
    Имена всех конструкций модуля `MAIN` должны начинаться с префикса `MAIN::`. Например:

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

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

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

   и содержать:

   1. два правила для распознавания запрещенных ситуаций wolf-eats-goat и goat-eats-cabbage;
   2. 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::wolf-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.

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

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

   и содержать:

   1. объявление шаблона факта-решения moves;
   2. правило распознавания целевого состояния goal-test;
   3. правило построения пути-решения – build-solution;
   4. 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))
  ...
)
```

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

Содержание отчёта

- Цель работы.
- Краткое изложение основных теоретических понятий.
- Постановка задачи с кратким описанием порядка выполнения работы.
- Дерево решений.
- Трассировка решения, оформленная в виде таблицы, с краткими выводами.
- Результаты работы программы.
- Общий вывод по проделанной работе.
- Код программы.