Seminar Report

Multiple Constraint Acquisition

A report submitted in partial fulfillment for the Ph.D qualifier seminar

By

Rajiv Kumar V (Roll No. 164050015)

Under the guidance of

Prof. Siva Kumar

Department of Computer Science and Engineering, IIT Bombay

November 2017

Abstract

Constraint satisfaction problems(CSP) are mathematical problems whose solutions are those states in the state space that satisfy constraints from the problem definition. CSPs have deep roots in many of the known real world problems including timetabling, scheduling, planning, and resource allocation. These problems are inherently combinatorial in nature and have constraints to satisfy, where each constraint specifies a restriction on some set of variables in the problem. A problem modeled as CSP can be effectively solved using CSP solvers which applies techniques like Constraint propagation, Backtracking, local search.etc. However, modeling a combinatorial problem in the constraint formalism requires significant expertise in constraint programming. CSP solvers are not widely used in practice due to the inherent difficulty of modeling a problem as CSP. Consequently, an expert has to identify the constraints and their domains to model a problem as CSP.

Constraint acquisition is a technique by which a problem can be modeled as CSP by providing examples and answering queries of the learner. It is the task of automatically acquiring a constraint network formulation of a problem from a subset of its solutions and non-solutions. In practice, the user may have already solved several instances of the problem without the help of a solver and knows how to classify an example as a solution or a non-solution to it. Based on these considerations, the overall aim of constraint acquisition is to induce from examples a general constraint network that adequately represents the target problem. This approach allows to use the learned network with different initial domains in order to solve further tasks supplied by the user. This seminar is a review of the literature, algorithms, and developments in Constraint acquisition and Multiple Constraint Acquisition with the focus on finding the challenges and future scope in the area.

Acknowledgement

The satisfaction that accompanies the successful completion of this seminar would be incomplete without the mention of the people who made it possible. I consider myself privileged to express gratitude and respect towards all those who guided and helped me throughout till the completion of this seminar.

I take this occasion to thank the Almighty Lord who has instilled in me the courage and strength to take up and successfully complete this endeavor. I am sincerely thankful to my Seminar guide **Prof. Siva Kumar**, Department of Computer Science and Engineering for enlightening me with the knowledge from his past experiences giving constant support and guidance which was of a great help to complete this seminar successfully. Last but not least are the names of my family and friends who have stood by my side all these times. I would also like to thank them all for their help and support.

List of Figures

List of Tables

Contents

Chapter 1

1 Introduction

1.1 Overview

Constraint Satisfaction problems(CSPs) are mathematical problems that exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. They are used in industries and other scientific areas where there is a need for planning, scheduling, and allocating resources in order to improve the quality of service, reduce costs, or optimize resource consumption. Examples range from power companies generating and distributing electricity to hospitals planning their surgeries and public transportation companies scheduling their time-tables. However, CSPs are not limited to scientific areas; they are applicable in many problem domains.

Some well known domains and problems which are known to be CSP are: Bioinformatics, Network design, Scheduling and related problems, Stochastic Constraint Programming, Time Tabling, Bin packing problems, Clustering, Design and Configuration, Distributed CSP/COP, Games and puzzles, and other combinatorial problems. In other words, Boolean Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and Answer Set Programming (ASP) can be thought of and modeled as Constraint Satisfaction Problems(CSP).

Constraint Programming(CP) is the programming paradigm used to solve Constraint Satisfaction Problems. The solution involves modeling the problem as a constraint network and then solving the constraint network using CSP solvers. A CSP then reduces to the problem of finding a solution for a given constraint network. A constraint network can loosely be defined as a graph with nodes and edges; where the nodes represent the scope of the constraints and the edges connecting the nodes represent the constraints between the variables. A solution of the constraint network is an assignment of variables to domain values that satisfies every constraint in the network.

In constraint programming, a distinction is made between model and data. The model contains generic constraints while the data, on the other hand specifies the features of a problem instance. The cost of learning the model can be amortized over the lifetime of the model which motivates the acquisition of Constraint models. In all these applications of combinatorial problems it is very hard to acquire the constraints and criteria. i.e. the model needed to specify the problem. Even if the model has been obtained at some point, it needs to be to changed over time to reflect changes in the environment. In short, a generalized model is unavailable for some CSP in some domain to be used with different initial domains.

1.2 Problem Statement

Constraint acquisition is the technique by which a constraint network for a problem is acquired with the help of examples given to the learner. Assuming the constraints of the problem can be constructed with the set of relations provided to the learner, the aim of constraint acquisition is to induce from a set of examples, a general constraint network that can represent the target problem, which is the problem that the user has in mind. This approach then allows the constraint network learned from the acquisition procedure to be used with different initial domains in order to solve further tasks supplied by the user.

There are two acquisition approaches: Active constraint acquisition or Passive constraint acquisition, depending upon the learning method. In Passive constraint acquisition the learner cannot ask queries to the user. The learner has to learn from the positive and negative examples provided by the user. Here the learner has to inform the user about the state of it's version space and whether the version space has converged. On the other hand, in Active constraint acquisition, the learner can choose an example and ask whether it is a solution or not of the target problem. The number of examples required to acquire a constraint network is significantly reduced if queries are selected carefully. However, when acquiring constraint networks, computing good queries is a hard problem.

In the classic query generation strategy, regardless of the classification of the query, the size of the version space is reduced by half which ensures the convergence to the target constraint network in a logarithmic number of queries. In the same classic setting, a query can be generated in time polynomial in the size of the version space. However, when acquiring constraint networks, query generation becomes NP-hard. This leads to variants of the queries that the learner can ask, from partial queries to recommendation queries, and where the user can provide background knowledge to the learner to aid in the constraint acquisition process, which in turn makes the constraint acquisition with quick convergence or acquisition with fewer queries an interesting problem.

1.3 Outline

The seminar report has been organized as follows:
Chapter 1 gives an Introduction to Constraint acquisition;
Chapter 2 deals with the Terminology and Related work;
Chapter 3 deals with Multiple Constraint Acquisition;
Chapter 4 deals with the Conclusion and Future Scope;
This is followed by the references of the publications.

Chapter 2

2 Related Work

2.1 Introduction

Constraint Programming (CP) provides a powerful paradigm for solving combinatorial problems. The success of constraint programming comes from its efficiency to solve combinatorial problems represented as constraint networks. However, the specification of constraint networks still remains limited to specialists in the field since constraint networks are much more complex to acquire than simple conjunctive concepts represented in propositional logic. While in conjunctive concepts the atomic variables are pairwise independent, in constraint satisfaction there are dependencies amongst them. Each constraint of the constraint system is expressed as a clause and the constraint system can be expressed in conjunctive normal form (CNF).

Let's see the example of Sudoku to better understand the constraint acquisition problem.

SUDOKU: Sudoku [4] is a logic puzzle with 9 X 9 grid. The grid must be filled with numbers from 1 to 9 in such a way that all the rows, columns and the 9 non overlapping 3 X 3 squares contain the numbers 1 to 9. The target network has 81 variables with domains of size 9, and 810 binary $\neq$ constraints on rows, columns and squares. Assuming the user knows positive and negative examples of Sudoku and constraint language $\lambda = \{=, \neq\}$ made available to the learner; the bias of 6480 binary constraints needs to be acquired for the constraint system by answering the queries of the learner or by providing enough examples until the learner has converged on the target constraint network. This technique of acquiring the constraint network is called Constraint Acquisition.

2.2 Terminology

Constraint Language / Constraint Library: A constraint language is a set = $\{r_1, ..., r_t\}$ of t relations over some subset of Z. The learner owns a language of relations from which it can build constraints on specified sets of variables. The relations are operators of some fixed arity that can be used as they are or even combined to formulate constraints between the variables. Mostly all derivable relations are not defined in the language. Those constraints which depend on the derivable relations are defined using derivable constraints. Usually a language is also assumed to be known to the learner, or else it is input to the learner.
E.g. In the Sudoku example the constraint language is $\lambda = \{=, \neq\}$

Constraint bias: Given a prefixed vocabulary (X, D), the bias for the learning task is a set $B \subseteq B_\Gamma$. Set B of constraints built from the constraint language on the vocabulary. These are relations from the language whose scope is defined on the finite set of variables.

E.g. If $\lambda = \{\leq, \neq, \geq\}$ it is possible to derive new constraint relations like $\{<, >\}$ by combining the existing relations. The whole set of relations that can be built from the constraint library forms the constraint bias.

Vocabulary: For the constraint acquisition process, the user and learner needs to share the same vocabulary to communicate. A vocabulary is a pair (X, D) such that X is a finite set $\{x_1, ..., x_n\}$ of variables, and D is a finite subset of Z called the domain.
E.g. In the sudoku example, X stands for variables $\{x_1, ..., x_{81}\}$ representing each number from the squares while D stands for the domain of each variable; which is $\{1...9\}$

Constraints network: It is a set C of constraints on the vocabulary (X, D), X being the constraint scope and D being the domain. A constraint network over a given vocabulary (X, D) is a finite set C of constraints. Each constraint c in C is a pair $\{var(c), rel(c)\}$, where $var(c)$ is a sequence of variables of X, called the constraint scope of c, and $rel(c)$ is a relation over $D|var(c)|$, called the constraint relation of c. For each constraint c, the tuples of $rel(c)$ indicate the allowed combinations of simultaneous value assignments for the variables in $var(c)$. The arity of a constraint c is given by the size $|var(c)|$ of its scope.

Example: An example e is an assignment on a set of variables from their domain range. It can be classified as either positive or negative depending on whether the assignment satisfies the constraints of the target constraint network. An example can be partial if only a proper subset of the variables are given an assignment. A complete example on the other hand is an assignment of all variables from their domain.
E.g. An example of Sudoku puzzle has all 9 X 9 squares filled in with numbers without any constraints failing.

Training set: It is a set of positive or negative examples. It is input to the learner for passive acquisition techniques.
E.g. A set of completely solved Sudoku with different initial configurations serve as positive examples ; whereas puzzle configurations where at least one or more rows or columns(similarly 3 X 3 subsquares) having duplicates can be deemed as negative examples in the training set.

Solution: An assignment e is a solution of C iff for all constraints c in C, the constraint c does not reject the example e. In other words, a set of constraints C accepts an assignment e iff there does not exist any constraint from constraint set rejecting e.

Assignment: An assignment can be either partial or complete depending upon the subset of the variables involved. A complete assignment can be either a solution or non-solution.

Sol(C): The set of solutions of C

Concept: Given a vocabulary, a concept is a Boolean function defined over the domain of the variables, assigning to each assignment a value in the set $\{0, 1\}$. It is a map that assigns to each example a value in $\{0, 1\}$. Loosely said, a concept classifies an example as positive or negative.

Target concept: The concept that returns a 1 for example e if and only if e is a solution of the problem the user has in mind. If the example e is not a solution then 0 is returned.

Consistent concept: If the concept is consistent with every example in that training set.

Query: A query is essentially a question about the constraint network that the user must answer based on the target network.

Membership query: A query where a user is required to classify an example as positive or negative.

Ask(e): A classification query asked to the user with variable(e) subset of X. The answer to Ask(e) is yes if and only if the target network accepts e.

Partial queries: Partial queries are assignments of only a subset of the variables of the problem.

Recommendation query / AskRec(c): Asks a user whether or not a predicted constraint belongs to the target constraint network.

Equivalence query: The user is requested to decide whether a proposed concept is equivalent to the target. In case of a negative answer the user should provide with an example to show the discrepancy between the 2 models. In a setting of constraint acquisition, asking a user equivalence query is unreasonable as the user may not be able to articulate the constraints of the target network directly. Given any bias and any target concept representable by a bias, the concept is learnable by equivalence queries.

Generalization query: The user provides the variable types and based on these types, generalization queries ask the user whether or not a learned constraint can be generalized to other scopes of variables of the same type as those on the learned constraint. This is based on an aggregation of variables into types.

Admissible network: A constraint network C is said to be admissible for a bias B if for each constraint c_{ij} in C there exists a set of constraints $\{b_{ij}^1, ..., b_{ij}^k\}$ in B such that $c_{ij} = b_{ij}^1 U ... b_{ij}^k$

Identification/Learnability: Given a bias B, a training set E on a vocabulary (X, D), and a type $TQ \in \{membership, equivalence\}$ of queries, the identification problem for a target concept f representable by Γ is to find a sequence $\{q_1, ..., q_m\}$ of queries of type TQ leading to the detection of a constraint network C representing f. If the length m of the sequence is polynomial in the number $|B|$ of constraints in the bias, we say that f is learnable by queries of type TQ.

Consistency problem: Given a bias B and a training set E, the consistency problem is to determine whether $C_B(E) = \Phi$. If the answer is "yes", then a representation C of some concept in $C_B(E)$ must be returned.

Convergence problem: Given a bias B and a training set E, the convergence problem is to

determine whether $C_B(E)$ has converged. If the answer is "yes", then a representation C of the target concept in $C_B(E)$ must be returned.

Clausal representation: Given a bias B and a training set E, the clausal representation of $C_B(E)$ is the dual Horn formula.

Query generation problem: Given a bias B, a training set E, an integer t, (and optionally a non-unary positive clause α in T), the query generation problem is to find a query q such that $K_{[T]}(q)$ does not cover any clause in T (resp. $K_{[T]}(q)$ $constraints(\alpha)$) and $|K_{[T]}(q)| = t$.
Deciding whether such a query q exists is called the query existence problem.

Intractability of query generation: The query existence problem is *NP-complete* for any value of t, and thus the generation problem is *NP-hard* for any value of t.

Hitting sets: Given a collection of sets from some finite domain D, a hitting set is a set of elements from D that *hits* every set by having at least one element in common with it.

Minimal / Irreducible hitting sets: They are hitting sets from which no element can be removed without losing the property of being a hitting set.

Minimal Unsatisfiable Subset (MUS): Given an unsatisfiable system of constraints C, an *MUS* of C is a subset of those constraints that is unsatisfiable and minimal, in the sense that removing any one of its elements makes the remaining set of constraints satisfiable.

Minimal Correction Subset (MCS): Given an unsatisfiable system of constraints C, *MCS* is a subset of the constraints of an infeasible constraint system whose removal from that system results in a satisfiable set of constraints and that is minimal in the same sense that any proper subset does not have that defining property.

Community: A network or a graph is said to have community structure, if the nodes of the network can be easily grouped into (potentially overlapping) sets of nodes such that the groups have more internal edges than outgoing edges. Such a group is called a Community.

2.3 Related Work

Modeling constraint satisfaction problem requires knowledge from examples, which is done by iteratively improving upon the learned constraint network by adding new constraints learned from each new example.

The next challenge to constraint acquisition is to reduce the dialog length between the user and the learner, that is, to reduce the number of asked queries to get the target model. Several papers have already proposed to use the structure of the constraint graph to decrease the number of examples needed to learn the target constraint network.

2.3.1 Constraint acquisition

In the paper *Constraint acquisition*, [1] Bessiere et. al. states the theoretical questions regarding constraint acquisition. They propose *CONACQ*, a system that uses a concise representation of the learner's version space into a clausal formula. They also discuss the computational properties of the strategies and their practical effectiveness.

Analysis:

- The consistency problem can be solved in polynomial time.
- The convergence problem is *coNP-complete*
- Given any bias B and any target concept f representable by B, f is learnable by equivalence queries.
- There exist biases B with fixed arity constraints, training sets E, and target concepts f representable by B that are not learnable by membership queries, even if E contains a positive example.

Passive CONACQ.1 Algorithm

In Passive constraint acquisition, the learner must determine the state of its version space from the pool of examples while having no control over the training set information given by the user. It takes as input a bias B along with the training set E, and returns as output a clausal theory T that encodes the version space $C_B(E)$.

The algorithm *(fig 1, page 14)* starts from the empty theory and iteratively expands it by encoding each example in the training set. If e is negative, all concepts that accept $x(e)$ are discarded from the version space. This is done by expanding the theory with the clause consisting of all literals $a(c)$

in $K(x(e))$. Dually, if e is positive, all concepts that reject $x(e)$ is discarded from the version space. This is done by expanding the theory with a unit clause $\not a(c)$ for each constraint c in $K(x(e))$. If the theory is no longer satisfiable after encoding the example, a *collapse* message is returned.

After processing all examples in the training set, a convergence procedure is called to determine whether $C_B(E)$ is reduced to a singleton set, or not. Finally *CONACQ.1* returns the resulting theory encoding $C_B(E)$ together with the flag for convergence. As the convergence problem is *coNP-complete*, a naive strategy for implementing the convergence procedure is to start from the interpretation I, encoding the maximally specific network, and to explore the other models of T in order to find an interpretation I for which the constraint networks ϕ and are not equivalent.

Algorithm 1: The CONACQ.1 algorithm.

Input: a bias **B** and a training set E
Output: a clausal theory T encoding $\mathbf{C_B}(E)$, a Boolean value v saying if convergence is reached

1 $T \leftarrow \varnothing$
2 **foreach** *example* $e \in E$ **do**
3 $\quad \kappa(\boldsymbol{x}(e)) \leftarrow \{c \in \mathbf{B} \mid \boldsymbol{x}(e)$ violates $c\}$
4 $\quad$ **if** $y(e) = 0$ **then** $T \leftarrow T \wedge \left(\bigvee_{c \in \kappa(\boldsymbol{x}(e))} a(c)\right)$
5 $\quad$ **if** $y(e) = 1$ **then** $T \leftarrow T \wedge \bigwedge_{c \in \kappa(\boldsymbol{x}(e))} \neg a(c)$
6 $\quad$ **if** T *is unsatisfiable* **then return** *"collapse"*
7 $v \leftarrow$ CONVERGENCE(T)
8 **return** (T, v)

Figure 1: Conacq.1 Algorithm [1]

Analysis:

- Let (X, D) be a prefixed vocabulary, B bias, and E a training set. Let T be the clausal representation of $C_B(E)$. Then, $I \epsilon\ models(T)$ if and only iff $\phi(I) \epsilon\ C_B(E)$
- The consistency problem can be solved in $O(|E|.|B|)$ time.
- The update operation takes $O(|B|)$ time.
- The intersection operation takes $O((|E_1| + |E_2|).|B|)$ time.
- The subset and equality tests take $O(|E1|.|E2|.|B|^2)$ time.
- The membership test takes $O(|E|.|B|)$ time.
- The prediction test takes $O(|E|.|B|)$ time.

- If $C_B(E) = \phi$, then the maximally specific concept is unique and a representation of it can be computed in $O(|E|.|B|)$ time.
- If $C_B(E) = \phi$ and T is a monomial, then the maximally general concept is unique and a representation of it can be computed in $O(|E|.|B|)$ time.

Active CONACQ.2 algorithm

CONACQ.2 is an active learning version of the *CONACQ* algorithm where the learner can ask membership queries. i.e. select an assignment x and ask the user to classify the assignment with a positive or negative label. If x is a solution of the target problem the user answers *yes*; and *no* otherwise. *CONACQ.2 (fig 2, page 16)* takes as input a constraint bias B, background knowledge K for B, and a query generation strategy used by function *QueryGeneration (fig 3, page 16)* for the generation of queries and returns a clausal theory T that encodes a constraint network representing the target concept.

Each time an interdependency among constraints that is not captured by the background knowledge K is discovered, we encode the interdependency as a logical no good that is stored in a set N to avoid repeatedly discovering it. $CONACQ.2$ starts from an empty theory T and iteratively expands it by an example (x, y) formed by the query $q = x$, and the user response y supplied. The query generation process is implemented by *QueryGeneration*, which takes as input the bias B, the current clausal theory T, the given background knowledge K, the current set of no goods N, and the given strategy *Strategy*.

If there exists irredundant queries, *QueryGeneration* returns an irredundant query q following *Strategy* as much as possible, i.e. with $|K_T(q)|$ as close as possible to the specified t. If there is no irredundant query, this means that convergence is reached and the theory encoding the target network is returned. Otherwise, the query q is supplied to the user for membership, who answers by *yes* or *no*. If q is classified as negative by the user, all concepts that accept q must be discarded from the version space. This is done by expanding the theory with the clause consisting of all literals $a(c)$ with c in $K(q)$. Dually, if q is classified as positive, all concepts that reject q must be discarded from the version space. This is done by expanding the theory with a unit clause $\not{a}(c)$ for each constraint c in $K(q)$.

Algorithm 4: The CONACQ.2 algorithm.

Input: a bias **B**, background knowledge K, a strategy *Strategy*
Output: a clausal theory T encoding the target network

1 $T \leftarrow \varnothing$; *converged* $\leftarrow$ *false*; $N \leftarrow \varnothing$
2 **while** $\neg$*converged* **do**
3 $q \leftarrow$ QUERYGENERATION(**B**, T, K, N, *Strategy*)
4 **if** $q = nil$ **then** *converged* $\leftarrow$ *true*
5 **else**
6 **if** ASK(q) $=$ *no* **then** $T \leftarrow T \wedge \left(\bigvee_{c \in \kappa(q)} a(c)\right)$
7 **else** $T \leftarrow T \wedge \bigwedge_{c \in \kappa(q)} \neg a(c)$
8 **return** T

Figure 2: Conacq.2 Algorithm [1]

Algorithm 5: QUERYGENERATION.

Input: the bias **B**, the clausal theory T, background knowledge K, a nogood set N, and a strategy *Strategy*
Output: a query q

$q \leftarrow nil$; $\alpha \leftarrow \varnothing$
while (T *is not a monomial*) *and* ($q = nil$) **do**
  if ($\alpha = \varnothing$) *and* (T *contains non-unary unmarked clauses*) **then**
    read a non-unary unmarked clause α in T
    $\epsilon \leftarrow 0$; Assign t depending on *Strategy*
  splittable $\leftarrow (\alpha \neq \varnothing) \wedge ((t + \epsilon < |\alpha|) \vee (t - \epsilon > 0))$
  $F \leftarrow$BUILDFORMULA(*splittable*, T, α, t, ϵ)
  if *models*($F \wedge K \wedge N$) $= \varnothing$ **then**
    if *splittable* **then** $\epsilon \leftarrow \epsilon + 1$
    else $T \leftarrow T \cup \{a(c) \mid a(c) \in \alpha\} \setminus \{\alpha\}$; $\alpha \leftarrow \varnothing$
  else
    select $I \in$ *models*($F \wedge K \wedge N$)
    if *Sol*($\varphi(I)$) $= \varnothing$ **then**
      foreach *CS* $\in$ *ConflictSets*($\varphi(I)$) **do**
        $N \leftarrow N \cup \{\bigvee_{c \in CS} \neg a(c)\}$
    else
      select any q in *sol*($\varphi(I)$)
      if ($\neg$*splittable*) *and* ($\alpha \neq \varnothing$) **then** mark α
if $q = nil$ **then** $q \leftarrow$ IRREDUNDANTQUERY(T)
return q

Figure 3: Query Generation Algorithm [1]

2.3.2 Quick Acquisition

Constraint Acquisition via Partial Queries [2] introduces an algorithm for Quick Acquisition called *QUACQ*. *QUACQ* is an active learner which, in addition to membership queries, is able to ask the user to classify partial queries. The algorithm asks partial queries to the user until it has converged on a constraint network C_L equivalent to the target network C_T, or collapses. As opposed to membership queries, partial queries do not involve all variables; they are assignments of only part of the variables of the problem. Even if the user is not able to articulate the constraints of his problem, the user is assumed to be able to decide if partial assignments of variables satisfies or violates some requirements.

QUACQ takes as input a bias B on a vocabulary (X, D). QUACQ then iteratively generates partial queries and asks the user to classify them. When the answer of the user is *yes*, *QUACQ* as usual reduces the search space by removing constraints that reject the positive example. For a negative answer, *QUACQ* focuses on a single constraint from the negative example. This is performed in a number of queries logarithmic in the size of the example which allows *QUACQ* to converge on the target network in a polynomial number of queries. Despite this good theoretical bound, *QUACQ* may require a lot of queries to learn the target constraint network, especially when the problem is highly structured and involves a large number of constraints. E.g. Sudoku.

The advantage of QUACQ over *CONACQ* comes in the case of a negative example where *CONACQ* produces a non-unary positive clause on the candidate constraints for rejection, while *QUACQ* uses partial queries to return exactly a constraint of the target network in logarithmic time. Another advantage of *QUACQ* is that it provides less of a burden on the user. First, it often converges quicker than other methods. Second, partial queries will be easier to answer than complete queries. Third, the user does not need to give positive examples, which might be useful if the problem has not yet been solved, and there are no examples of past solutions.

Algorithm

Bessiere et.al [2] explains the algorithm *(fig 4, page 18)* as given below. *QUACQ* initializes the network C_L it will learn to the empty set. If C_L is unsatisfiable, the space of possible networks collapses because there does not exist any subset of the given bias B that is able to correctly classify the examples already asked of the user. *QUACQ* computes a complete assignment e satisfying C_L but violating at least one constraint from B. If such an example does not exist, then all constraints in B are implied by C_L, which implies convergence. If the constraint network has not been converged, the example e is proposed to the user, who will answer by *yes* or *no*. If the answer is *yes*, the set $B(e)$ of all constraints in B that reject e can be removed from B. If the answer is *no*, it implies that e violates at least one constraint of the target network C_T. The function *FindScope* (fig 5, page 19) is then called to discover the scope of one of these violated constraints.

FindC (fig 6, page 20) will identify which one with the given scope is violated by the negative e. If no constraint is returned, this implies a collapse as a constraint rejecting one of the negative

Algorithm 1: QUACQ: Acquiring a constraint network C_T with partial queries

```
1  C_L ← ∅;
2  while true do
3      if sol(C_L) = ∅ then return "collapse";
4      choose e in D^X accepted by C_L and rejected by B;
5      if e = nil then return "convergence on C_L";
6      if ASK(e) = yes then B ← B \ κ_B(e) ;
7      else
8          c ← FindC(e, FindScope(e, ∅, X, false));
9          if c = nil then return "collapse";
10         else C_L ← C_L ∪ {c};
```

Figure 4: Quacq Algorithm [2]

examples could not be found in B. Otherwise, the constraint returned by *FindC* is added to the learned network C_L.

FindScope is a recursive function that takes as parameters an example e, two sets R and Y of variables, and a Boolean *ask_ query*. An invariant of *FindScope* is that e violates at least one constraint whose scope is a subset of R U Y. When *FindScope* is called with *ask_ query* = *false*, we already know whether R contains the scope of a constraint that rejects e. If *ask_ query* = *true* the user is asked whether $e[R]$ is positive or not. If *yes*, all the constraints that reject $e[R]$ from the bias are removed, otherwise the empty set is returned. The line 4 is reached only in case $e[R]$ does not violate any constraint. We know that $e[R\ U\ Y]$ violates a constraint. Hence, as Y is a singleton, the variable it contains necessarily belongs to the scope of a constraint that violates $e[R\ U\ Y]$. The function returns Y. If none of the return conditions are satisfied, the set Y is split in two balanced parts and a technique similar to *QUICKXPLAIN* [5] is applied to elucidate the variables of a constraint violating $e[R\ U\ Y]$ in a logarithmic number of steps.

The function *FindC* takes as parameter e and Y, e being the negative example that led *FindScope* to find that there is a constraint from the target network C_T over the scope Y. *FindC* first removes from B all constraints with scope Y that are implied by C_L because there is no need to learn them. The set Δ is initialized to all candidate constraints violated by e. If Δ no longer contains constraints with scope Y, ϕ is returned, which will provoke a collapse in *QUACQ*.

An example e_0 is chosen in such a way that contains both constraints rejecting e_0 and constraints satisfying e_0. If no such example exists, this means that all constraints in Δ are equivalent w.r.t $C_L[Y]$. Any of them is returned except if Δ is empty. If a suitable example is found, it is proposed

Algorithm 2: Function `FindScope`: returns the scope of a constraint in C_T

function `FindScope`(**in** e, R, Y, ask_query)**: scope**;
begin
1 | **if** ask_query **then**
2 | | **if** $ASK(e[R]) = yes$ **then** $B \leftarrow B \setminus \kappa_B(e[R])$;
3 | | **else return** $\varnothing$;
4 | **if** $|Y| = 1$ **then return** Y;
5 | split Y into $< Y_1, Y_2 >$ such that $|Y_1| = \lceil |Y|/2 \rceil$;
6 | $S_1 \leftarrow$ `FindScope`$(e, R \cup Y_1, Y_2,$ **true**$)$;
7 | $S_2 \leftarrow$ `FindScope`$(e, R \cup S_1, Y_1, (S_1 \neq \varnothing))$;
8 | **return** $S_1 \cup S_2$;
end

Figure 5: Findscope Algorithm [2]

to the user for classification. If classified positive, all constraints rejecting it are removed from B and Δ, otherwise all constraints accepting that example are removed from Δ. To ensure quick converge, we want a query answered *yes* to prune B as much as possible. This is best achieved when the query generated in line 4 of *QUACQ* is an assignment violating a large number of constraints in B.

Analysis

- Given a bias B built from a language Γ, a target network C_T, a scope Y, *FindC* uses $O(|\Gamma|)$ queries to return a constraint c_Y from C_T if it exists.
- Given a bias B, a target network C_T, an example $e \in D^X \setminus sol(C_T)$, *FindScope* uses $O(|S|.log|X|)$ queries to return the scope S of one of the constraints of C_T violated by e.
- Given a bias B built from a language Γ of bounded arity constraints, and a target network C_T, *QUACQ* uses $O(|C_T|.(log|X| + |\Gamma|))$ queries to find the target network or to collapse and $O(|B|)$ queries to prove convergence.
- *QUACQ* learns Boolean networks on the language $\{=, \neq\}$ in an asymptotically optimal number of queries.
- It can learn constraint networks with unbounded domains on the language $\{=\}$ in an asymptotically optimal number of queries.
- It can learn Boolean constraint networks on the language $\{<\}$ in O(n) queries.

Algorithm 3: Function `FindC`: returns a constraint of C_T with scope Y

```
   function FindC(in e, Y): constraint;
   begin
1      B ← B \ {c_Y | C_L ⊨ c_Y};
2      Δ ← {c_Y ∈ B[Y] | e ⊭ c_Y};
3      if Δ = ∅ then return ∅;
4      while true do
5          choose e' in sol(C_L[Y]) such that
           ∃c, c' ∈ Δ, e' ⊨ c and e' ⊭ c';
6          if e' = nil then
7              if Δ = ∅ then return nil;
8              else pick c in Δ; return c;
9          if ASK(e') = yes then
10             B ← B \ κ_B(e'); Δ ← Δ \ κ_B(e');
11         else Δ ← Δ ∩ κ_B(e');
   end
```

Figure 6: FindC Algorithm [2]

- It does not learn Boolean networks on the language $\{<\}$ with a minimal number of queries.

2.3.3 Ask & Solve

Bessiere et.al proposes *ASK&SOLVE* [6], an elicitation based algorithm that solves a constraint problem without the need of a constraint network representing it. The elicitation here consists in asking the user to classify partial assignments as positive or negative. It tries to find the best tradeoff between learning and solving to converge on a solution with as few queries to the user as possible.

ASK & SOLVE learns constraints by asking queries during search. During the initial steps of learning, if queries with complete assignment are tried, it leads to expensive constraint elicitation steps on long negative assignments when the problem is critically constrained. If exploring first plenty of short assignments is promoted it leads to a fast and cheap learning of constraints which but it can be a waste of time if the problem is easy to solve.

The idea in *ASK&SOLVE* is to try to extend step by step, a scope on which at least one assignment is accepted by the target network C_T. Each time there is a chance that the assignment generated by *ASK&SOLVE* violates one of the constraints in the bias B, *ASK&SOLVE* asks the query to the user. If the answer to the query is negative, *ASK&SOLVE* immediately launches a procedure that learns a culprit constraint to avoid generating again an assignment rejected for the same reason.

Based on the *ASK&SOLVE* algorithm several strategies [6] are proposed which includes restart policies/variable ordering to speed-up even more convergence on a solution. Restart policies have been a long-held goal in AI and they are one of the approaches developed to boost combinatorial search. Three restart policies are proposed that triggers a restart, when the number of negative answers is greater than or equal to the cutoff value returned by restart().

Fixed cutoff: The first policy uses a fixed cutoff, noted FC. This is done by implementing the function *restart()* so that it always returns $|X|$.

Geometric: The second policy is the Geometric strategy. The cutoff value returned by *restart()* grows geometrically by a factor of 1.5. We use an initial cutoff size of $|X|$.

Luby: The third policy is more elaborated. In this case the restart() function implements the universal Luby-restarts policy. The Luby-restarts policy is given by the sequence (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1...) that is multiplied by the factor |X|. Such a policy has been used with great success in SAT solvers. The goal is, on the one hand to cut the search early enough to escape from bad subtrees and to produce short nogoods, and on the other hand to let the solver the chance to go to a solution.
In a restart policy, there is some particular order in which to select variables in the search following immediately the restart.
Random (RAND): At each restart event, the variables are reordered randomly.

Lexicographic (LEX): Use the basic lexicographic order. When a restart occurs, the same variable order is followed: $x_1, x_2, x_3 \rightarrow x_1, x_2, x_3, x_4...$

Reverse-lex (R-LEX): The algorithm is initialized with a *LEX* order on the variables. Once a restart event occurs, the variables in the scope *scp* are reversed: Start by selecting the last variable that was added to scp and continue until the first. If we reach the first (that is, we exhausted all variables from that previous scope), we select the remaining variables in the same order as they were ordered before the restart. And so on.

Continuous-lex (C-LEX): A variant of the lexicographic order is used. At the beginning we have a *LEX* order on the variables. Once a restart event occurs, we select the last variable in *scp* and we continue on the same *LEX* order. (This LEX order has to be circular: x_n is followed by x_1.) This heuristic allows the search process to explore the variables in a balanced way. $x_1, x_2, x_3 \rightarrow x_3, x_4 \rightarrow x_4, x_5, x_6 ...$

2.3.4 Recommendation Queries

Link prediction in dynamic graphs is an important research field in data mining. All these link prediction techniques compute and assign a score to pairs of nodes (x, y) based on the input graph and then produce a ranked list in a decreasing order of scores. They can be viewed as computing a measure of proximity or similarity between nodes x and y with respect to the network topology. Most of these techniques are based either on node neighborhood or on path ensemble. The paper *Constraint Acquisition with Recommendation Queries* [7] selects one link prediction technique representative of node-neighborhood-based techniques *(Adamic/Adar)*, and one representative of path-ensemble-based techniques *(Leicht-Holme-Newman Index)* to improve upon the basic version of *QUACQ*.

When a constraint network has some structure, variables of the same given type are often involved in constraints with the same relation. Hence, it is expected that when variable types are not known in advance, predicting type similarity or type proximity of variables can be done by prediction link techniques. By using techniques borrowed from link prediction in dynamic graphs, constraints that are more likely to belong to the target constraint network are inferred, and are validated by asking recommendation queries to the user.

A recommendation query asks the user whether or not a predicted constraint belongs to the target constraint network. To deal with recommendation queries, a constraint recommender algorithm called *PREDICT & ASK* is proposed, which when plugged into the *QUACQ* constraint acquisition system leads to the *P-QUACQ* algorithm. The idea behind *P-QUACQ* algorithm is to predict missing constraints in the partial network learned so far, and then to recommend the predicted constraints to the user through recommendation queries. The algorithm *PREDICT & ASK* takes as argument the set of constraints C learned so far, a relation r, and the predictor score that corresponds to the strategy used to assign a cost to a candidate constraint for recommendation.

Algorithm

PREDICT & ASK starts by initializing L to the empty set. The set L will contain the output of *PREDICT & ASK*. i.e. all constraints learned by prediction plus recommendation query. The constraint graph $G = (Y, E)$ is restricted to the relation r. The counter counts the number of consecutive times recommendation queries have been classified negative by the user. It is initialized to zero. All constraints that are candidate for recommendation are put in.

In *PREDICT & ASK*, for each iteration, a constraint is picked such that its score is maximum. A constraint with a high score implies that this constraint belongs to the target constraint network. Hence, *PREDICT & ASK* asks a recommendation query on $((x; y); r)$. If the user says *yes*, $((x; y); r)$ is a constraint of the target network. Hence, we put $((x; y); r)$ in L. The edge $(x; y)$ to E has to be taken into account in the next iteration when computing the score. Num is reinitialized to zero. If the user says *no*, remove the constraint $((x; y); r)$ from B and Num is incremented. The loop ends when Δ is empty or when Num reaches the given threshold, and L is returned.

P-QUACQ initializes the constraint network C_L to the empty set. When C_L is unsatisfiable, the space of possible networks collapses because there does not exist any subset of the given basis B that is able to correctly classify the examples already asked to the user. *P-QUACQ* computes a complete assignment e satisfying C_L and violating at least one constraint from B. If such an example does not exist, then all constraints in B are implied by C_L, and the algorithm has converged. Otherwise, the example e is proposed to the user, who will answer by *yes* or *no*.

If the answer is *yes*, the set $B(e)$ of all constraints in B that reject e can be removed from B. If the answer is *no*, we are sure that e violates at least one constraint of the target network C_T. We then call the function *FindScope* to discover the scope of one of these violated constraints. Here, *FindScope* acts in a dichotomous manner and asks a number of queries logarithmic in the size of the example. *FindC* selects which constraint with the given scope is violated by e. If no constraint is returned, this is a condition for collapsing as we could not find in B a constraint rejecting one of the negative examples. Otherwise, we know that the constraint c returned by *FindC* belongs to the target network C_T , we then add it to the learned network C_L. Afterwards, we call *PREDICT & ASK* to mine the learned constraint network C_L in order to predict and recommend missing constraints that may belong to the target network. *P-QUACQ* updates C_L by adding all learned constraints.

2.3.5 Other related work

GENACQ [8] is an algorithm based on generalization queries that require the aggregation of variables into types. The idea is to infer potential types by analyzing the structure of the current constraint network and to use the extracted types to ask generalization queries, based on an aggregation of variables into types. The user provides the variable types and based on these types, generalization queries ask the user whether or not a learned constraint can be generalized to other scopes of variables of the same type as those on the learned constraint.

The aggregation of variables into types may not always be a straightforward task for the user especially when the problem under consideration has a hidden structure. The idea is to analyze the structure of the partial constraint network learned so far in order to detect potential types and to build generalization queries. The variables of the same type are often tightly connected with similar constraints whereas the variables of different types are connected in a weaker way. This point can be illustrated by considering the Lewis Carroll's Zebra problem.

An important characteristic that commonly occurs in networks with tightly connected sub-graphs such as social networks and biochemical networks is community structure. Given the similarity between the structure of a type and that of a community, it is possible to detect potential types by finding communities in the constraint network during the constraint acquisition process. Several methods for community finding have been proposed in the literature. The paper considers three different techniques.

The first one is based on the concept of modularity which provides information on the strength

of division of a network into communities. Networks with high modularity have dense connections between the nodes within communities but sparse connections between nodes in different communities. The second technique exploits the notion of edge betweenness, which is a measure that assigns a score to each edge. The edges that lie between many pairs of nodes have high scores which enables their easier identification. Removing these edges will leave behind just the communities themselves. The third technique is more straightforward. It is based on the assumption that the variables of the same type will tend to form *quasicliques* during the constraint acquisition process. That is, this technique finds sub-graphs with an edge density exceeding a threshold parameter.

Preference elicitation [9] mechanism solves soft constraint problems where some of the preferences are unspecified. The idea is to combine a branch & bound search with elicitation steps where candidate solutions are presented to the user who provides missing preferences on the variables. When restricted to hard constraint problems (that is, preferences are either accept or reject), this approach becomes very close to the matchmaker agent, requiring the user to be able to express her constraints one by one. The matchmaker agent [10] is an agent that interactively asks the user to provide one of the constraints of the target problem each time the system proposes an incorrect solution.

Lallouet et al. [11] have proposed a passive constraint acquisition system based on inductive logic programming. Their system is able to learn constraints from a given language that classify correctly the examples. To overcome the problem of the huge space of possible candidate constraint networks, their system requires the structure of the constraint network to be put in the background knowledge. They illustrate their approach on graph coloring problems. The positive/negative examples (i.e., correct and wrong colorations) are provided with their logical description using a set of given predicates. The background knowledge already contains all edges of the graph.

ModelSeeker [12] is a passive constraint acquisition system devoted to problems having a regular structure, such as matrix models. In *ModelSeeker* the bias contains global constraint from the global constraints catalog whose scopes are the rows, the columns, or any other structural property *ModelSeeker* can capture. It also provides an efficient ranking technique that returns the best candidate constraints representing the pattern occurring on a particular set of variables (e.g. variables composing a row).

Modelseeker can handle positive examples only. Its very specific bias allows it to quickly find candidate models when the problem has a good structure. The counterpart is that it misses any constraint that does not belong to one of the structural patterns it is able to handle. Finally, its efficiency also comes from the fact that it does not prove convergence. It provides candidate constraints. It is the user who selects the constraints that fit the best the target problem. It lists the global constraints satisfied by all examples using the *constraint seeker*, which uses multiple criteria to order the constraints according to pertinence.

Chapter 3

3 Multiple Constraint Acquisition

3.1 Motivation

We have already seen *QUACQ* using partial queries performing Quick Constraint Acquisition. With *QUACQ*, if an example e has been classified positive, all constraints rejecting it are removed from the bias B. Otherwise, when e is classified as negative, *QUACQ* learns and focuses on one constraint each time there is a negative example, whereas Multiple Constraint Acquisition(*MULTIACQ*) tries to learn more than one constraint of why the user classifies a given example as negative. *MULTIACQ* behaves just like *QUACQ* taking as input a bias B on a vocabulary (X, D) and returning a constraint network C_L equivalent to the target network C_T by asking partial queries. The motivation for *MULTIACQ* is given below.

The paper *MERGEXPLAIN: Fast Computation of Multiple Conflicts for Diagnosis* [13] discusses about computation of minimal conflict sets by introducing *MERGEXPLAIN*, a non-intrusive conflict detection algorithm which implements a divide-and-conquer strategy to decompose a problem into a set of smaller independent subproblems. The paper also mentions *QUICKXPLAIN* [5], originally developed in the context of constraint problems, is an example of a very efficient non-intrusive conflict detection technique designed to find one minimal conflict based on a divide-and-conquer strategy. One limitation of *QUICKXPLAIN* is that it will only return one conflict in each call and will be restarted with a slightly different configuration when the next search node is created.

QUICKXPLAIN applies a recursive procedure which relaxes the input set of faulty constraints C, by partitioning it into sets C_1 and C_2. If C_1 is a conflict, the algorithm continues partitioning C_1 in the next recursive call. Otherwise, if the last partitioning has split all conflicts in C, the algorithm extracts a conflict from the sets C_1 and C_2. This way, *QUICKXPLAIN* finally identifies single constraints which are inconsistent with the remaining consistent set of constraints and the background theory. This working of *QUICKXPLAIN* is analogous to *QUACQ*.

MERGEXPLAIN adopts the general divide and conquer principle of *QUICKXPLAIN*, but is designed to compute several minimal conflicts, if they exist during one problem analysis run. The main design rationale of *MERGEXPLAIN* is that (a) being able to identify multiple conflicts early on can help to speed up the overall diagnosis process; and that (b) the identification of additional conflicts is faster when we investigate only smaller subsets of the original components due to the decompose and merge strategy of *MERGEXPLAIN*.

MERGEXPLAIN accepts two sets of constraints as inputs, B as the assumed correct set of background constraints and C, the diagnosable constraints. In case C U B is inconsistent, *MERGEXPLAIN* returns a set of minimal conflicts by calling the recursive function *FINDCONFLICTS*. This function again accepts B and C as an input and returns a tuple C', ζ ; where ζ is a set of minimal

conflicts and C' subset of C is a set of constraints that does not contain any conflicts, i.e. $B\ U\ C'$ is consistent.

The logic of *FINDCONFLICTS* is similar to *QUICKXPLAIN* in that the problem is decomposed into two parts in each recursive call. However differently from *QUICKXPLAIN*, conflicts are looked for in both splits C_1 and C_2 independently and then combine the conflicts that are eventually found in the both halves. If there is a conflict in the first part and one in the second, *FINDCONFLICTS* will find them independently from each other. There might also be conflicts in C whose elements are spread across both C_1 and C_2, i.e. the set $C_1\ U\ C_2\ U\ B$ is inconsistent. The computation of a minimal conflict is done by two calls to *GETCONFLICT*. In the first call this function returns a minimal set X subset C_1' such that $U\ C_2'\ U\ B$ is a conflict. We then look for a subset of C_2', say Y, such that $Y\ u\ X$ corresponds to a minimal conflict CS. The latter is added to . In order to restore the consistency of $C_1'UC_2'UB$ we have to remove at least one element $\alpha\ \epsilon\ CS$ from either $C_1'UC_2'$. Therefore, the algorithm removes $\alpha\ \epsilon\ X \subset CS$ from C_1'.

Given a background theory B and a set of constraints C, *MERGEXPLAIN* always terminates and returns 'no solution', if B is inconsistent; if BUC is consistent; and a set of minimal conflicts, otherwise. From the algorithm design, *QUICKXPLAIN* applies a constructive conflict computation procedure prior to partitioning, whereas *MERGEXPLAIN* does the partitioning first, thereby removing multiple constraints at a time, and then uses a divide and conquer conflict detection approach. Finally, *MERGEXPLAIN* can, depending on the configuration, make a guarantee about the existence of a diagnosis given the returned conflicts without the need of computing all existing conflicts.

Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints [14] discusses about the algorithm *Compute All Minimal Unsatisfiable Subsets* (*CAMUS*), a sound and complete algorithm for producing all *Minimal unsatisfiable subsets* (*MUSes*) of an unsatisfiable constraint system. *MUSes* are explanations of type errors, and all *MUSes* are required to generate the best explanation of type errors. The set of *MUSes* is equivalent to the set of all irreducible hitting sets of the *MCSes*, and the *MCSes* are likewise all the irreducible hitting sets of the *MUSes*. The goal is to find minimal sets of clauses [15] whose removal renders the given formula satisfiable. As noted above, this is equivalent to finding Maximal Satisfiable subsets (*MSSes*) because the complement of any *MCS* (resp. *MSS*) is an *MSS* (resp. *MCS*). In the first phase, *CAMUS* finds *MCSes*. Once the entire collection of *MCSes* has been computed, the second phase of CAMUS produces all *MUSes* of the given instance by finding all irreducible hitting sets of the *MCSes*. *CAMUS* finds *MSSes* by iteratively finding the largest satisfiable subset that has not been found in a previous iteration.

Loosely said, if we were to apply the analogous ideas from *MERGEXPLAIN* [13] as it improved on *QUICKXPLAIN* [5] to *QUACQ* [2] with the concepts of MUSes and MCSes in [14], we would obtain Multiple Constraint Acquisition. The algorithm for Multiple Constraint Acquisition is as explained below.

3.2 Algorithm

Multiple Constraint Acquisition (*MULTIACQ*) *(fig 7, page 28)* starts by initializing the C_L network to the empty set. If C_L is unsatisfiable, the acquisition process will reach a collapse state. A complete assignment e satisfying the current learned network C_L and violating at least one constraint in B is computed. If such an example does not exist, then all the constraints in B are implied by C_L, and the algorithm has converged. Otherwise, the function *findAllScopes (fig 8, page 29)* is called on the example e. If e is negative, *findAllScopes* returns the set $MSes$ of minimal scopes of e. As each minimal scope in $MSes$ represents the scope of a violated constraint that must be learned, the function *findC (fig 6, page 20)* is called for each such minimal scope. It returns a constraint from C_T with the given minimal scope as scope, that rejects e. If no constraint is returned, this is a second condition for collapsing as no bias B a constraint rejecting the negative example we could be found. Otherwise, the constraint returned by *findC* is added to the learned network C_L.

Algorithme 1 : MULTIACQ

```
1  C_L ← ∅
2  while true do
3      if sol(C_L) = ∅ then return "collapse"
4      choose e in D^X accepted by C_L and rejected by B
5      if e = nil then return "convergence on C_L"
6      else
7          MSes ← ∅
8          findAllScopes (e, X, MSes)
9          foreach Y ∈ MSes do
10             c_Y ← findC(e, Y)
11             if c_Y = nil then return "collapse"
12             else C_L ← C_L ∪ {c_Y}
```

Figure 7: Multiple Acquisition Algorithm [3]

The recursive function *findAllScopes* takes as input a complete example e and a subset of variables Y (X for the first call). Function *findAllScopes* returns true if and only if there exists a minimal scope of e in Y . But *findAllScopes* fills the set $MSes$ with all the minimal scopes of e. Function *findAllScopes* starts by checking if the subset Y is already reported as a minimal scope. If this occurs, *true* is returned. It is assumed that the bias is expressive enough to learn C_T , when $K_B(e_Y) = \phi$ (i.e. there is no violated constraint in B to learn on Y), it implies that $ASK(e_Y) = yes$ and false is returned. As a third check, it is verified that a subset of Y is not reported as a minimal scope. If it is reported, $ASK(e_Y) = no$. If not, the user is asked to classify the (partial) example e_Y. If it is positive, all the constraints rejecting eY are removed from B and false is return. If $ASK(e_Y) = no$, this means that there exist minimal scopes in Y. For reporting that, *findAllScopes* is called

Function 1 : `findAllScopes` $(e,\ Y,\ MSes)$: **Boolean**

```
1  if Y ∈ MSes then return true
2  if κ_B(e_Y) = ∅ then return false
3  if ∄M ∈ MSes | M ⊂ Y then
4  |   if ASK(e_Y) = yes then
5  |   |   B ← B \ κ_B(e_Y)
6  |   |   return false
7  flag ← false
8  foreach x_i ∈ Y do
9  |   flag ← findAllScopes(e, Y\{x_i}, MSes) ∨ flag
10 if ¬flag then MSes ← MSes ∪ {Y}
11 return true
```

Figure 8: FindAllScopes Algorithm [3]

on each subset of Y built by removing one variable from Y. If any subcall to *findAllScopes* returns true, the Boolean flag will be set to *true*, which means that Y itself is not a minimal scope and *true* is returned. Otherwise, when all the subcalls of *findAllScopes* on Y subsets return $false$, this means that Y is a minimal scope and it is added to the set $MSes$.

Given a negative example, *findAllScopes* asks partial queries to compute the set of minimal scopes of constraints that explain why the user classified negatively. Function *findAllScopes* needs to explore, in the worst case, a search space containing $2|X|$ candidate scopes. A query in *findAllScopes* can be time-consuming.

Since it is not satisfactory, in an interactive process, to let the human user wait too long between two queries, *MULTIACQ* implements a heuristic to use a cutoff on the waiting time between two queries. However, if the cutoff is too short, it would not be able to explore the last branches of the search for minimal scopes, those branches including the last variables. As a result, the cutoff technique is combined with a second heuristic based on reordering the variables.

Given a call to *findAllScopes* on a complete (negative) example e, after triggering the cutoff for the first time, *findAllScopes* is called again on the same complete example e, but on a reverse order of the variables. If a second cutoff is triggered, we come back to *MULTIACQ*, generate a new example and make a shuffle on the variable order. To ensure termination of *MULTIACQ*, *findAllScopes* is forced to return at least a minimal scope before cutting off.

XP	Algorithm	$\|C_L\|$	$\bar{T}$	σ_T	$\#q$	$\bar{q}$	$\#q_c^+$	$\#q_c^-$	$\#q_p^+$
Murder	QUACQ	52	0.01	0.03	518	10.35	1	52	90
	MULTIACQ	52	**0.00**	0.06	**404**	**4.93**	2	3	248
Latin	QUACQ	**90**	0.02	0.03	1078	12.1	4	90	140
	MULTIACQ	100	**0.00**	0.05	**379**	**3.53**	8	1	200
Golomb 12	QUACQ	**323**	**0.53**	1.93	4258	6.45	1	323	61
	MULTIACQ	418	1.20	1.19	**1946**	**5.14**	0	43	567
Sudoku	QUACQ	**622**	**0.08**	1.43	10110	36.24	0	622	1107
	MULTIACQ	810	0.24	0.36	**3821**	**3.50**	10	1	2430

Table 1: Quacq Vs Multiacq [3]

3.3 Performance

If MULTIACQ is compared to QUACQ *(table 1, page 30)*, the main observation is that the use of *findAllScopes* to find all minimal scopes of a negative example reduces significantly the number of queries required for convergence.

Table 1 displays the performance of MULTIACQ and QUACQ. The notations are $|C_L|$: Size of the learned network, T: Average time needed to generate a query in seconds, σ_T of T: Standard deviation, #q: Total number of queries, $\bar{q}$: average size of all queries, $\#q_c^+$: number of complete positive (resp. negative) queries, (resp. $\#q_c^-$): number of complete negative queries, $\#q_p^+$: number of partial positive queries.

If we compare MULTIACQ to QUACQ, the main observation is that the use of *findAllScopes* to find all minimal scopes of a negative example reduces significantly the number of queries required for convergence. Let us take the Murder problem.

- *MULTIACQ* exhibits a gain of 22% on the number of queries.
- MULTIACQ reduces significantly the average size of the queries (52%), which is probably easier to answer by the user.
- *MULTIACQ* needs only 3 complete negative examples instead of 52 for QUACQ.
- The same observations on the performance of MULTI- ACQ comparing to QUACQ are true on the other problem instances:

- Number of queries reduction (i.e., gain of 65% on Latin Square, 55% on Golomb Rulers and 73% on Sudoku).
- The average size of the queries (i.e., 71% on Latin- Square, 20% on Golomb Rulers and 90% on Sudoku).
- Obviously, we need less complete negative example (i.e. only one instead of 90 on Latin Square, 43 instead of 323 on Golomb Rulers and only one instead of 622 on Sudoku)
- The number of constraints learned by MULTIACQ is always greater than or equal to the number of constraints learned using QUACQ. MULTIACQ can learn redundant constraints, which is not the case using QUACQ. Take three constraints c_1, c_2 and c_3 such that c_1 c_2 c_3. If we generate a negative example e that violates the three constraints, MULTIACQ will return three minimal scopes corresponding to the three constraints. By contrast, QUACQ will return only one minimal scope, let us say the one of c_2. If in a second iteration QUACQ learns c_1, c_3 is automatically satisfied in any next iteration and will never be learned by *QUACQ*.
- A last observation we can make on Table 1 is related to the average time needed to generate a query. If we take the instances of Golomb Rulers and Sudoku, we observe that *MULTIACQ* respectively needs twice more time (1.20s instead of 0.53s) and three times more time (0.24s instead of 0.08s) than QUACQ. On these two instances, generating a query is starting to become time-consuming. Table 1 does not report the results of MACQ-CO because it performs exactly the same as the basic version of MULTIACQ. The reason is that the average time (and standard deviation) needed to generate a query is significantly below the value of the cutoff (5s in our case).
- The observations made on Table 1 remain true on RLFA and Langford instances. The difference is in the average time needed to generate a query. Using MULTIACQ, the time increases significantly. For instance, on Langford, QUACQ takes about 0.03 to 0.13 seconds to generate a query whereas MULTIACQ takes 4.24 to 23.03 seconds. On Zebra and Graceful Graphs instances, we observe the same behavior as on the previous two instances for the time to generate a query. However, as for the number of queries, QUACQ is better than MULTIACQ. This can be explained by the high number of partial positive queries asked by MULTIACQ (e.g., for Zebra we have $\#q_p^+ = 592$ with MULTIACQ against 255 with QUACQ). By definition, MULTIACQ has a trend to ask small queries and then, needs an important number of partial positive queries to reduce the bias and thus, to converge.

Analysis

- If $ASK(e_Y) = yes$ then for any $Y \subseteq Y'$ we have $ASK(e_{Y'}) = yes$
- If $ASK(e_Y) = no$ then for any $Y' \supseteq Y$ we have $ASK(e_{Y'}) = no$.
- Given a bias B and a target network C_T representable by B, function *findAllScopes* is correct.
- Given a bias B built from a language λ of bounded arity constraints, and a target network C_T , *MULTIACQ* uses $O(|C_T|.(|X| + |\lambda|) + |B|)$ queries to prove convergence on the target network or to collapse.

Chapter 4

4 Conclusion & Future Scope

4.1 Future Scope

Learning the Parameters of Global Constraints Using Branch-and-Bound [16] uses their algorithm to constraints such as *Among* and *Sequence* commonly used in timetabling. The paper uses branch and bound for this and introduces *SubsetFocus* which is useful in modelization of timetabling problems. This paper is important due to the focus on global constraints. There are Machine learning techniques like Neural networks and Decision trees that recognize the solution to a problem and then embed the trained neural network or decision trees into a global constraint.

The paper *Embedding Decision Trees and Random Forests in Constraint Programming* [17], discusses about Table based Encoding and MDD based Encoding. A Decision tree can be converted to a set of classification rules by interpreting each path from root to leaf as logical implication. A simple approach to encode a Decision tree in Constraint Programming consists in translating each implication into a boolean meta-constraint. Embedding random forests in Constraint Programming requires to: embed in the CP model each Decision tree from the forest, and define a constraint model for the mode computation, i.e. for aggregating the DT results and obtain the final classification.

Constraint Acquisition Using Recommendation Queries [7] lists out that an interesting direction would be to use reinforcement learning to decide on the use of neighborhood-based predictions or path-ensemble based predictions to predict constraints that are likely to belong to the target network.

Constraint Acquisition [2] tells that background knowledge is a well-known technique in machine learning consisting in using properties of the problem to learn to reduce the bias as much as possible so that the version space to explore becomes small enough. Complex queries is another way to speed up convergence by allowing a more informative communication between the user and the learner so as to capture more information.

Improving the efficiency of CAMUS [14] in finding MCSes, MSSes, and MUSes can also improve the time taken for finding all scopes. Also new techniques for computing a diverse subset of the MUSes containing some desired structural characteristic is also preferred for time improvements.

4.2 Conclusion

This seminar is a concise literature survey to find various aspects of Constraint Acquisition: from passive constraint acquisition to active constraint acquisition and how different acquisition tech-

niques and methods motivated the development of Multiple Constraint Acquisition. While in the process, I have been able to list out the challenges and the future scope.

The weakness of passive learning is that the user needs to provide diverse examples for the target set of constraints to be learned. It is noticeable that *CONACQ.1* often requires a huge number of examples to converge to the target network. Moreover, the user has to verify for oneself when it comes to candidate constraints and finalize for which constraints would fit the target problem. Passive learning leaves the user in a helpless position to acquire lots of positive examples when acquiring positive solutions is difficult or even impossible. Despite all these drawbacks, even when *CONACQ.1* has not converged, the most specific network in the version space is often quite close to the target network.

Even though active learning solves many of the problems associated with passive learning, it is associated with two important computational challenges: how the system generates a useful query, and how many queries are needed for the system to converge to the target set of constraints. It has been shown that the number of membership queries required to converge to the target set of constraints can be exponentially large. *CONACQ.2*, by its active behavior, reduces the number of examples needed to converge to the target network. Another advantage is that the user need not be a human. It might be a previous system developed to solve the problem.

QUACQ has two main advantages over learning with membership queries: Small average size of queries q, which are probably easier to answer by the user, and lesser time to generate queries. Generating good queries in *QUACQ* is not computationally difficult and that when the bias increases in size, the number of queries increase only in logarithmic shape. *Conacq.2* needs to find examples that violate exactly one constraint of the bias to make progress towards convergence which can be expensive to compute, preventing the use of *Conacq.2* on large problems. *QUACQ* and *MULTIACQ*, on the other hand, uses cheap heuristics to generate queries.

QUACQ always converges on the target constraint network in a polynomial number of queries. *QUACQ* and *MULTIACQ* as opposed to existing techniques does not need provide positive examples to converge which can be very useful when the problem has not been previously solved. Problems with dense constraint networks (as Sudoku) require a number of queries that could be too large for a human user. An interesting direction would be to combine *ModelSeeker* and *QUACQ* to quickly learn global constraints and use QUACQ to finalize the model.

For many concept classes, the use of membership queries in conjunction with equivalence queries is known to dramatically accelerate the learning process. Though equivalence queries are not considered in constraint acquisition, and membership queries alone are not powerful enough to guarantee convergence in a polynomial number of queries, the use of membership queries in conjunction with a given training set can substantially improve the acquisition process. More studies should focus on the translation of problem description written in natural language in textual form as well as speech into a formal model.

References

[1] C. Bessiere, F. Koriche, N. Lazaar, and B. O'Sullivan, "Constraint acquisition," *Artificial Intelligence*, vol. 244, no. Supplement C, pp. 315 – 342, 2017, combining Constraint Solving with Mining and Learning. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370215001162

[2] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, and T. Walsh, "Constraint acquisition via partial queries," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, pp. 475–481. [Online]. Available: http://dl.acm.org/citation.cfm?id=2540128.2540198

[3] R. Arcangioli and N. Lazaar, "Multiple constraint acquisition," in *Proceedings of the 2015 International Conference on Constraints and Preferences for Configuration and Recommendation and Intelligent Techniques for Web Personalization - Volume 1440*, ser. CPCR+ITWP'15. Aachen, Germany, Germany: CEUR-WS.org, 2015, pp. 16–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=2907084.2907088

[4] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, "Using constraint programming to solve sudoku puzzles," in *2008 Third International Conference on Convergence and Hybrid Information Technology*, vol. 2, Nov 2008, pp. 926–931.

[5] U. Junker, "Quickxplain: Preferred explanations and relaxations for over-constrained problems," in *Proceedings of the 19th National Conference on Artifical Intelligence*, ser. AAAI'04. AAAI Press, 2004, pp. 167–172. [Online]. Available: http://dl.acm.org/citation.cfm?id=1597148.1597177

[6] N. L. Christian Bessière, Remi Coletta, "Solve a constraint problem without modeling it," *ICTAI: International Conference on Tools with Artificial Intelligence*, pp. 1–7, Nov 2014. [Online]. Available: www.lirmm.fr/~bessiere/stock/ictai14.pdf

[7] A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar, and E. H. Bouyakhf, "Constraint acquisition with recommendation queries," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 720–726. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060621.3060722

[8] A. Daoudi, N. Lazaar, Y. Mechqrane, C. Bessiere, and E. H. Bouyakhf, "Detecting types of variables for generalization in constraint acquisition," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, Nov 2015, pp. 413–420.

[9] M. Gelain, M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh, "Elicitation strategies for soft constraint problems with missing preferences: Properties, algorithms and experimental studies," *Artificial Intelligence*, vol. 174, no. 3, pp. 270 – 294, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370209001453

[10] E. C. FREUDER and R. J. WALLACE, "Suggestion strategies for constraint-based matchmaker agents," *International Journal on Artificial Intelligence Tools*, vol. 11, no. 01, pp. 3–18, 2002. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0218213002000769

[11] *On Learning Constraint Problems*, vol. 1, 10 2010.

[12] N. Beldiceanu and H. Simonis, *A Model Seeker: Extracting Global Constraint Models from Positive Examples*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 141–157. [Online]. Available: https://doi.org/10.1007/978-3-642-33558-7_13

[13] K. Shchekotykhin, D. Jannach, and T. Schmitz, "Mergexplain: Fast computation of multiple conflicts for diagnosis," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, pp. 3221–3228. [Online]. Available: http://dl.acm.org/citation.cfm?id=2832581.2832698

[14] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, Jan 2008. [Online]. Available: https://doi.org/10.1007/s10817-007-9084-z

[15] M. G. de la Banda, P. J. Stuckey, and J. Wazny, "Finding all minimal unsatisfiable subsets," in *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, ser. PPDP '03. New York, NY, USA: ACM, 2003, pp. 32–43. [Online]. Available: http://doi.acm.org/10.1145/888251.888256

[16] É. Picard-Cantin, M. Bouchard, C.-G. Quimper, and J. Sweeney, *Learning the Parameters of Global Constraints Using Branch-and-Bound*. Cham: Springer International Publishing, 2017, pp. 512–528. [Online]. Available: https://doi.org/10.1007/978-3-319-66158-2_33

[17] A. Bonfietti, M. Lombardi, and M. Milano, *Embedding Decision Trees and Random Forests in Constraint Programming*. Cham: Springer International Publishing, 2015, pp. 74–90. [Online]. Available: https://doi.org/10.1007/978-3-319-18008-3_6