УДК 004.89

БЫСТРЫЙ ЛОГИЧЕСКИЙ ВЫВОД В СРЕДЕ ПРОГРАММИРОВАНИЯ VISUAL PROLOG

И.А. Бессмертный

В статье обсуждается реализация логического вывода в системах искусственного интеллекта, построенных на продукционной модели знаний, с использованием теоретико-множественных операций в среде программирования Visual Prolog. Предлагается набор предикатов, реализующих быстрое выполнение операций реляционной алгебры. Приводятся результаты натурных экспериментов, подтверждающих существенное, не менее чем на три порядка, ускорение логического вывода для баз знаний с большим числом фактов.
Ключевые слова: искусственный интеллект, реляционная алгебра, Prolog

Введение

Язык программирования Prolog, имеющий встроенный механизм исчисления предикатов MODUS PONENS, является привлекательным инструментом для построения систем искусственного интеллекта, поскольку конструкции на языке Prolog максимально приближены к логическим выражениям, которыми может быть описана решаемая проблема. Обратной стороной изящности Prolog-программ является их низкая производительность в связи с тем, что резолюция цели достигается путем «наивного» логического вывода, т.е. перебора всех возможных вариантов, а значит, обхода всех ветвей на дереве решений.

Существующие методы ускорения логического вывода в системах искусственного интеллекта, построенных на продукционной модели знаний, связаны с предварительной обработкой правил. Наиболее известным является алгоритм Rete [1], в котором для каждого условия правила строится список фактов, удовлетворяющих данному условию. Списки фактов образуют префиксное дерево, которое должно создаваться для каждого правила. Таким образом, быстродействие обеспечивается многократным дублированием фактов. Узким местом алгоритма Rete является необходимость обновления всех префиксных деревьев при каждой модификации базы фактов, что обусловливает ограниченное применение алгоритма.

В работе [2] предлагается индексация и предварительный отбор фактов, релевантных правилам. Эффективность данного подхода существенно зависит от доли фактов, участвующих в каждом правиле: чем она выше, тем меньше выигрыш от использования индексов.

Необходимость в создании машины логического вывода (интеллектуального агента) предусмотрена в концепции Глобальной семантической сети (Semantic Web) [3], которая базируется на продукционной модели знаний. При этом набор операций над фактами, предусматриваемый в языках представления правил, в частности, SWRL, является ограниченным, поскольку элементами правил (атомами) могут быть только $C(x)$, $P(x, y)$, *sameAs*(x, y) или *differentFrom*(x, y), где C – свойство, P – отношение, x, y – переменные [4]. Как показано в работе [2], интерпретация правил может быть заменена операциями реляционной алгебры над кортежами значений переменных, хранящихся в фактах. Хранение фактов и логический вывод могут быть перенесены в среду систем управления базами данных (СУБД). Использование СУБД для построения систем искусственного интеллекта порождает новые проблемы: необходимость трансляции правил в язык запросов SQL, преобразования имен и др. В этой связи представляет интерес логический вывод с использованием реляционных операций в среде языка программирования Prolog как наиболее удобной для манипулирования знаниями.

Реализация быстрых реляционных операций в среде Visual Prolog

Набор базовых операций реляционной алгебры включает в себя пересечение $X \cap Y$ (INTERSECT), вычитание $X–Y$ (DIFFERENCE), объединение $X \cup Y$ (UNION), соединение (JOIN), проекцию, дополнение (NOT) и фильтрацию (WHERE) [5]. Реализация над двумя списками $X=\{x\}$ и $Y=\{y\}$ операций пересечения, разности и объединения предполагает в среднем развертывание $n_x n_y/2$ вершин дерева поиска (список Y сканируется до первого появления искомого значения), где n_x, n_y – количество фактов в множествах X и Y соответственно. Для операции соединения развертывается $n_x n_y$ вершин, поскольку соединение представляет собой декартово произведение с фильтрацией. Операции дополнения и фильтрации в языке SWRL являются простыми, поскольку здесь не допускается вложенный поиск, и требуют развертывания только n_x вершин.

Ниже приведен текст предиката *list:: intersection*, входящего в состав библиотеки *list* языка Visual Prolog 7.2 (http://www.visual-prolog.com/). Здесь и далее вместо множеств мы будем говорить о списках. Во-первых, список – это агрегат данных, которым манипулирует Prolog; во-вторых, списки допускают повторяющиеся значения, сохранение которых необходимо в некоторых операциях.

```
    predicates
intersection: (Elem* ListA, Elem* ListB) → Elem* IntersectionAB.
    clauses
intersection([],_) = [].
intersection([X/Xs],Ys) = [X/intersection(Xs,Ys)]:-
```

```
isMember(X,Ys), !.
intersection([_|Xs],Ys) = intersection(Xs,Ys).
```

Предикат *list::isMember* вызывает извлечение в среднем $n_y/2$ элементов списка Y, для каждого элемента списка X, что и обусловливает низкую скорость работы данного предиката.

Отсортируем списки $X=\{x\}$ и $Y=\{y\}$ по возрастанию значений. В этом случае оба списка можно обрабатывать совместно, и дерево поиска будет состоять из n_x+n_y вершин. Таким образом, ожидаемое ускорение обусловлено переходом от квадратичной к линейной зависимости сложности поиска от числа фактов. Предикат нахождения пересечения отсортированных списков *intersectSorted* представлен ниже.

```
    predicates
intersectSorted : (Elem* ListX, Elem* ListY) → Elem* IntersectionXY.
    clauses
intersectSorted([],_) = [] :-!.
intersectSorted(_,[]) = [] :-!.
    intersectSorted([Y/Xs],[Y/Ys]) = [Y/intersectSorted(Xs,[Y/Ys])] :-!.
intersectSorted([X/Xs],[Y/Ys]) = intersectSorted([X/Xs],Ys) :- X>Y,!.
intersectSorted([_/Xs],[Y/Ys]) = intersectSorted(Xs,[Y/Ys]).
```

Похожим образом реализуется операция вычитания множеств $X–Y$ *differenceSorted*:

```
    predicates
differenceSorted : (Elem* ListX, Elem* ListY) → Elem* ListXExceptListY.
    clauses
differenceSorted([],_) = [] :-!.
differenceSorted(R,[]) = R :-!.
differenceSorted([X/Xs],[X/Ys]) = differenceSorted(Xs,[X/Ys]) :-!.
differenceSorted([X/Xs],[Y/Ys]) = [X/differenceSorted(Xs,[Y/Ys])] :- X<Y,!.
differenceSorted([X/Xs],[_Y/Ys]) = differenceSorted([X/Xs],Ys).
```

Операция объединения множеств состоит из суммы двух вычитаний множеств плюс их пересечение:

$$X \cup Y = (X-Y) + (Y-X) + (X \cap Y),$$

т.е. требует пять операций. Создание отдельного предиката *unionSorted* для операции объединения позволит сократить размерность данной задачи:

```
    predicates
unionSorted : (Elem* ListX, Elem* ListY) -> Elem* UnionXY.
    clauses
unionSorted([],R) = R :-!.
unionSorted(R,[]) = R :-!.
unionSorted([X/Xs],[X/Ys]) = unionSorted(Xs,[X/Ys]) :-!.
unionSorted([X/Xs],Ys) = [X/unionSorted(Xs,Ys)].
```

Наиболее часто в процессе интерпретации условий правил требуется соединение двух множеств кортежей (x_1, x_2, x_m) и $(y_1, y_2, ..., y_k)$ по совпадению значений переменных $x_i = y_j$. В реляционной алгебре данной функции соответствует операция INNER JOIN. Количество развертываемых вершин дерева поиска для данной операции в случае наивного вывода составляет $n_x n_y$,а для соединения отсортированных списков

$$N = n_x + n_y + SUM(n_{xi}n_{yj},\ x_i=y_j,\ (n_{xi} > 1 \vee n_{yj} > 1)), \quad (1)$$

где n_{xi}, n_{yj}, – количество повторяющихся значений в первом и втором списке соответственно. Если пренебречь возможностью повторения одних и тех же значений в обоих списках $(n_{xi}>1 \wedge n_{yj}>1)$, а также случаями $(n_{xi} >1 \wedge n_{yj}=0)$ и $(n_{xi}=0 \wedge n_{yj}>1)$, то формулу (1) можно упростить:

$$N = n_x + n_y + d_x + d_y, \quad (2)$$

где d_x, d_y, – число дубликатов значений в списках X и Y соответственно. Ниже представлен текст предиката, реализующего данную операцию над двумя множествами кортежей $\{(x_1, x_2)\}$ и $\{(y_1, y_2)\}$ по совпадению $x_1 = y_1$.

```
        predicates
 innerJoinSorted:(t{X,X}* ListX, t{X,X}* ListY, t{X,X,X}* JoinXY) → t{X,X,X}* JoinXYfinal.
 innerJoinSorted1:(t{X,X}, t{X,X}*, t{X,X}*, t{X,X,X}*) procedure (i,i,o,o).
        clauses
 innerJoinSorted([], _, Final) = Final :- !.
 innerJoinSorted([t(X1,X2)/Xresidue],Y,In)=innerJoinSorted(Xresidue,YtoLookUp,Out) :-
 innerJoinSorted1(t(X1, X2), Y, YtoLookUp, XY),
 Out = list::append(XY, In).

 innerJoinSorted1(_, [], [], []) :- !.
 innerJoinSorted1(t(X1,_X2),[t(Y1,Y2)/Yresidue], [t(Y1,Y2)/Yresidue],[]) :-X1<Y1, !.
 innerJoinSorted1(t(X1,X2), [t(Y1,_)|Yresidue],YlookUp,XY) :-
 X1 > Y1, !,
 innerJoinSorted1(t(X1,X2), Yresidue, YlookUp, XY).

 innerJoinSorted1(t(X1,X2),[t(Y1,Y2)/Yresidue],[t(Y1,Y2)/YlookUp], [t(X1,X2,Y2)/XY]) :-
 innerJoinSorted1(t(X1,X2),Yresidue,YlookUp,XY).
```

В табл. 1 обобщены данные о каждой из реляционных операций и их сложности, измеряемой числом развертываемых вершин дерева поиска.

Операция	Оператор в СУБД	Встроенный предикат в Visual Prolog	Сложность встроенного предиката	Сложность быстрого предиката
Пересечение $X \cap Y$	INTERSECT	*list::intersection*	$n_x n_y/2$	$n_x + n_y$
Разность $X–Y$	MINUS	*list::difference*	$n_x n_y/2$	$n_x + n_y$
Объединение $X \cup Y = (X-Y)+(Y-X)+(X \cap Y)$	UNION	*list::union*	$n_x n_y/2$	$n_x + n_y$
Соединение	JOIN	отсутствует	$n_x n_y$	$n_x + n_y + d_x + d_y$

Таблица 1. Сложность операций реляционной алгебры

Исследование быстродействия

Для исследования быстродействия использовались множества положительных чисел, полученные с помощью генератора случайных чисел. Предикат, формирующий список положительных чисел, представлен ниже.

```
        predicates
 generate : (positive, positive, positive*) procedure (i,i,o).
        clauses
 generate(0,_,[]) :- !.
 generate(N,Range,[Num1|Residue]) :- Num1=math::random(Range),
         generate(N-1,Range,Residue).
```

Наивный логический вывод был реализован итеративными предикатами, приведенными ниже. Здесь *f*1(*X*), *f*2(*X*) и *f*(*X*) – исходные и результирующий факт соответственно.

```
        % Пересечение
isec() :- f1(X), f2(X), assert(f(X)), fail.
isec().
        % Вычитание
idiff() :- f1(X), not(f2(X)), assert(f(X)), fail.
idiff().
        % Соединение
ijoin() :- f1(t(X1,X2)), f2(t(X1,Y2)), assert(f3(t(X1,X2,Y2))), fail.
ijoin().
```

Приведенные в табл. 2 и на рис. 1 результаты натурных экспериментов показывают, что логический вывод с использованием реляционных операций над отсортированными множествами на три-четыре порядка быстрее наивного вывода, в то время как предикаты реляционной алгебры дают ускорение не более, чем на порядок по сравнению с наивным выводом.

Число фактов	100000	200000	400000	800000	1600000
Время сортировки списков, с	3,1	4,5	14,4	30,6	65,89
Время сортировки кортежей, с	3,73	6,5	18,11	36	75
Пересечение списков, наивный вывод, с	360	1536	5800		
Пересечение списков, библ.предикат, с	1400	6530	22605		
Пересечение быстрый предикат, с	0,08	0,15	0,33	0,73	1,41
Объединение списков, библ.предикат, с	2020	7203	22563		
Объединение списков, быстрый предикат, с	0,09	0,28	0,56	1,13	2,13
Вычитание списков, быстрый предикат, с	0,09	0,17	0,36	1,2	2,34
Соединение кортежей, наивный вывод, с	316	1804	6600	23400	
Соединение кортежей, быстрый предикат, с	0,26	0,55	1,4	2,5	5,4

Таблица 2. Результаты натурных экспериментов

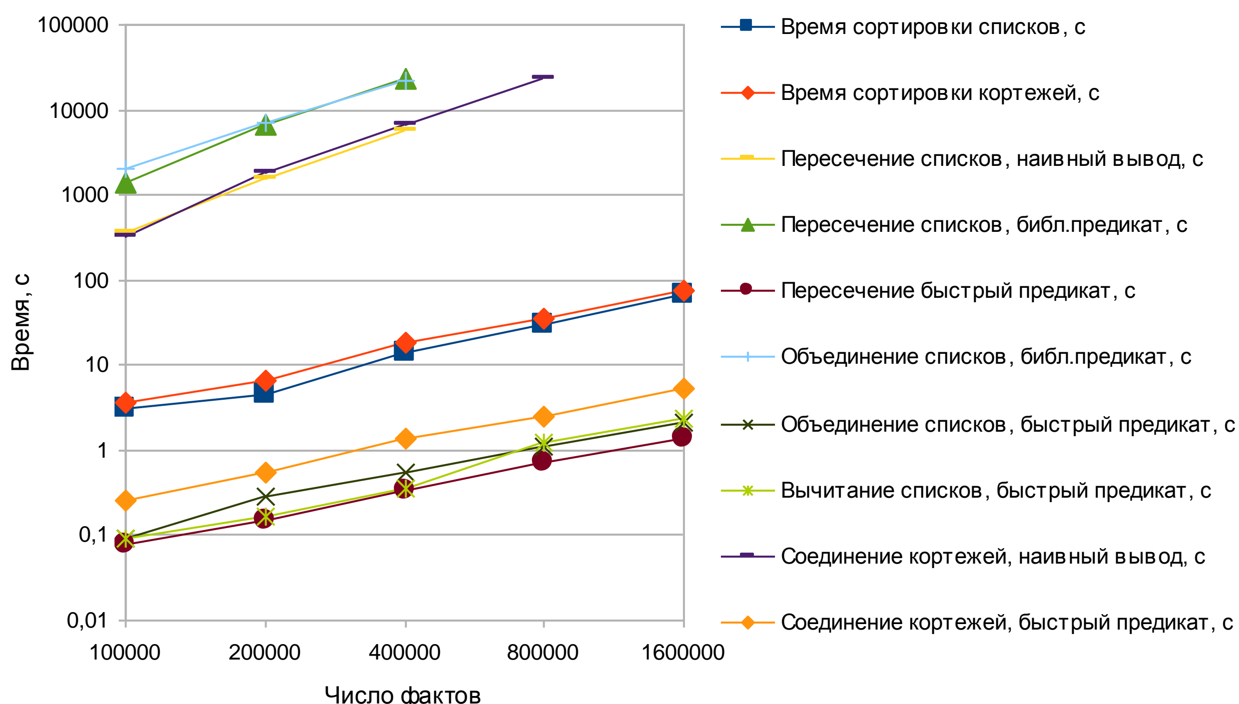

Рис.1. Результаты натурных испытаний методов логического вывода

Сортировка стандартным предикатом *list::sort* демонстрирует высокую скорость, поэтому даже если считать, что сортировка выполняется перед каждой реляционной операцией, выполнение таких операций над отсортированными списками более чем на два порядка быстрее, чем наивный логический вывод.

Предложенные здесь Prolog-предикаты не требуют больших объемов оперативной памяти, поскольку во всех предикатах хвостовая рекурсия (*tail recursion*) отсутствует. Объем памяти, занимаемый тестовой программой за вычетом памяти для хранения базы фактов, в течение всего процесса обработки тестовых фактов не превышает 10 Мб.

Натурные испытания позволили также проверить применимость формул, приведенных в табл. 1, для оценки времени вывода в предположении о том, что время вывода

$$T = tN, \quad (3)$$

где t – время развертывания одной вершины дерева поиска, N – число вершин. Рис. 2 и 3 демонстрируют, что данная формула может применяться как для наивного логического вывода, так и для операций реляционной алгебры, причем время t составляет порядка 4×10^{-8}с для наивного вывода, $2{,}85\times10^{-7}$с для стандартных предикатов, реализующих реляционные операции, и $4{,}4\times10^{-7}$с для быстрых предикатов.

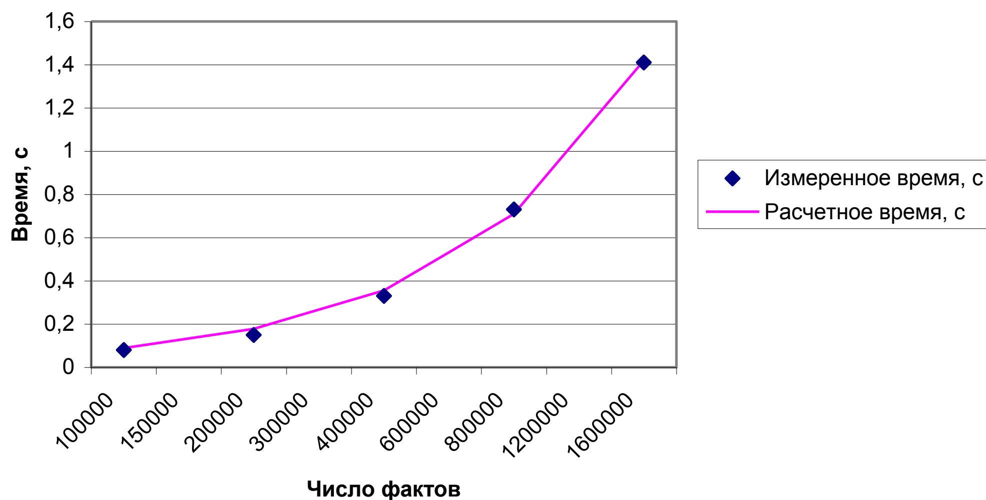

Рис. 2. Сравнение измеренного и расчетного времени вывода для операции быстрого пересечения

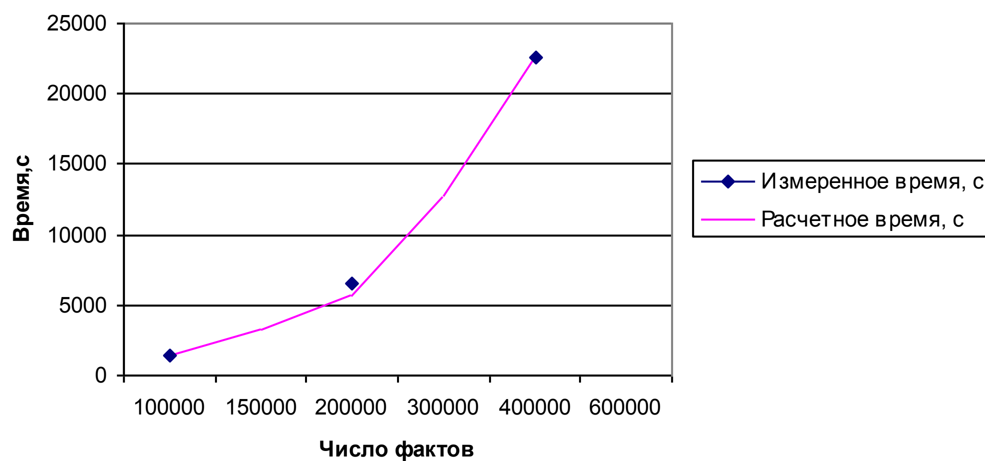

Рис. 3. Сравнение измеренного и расчетного времени вывода для операции list::intersection

Заключение

Замена наивного логического вывода операциями реляционной алгебры дает возможность решить проблему комбинаторной сложности задач поиска в системах искусственного интеллекта, построенных на продукционной модели знаний и, в частности, для создания интеллектуальных агентов для поиска решений в Глобальной семантической сети. В данной работе доказано, что среда программирования Prolog позволяет создавать эффективные предикаты, реализующие реляционные операции с высокой скоростью. На базе подхода, испытанного на базовых операциях реляционной алгебры, могут быть разработаны простые и эффективные предикаты для прочих операций, необходимость в которых может появиться при разработке интеллектуальных агентов для Глобальной семантической сети. Предикаты для операций реляционной алгебры, апробированные в среде Visual Prolog, могут быть перенесены на другие диалекты языка Prolog.

Литература

1. Forgy C.L. RETE: A fast algorithm for the many pattern / many object pattern match problem // Artificial Intelligence. – 1982. – Vol. 19. – P. 17–37.
2. Бессмертный И.А. Теоретико-множественный подход к логическому выводу в базах знаний // Научно-техн. вестник СПбГУ ИТМО. – 2010. – № 2. – С. 43–49.
3. Berners-Lee T., Hendler J., Lassila O. The Semantic Web // Scientific American Magazine. – May, 2001.
4. SWRL: A Semantic Web Rule Language. Combining OWL and RuleML. W3C Member Submission 21 May 2004 [Электронный ресурс]. – Режим доступа: http://www.w3.org/Submission/SWRL/, своб.
5. Дейт Дж. Введение в системы баз данных. – 8-е изд. – М.: Вильямс, 2006. – 1328 с.

Бессмертный Игорь Александрович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, кандидат технических наук, доцент, igor_bessmertny@hotmail.com