Логическое программирование

Кевролетин В.В. группа c8403a(246)

11 November 2012

Содержание

1 Задание 4

1.1 Условие

(Вариант 1.) The missionaries and cannibals problem is a good example of a puzzle that can be analyzed according to the search superstructure given above. The problem involves three missionaries and three cannibals, all six of whom are originally on one side of a river. There is one boat that will be used to ferry the missionaries and cannibals to the other side of the river. The boat holds two occupants at most, and there is no way to send the boat across the river without having at least one occupant in the boat. The threat is that, if the cannibals outnumber the missionaries in any circumstance, then the cannibals will cook and eat the missionaries (so the fable goes). Use the search superstructure to design a Prolog program that searches for ways to ferry all six persons to the other side of the river. Your program should be able to calculate two distinct minimal solutions each involving eleven boat trips across the river.

(Вариант 2.) 3 Missionaries and 3 Cannibals come to a river and wish to cross. There is a boat and it can transport at most 2 people. The problem this time is that if the missionaries ever outnumber the cannibals on the bank then the cannibals are converted - a fate worse than death!

1.2 Решение

1.3 Вариант 1

Чтобы найти минимальное решение будем использовать поиск в ширину. Поиск в ширину требует реализацию очереди, в которой хранятся достигнутые, но еще не обработанные состояния. Так же необходим специальный предикат, возвращающий список состояний, в которые можно перейти из данного состояния. Для реализации очередей используются техника “Difference lists”, реализация взята из книги “Искусство Пролога” листинг 15.11. Вместо предиката findall используется версия для “Difference lists” из листининга 16.3. Код приведен в конце данной работы.

1.3.1 Исходный код

Представим состояние в программе термом следующей структуры:

```
state(WhereTheBoatIs, MissionariesOnLeft, CannibalsOnLeft)
```

- WhereTheBoatIs - на каком берегу находится лодка (right, left)
- MissionariesOnLeft - число миссионеров на левом берегу
- CannibalsOnLeft - число канибалов на левом берегу

Ход представим таким термом

```
move(MissionariesInBoat, CannibalsInBoat)
```

- MissionariesInBoat - число, сколько миссионеров перевести на другой берег
- CannibalsInBoat - число, сколько канибалов перевести на другой берег

Так как необходимо восстановить последовательность ходов, то в программе вместе с состоянием будем хранить последовательность действий, которая приводит в это состояние из начального:

```
batch(State, Moves)
```

- State - состояние(см. выше)
- Moves - список ходом(см. выше)

Поиск в ширину реализуется следующим предикатами:

- solve_bfs(Moves) - возвращает решение в виде списка ходов
- solve_bfs(Queue, Visited, Res) - извлекает очередное состояние из очереди Queue, добавляет состояние, в которые можно перейти в другой конец очереди
  - Queue - очередь состояний, ожидающих обработку
  - Visited - посещенные состояния
  - Res - ответ, последовательность ходов
- enqueue_states(Batch, Visited, Xs\ Ys, Xs\ Zs) - добавляет в очередь состояния в которые можно перейти из текущего состояния
  - Batch - пара состояние+последовательность ходов
  - Visited - посещенные состояния
  - Xs\ Ys - очередь до добавления новых состояний
  - Xs\ Zs - очередь после добавления новых состояний

```
solve_bfs(Moves) :-
        initial_state(Init),
        enqueue(batch(Init, []), Q\Q, Q1),
        solve_bfs(Q1, [], Moves).

solve_bfs(Q, _, _) :- empty(Q), !, fail.
solve_bfs(Q, _, Moves) :-
        dequeue(batch(State, Moves), Q, _),
        final_state(State).
solve_bfs(Q, Visited, Res) :-
        dequeue(batch(State, Moves), Q, Q1),
        enqueue_states(batch(State, Moves), [State | Visited], Q1, Q2),
        solve_bfs(Q2, [State | Visited], Res).

empty([]\[]).

enqueue_states(Batch, Visited, Xs\Ys, Xs\Zs) :-
        find_all_dl(NewBatch, find_legal_move(Batch, Visited, NewBatch), Ys\Zs), !.
```

Осталось определить предикаты $\text{find}_{\text{legalmove}}$, $\text{initial}_{\text{state}}$, $\text{final}_{\text{state}}$.

- find_legal_move(Batch, Visited, Res) - возвращает новое до этого непосещенное состояние
  - Batch - пара состояние/ходы
  - Visited - список посещенных состояний
  - Res - пара состояние/ходы, новое состояние, в которое можно перейти
- move(State, Move) - допустимый по условию задачи ход
- update(OldState, Move, NewState) - применение хода к состоянию для получения нового состояния
- **legal(State)** - предикат, проверяющий выполнение условия "Ни на каком берегу канибалов не должно быть больше, чем миссионеров".

```
find_legal_move(batch(State, Moves), Visited, batch(State1, [Move|Moves])) :-
        move(State, Move),
        update(State, Move, State1),
        legal(State1),
        \+ member(State1, Visited).

initial_state(state(left, 3, 3)).

final_state(state(right, 0, 0)).

move(state(left,  M, _), move(1, 0)) :- M >= 1.
move(state(left,  _, C), move(0, 1)) :- C >= 1.
move(state(left,  M, C), move(1, 1)) :- M >= 1, C >= 1.
move(state(left,  M, _), move(2, 0)) :- M >= 2.
move(state(left,  _, C), move(0, 2)) :- C >= 2.
move(state(right, M, _), move(1, 0)) :- (3 - M) >= 1.
move(state(right, _, C), move(0, 1)) :- (3 - C) >= 1.
```

```
move(state(right, M, C), move(1, 1)) :- (3 - M) >= 1, (3 - C) >= 1.
move(state(right, M, _), move(2, 0)) :- (3 - M) >= 2.
move(state(right, _, C), move(0, 2)) :- (3 - C) >= 2.

update(state(left, M0, C0), move(MB, CB), state(right, M, C)):-
  M is M0 - MB, C is C0 - CB.
update(state(right, M0, C0), move(MB, CB), state(left, M, C)):-
  M is M0 + MB, C is C0 + CB.

legal(state(_, 3, _)):-!.
legal(state(_, 0, _)):-!.
legal(state(_, M, M)).
```

1.3.2 Тесты

```
?- solve_bfs(X).
X = [move(1, 1), move(1, 0), move(0, 2), move(0, 1), move(2, 0), move(1, 1),
     move(2, 0), move(0, 1), move(0, 2), move(1, 0), move(1, 1)]
X = [move(0, 2), move(0, 1), move(0, 2), move(0, 1), move(2, 0), move(1, 1),
     move(2, 0), move(0, 1), move(0, 2), move(1, 0), move(1, 1)]
X = [move(1, 1), move(1, 0), move(0, 2), move(0, 1), move(2, 0), move(1, 1),
     move(2, 0), move(0, 1), move(0, 2), move(0, 1), move(0, 2)]
X = [move(0, 2), move(0, 1), move(0, 2), move(0, 1), move(2, 0), move(1, 1),
     move(2, 0), move(0, 1), move(0, 2), move(0, 1), move(0, 2)]
false.
?-
```

Для наглядности, можно посмотреть последовательность состояний, к примеру для 1го решения:

```
show_solution([Init|States]) :-
        solve_bfs(Moves),
        initial_state(Init),
        map(Moves, Init, States).

map([Move|Xs], State, [NewState|Res]) :-
        update(State, Move, NewState),
        map(Xs, NewState, Res).
map([], _, []).

?- show_solution(X).
X = [state(left, 3, 3), state(right, 2, 2), state(left, 3, 2), state(right, 3, 0),
     state(left, 3, 1), state(right, 1, 1), state(left, 2, 2), state(right, 0, 2),
     state(left, 0, 3), state(right, 0, 1), state(left, 1, 1), state(right, 0, 0)]

?-
```

1.4 Вариант 2

Вротой вариант задачи полность аналогичен первому вариант, следует только поменять канибалов и миссионеров местами(договориться, что в структуре state(WhereTheBoatIs, MissionariesOnLeft, CannibalsOnLeft) канибалы теперь на второй позиции, а миссионеры на третей. Либо можно изменить ограничения: Вместо

```
legal(state(_, 3, _)):-!.
legal(state(_, 0, _)):-!.
legal(state(_, M, M)).
```

Нужно

```
legal(state(_, _, 3)):-!.
legal(state(_, _, 0)):-!.
legal(state(_, M, M)).
```

1.4.1 Дополнение

- Queue

```
/*
    queue(S) :-
        S is a sequence of enqueue and dequeue operations,
        represented as a list of terms enqueue(X) and dequeue(X).
*/

:- op(40,xfx,\).

queue(S) :- queue(S,Q\Q).

queue([enqueue(X)|Xs],Q) :-
        enqueue(X,Q,Q1), queue(Xs,Q1).
queue([dequeue(X)|Xs],Q) :-
        dequeue(X,Q,Q1), queue(Xs,Q1).
queue([], _).

enqueue(X,Qh\[X|Qt],Qh\Qt).
dequeue(X,[X|Qh]\Qt,Qh\Qt).

%       Program 15.11:  A queue process
```

- find_all_dl

```
:- op(40,xfx,\).

find_all_dl(X, Goal, _) :-
        asserta('$instance'('$mark')),
        Goal,
        asserta('$instance'(X)),
        fail.
find_all_dl(X, _, Xs\Ys) :-
        retract('$instance'(X)),
        reap(X,Xs\Ys), !.

reap(X,Xs\Ys) :-
        X \== '$mark',
```

```
        retract('$instance'(X1)), ! ,
        reap(X1,Xs\[X|Ys]).
reap('$mark',Xs\Xs).

%       Program 16.3 : Implementing an all-solutions predicate using
%                      difference-lists, assert and retract
```