Media Engineering and Technology Faculty
German University in Cairo

Exploring Algorithms for Constrained String Enumeration

Bachelor Thesis

Author: Ahmad Adel Roshdy Soliman
Supervisor: Prof. Haythem O. Ismail
Reviewer: Prof. Haythem O. Ismail
Submission Date: 1 June, 2014

This is to certify that:

(i) the thesis comprises only my original work towards the Bachelor Degree

(ii) due acknowledgement has been made in the text to all other material used

_____

Name of the Author

1 June, 2014

Abstract

Constrained String Enumeration is an optimization problem motivated by the evaluation of conjunctive rules of logic programming, where the ordering of the inner predicates is essential to optimize the time efficiency of the evaluation. Given the set of variables used in a certain rule, tuples of the related variables in each inner predicate and the maximum number of instances to be used for each tuple, the maximum number of instances for such rule should be calculated. Exact and approximation algorithms that solves this problem are to be discussed.

Contents

Chapter 1

Introduction

1.1 Motivation

The ordering of the subgoals of conjunctive rules in logic programming can notably affect the efficiency of evaluation. Ordering them accordingly according to their number of matches can increase the efficiency of the evaluation of such a rule.

1.2 Constrained String Enumeration

An instance of the problem can be represented as a triple P = < V, A, f > where

1. V is a totally ordered set of n variables;
2. A is a system of subsets of V, that contain all singletons but not the empty set;
3. f is a function that maps every item of A to an integer.

V contains all the variables present in the rule or its subgoals, A contains tuples of variables that corresponds to the subgoals present in the rule including a subgoal for each variable by itself, and f is a function that gives the maximum number of matches for each set in A.

1.3 Aim of the project

The main aim is to explore different approaches to solve the problem and find an algorithm that is either exact and efficient enough, or to find an efficient approximation algorithm with high accuracy.

Chapter 2

Background

In this chapter, previous work done on the problem will be discussed, from the brute force search algorithm, to an improved version with state reduction, and finally an approximation algorithm.

2.1 Naive Search Algorithm

If every variable in V has a unique name, then every instance of such variable can be represented as the name plus a suffix representing the instance number. For example a variable named "X" that has 3 instances, they would be represented as {X1, X2, X3}.

A certain assignment of the variables in V can be represented by a string which is the concatenation of the instances used for each variable; for eg. "X2Y1Z3W2".

f* is a function that calculates the total number of possible patterns for a rule.

Algorithm: generate all possible strings, start searching and in each level try removing any string and check if all rules are satisfied, once satisfied the number of left strings is the answer.

2.2 Reducing the Search Space

To avoid reaching equivalent states, we enforce two rules:

1. Lexicographical order of strings.
2. Eliminate repeated permutations of strings.

2.3 Approximation Algorithm

Chapter 3

New Algorithms

3.1 Pattern-Based Search Algorithm

The main difference between this search algorithm and the naive one is that, rather than trying to remove a single string per level, try to remove a whole pattern satisfying a single rule, removing a whole set of strings abiding by the pattern in a single level.

To satisfy a rule r, we need to assure keeping only f(r) different patterns in the remaining strings, thus we need to remove Q(r) = f*(r)-f(r) patterns from that rule. In this search algorithm, we try to remove all combinations of Q(r) from the f*(r) patterns, however pruning some combinations to eliminate repetition of states.

Algorithm: For every non-singleton rule r, satisfy the rule by trying to remove any Q(r) patterns, finding the maximum number of strings left after satisfying all rules.

To avoid reaching equivalent states, we enforce two rules:

1. Lexicographical order of patterns.
2. Eliminate repeated permutations of patterns.

Both were used in [1] within the "Optimally Efficient Algorithm".

The first is enforced by choosing patterns in increasing lexicographical order to avoid trying different permutations of patterns. And the second is enforced by assuring that any instance numbered X for any variable to be used in a pattern, all instances 1 till X-1 of that variable should have been used in some other patterns while satisfying any rule.

3.2 A Greedy Approach

Using a similar strategy of the algorithm mentioned in the previous section, however rather than trying to remove all combinations of L patterns, instead we choose the L patterns greedily.

The greedy hypothesis: choose the L patterns which has the highest number of already removed strings.

Using such a greedy hypothesis can lead to answers that are very far from the real one, however there is an assumption which is not yet proved, that is: there is a certain permutation of the rules that will allow the greedy hypothesis to find the optimal answer. Two different approaches utilizing this assumption were tried:

1. Based on this assumption, an algorithm which tries all permutations of the rules and try to satisfy them using the greedy hypothesis would be an exact algorithm.
2. We could try to find that optimal order of the rules based on other factors. For example, trying to sort the rules in descending order of the number of patterns that needs to be removed Q(r). Or sorting them in descending order of the ratio "number of patterns that needs to be removed : the total number of patterns", Q(r)/f*(r). However, both strategies were disproved and they lead to wrong answers in certain test cases.

References

[1] Algorithms for Approximating Rule Productivity. Haythem O. Ismail and Mohamed K. Gabr.