Time-Bounded Query Generator for Constraint Acquisition

Hajar Addi, Christian Bessière, Redouane Ezzahir, Nadjib Lazaar

Time-bounded Query Generator for Constraint Acquisition*

Hajar Ait Addi[1], Christian Bessiere[2], Redouane Ezzahir[1], and Nadjib Lazaar[2]

[1] LISTI/ENSA, University of Ibn Zohr, Morocco
[2] University of Montpellier, CNRS, France
{hajar.aitaddi,r.ezzahir}@uiz.ac.ma, {bessiere,lazaar}@lirmm.fr

Abstract. QUACQ is a constraint acquisition algorithm that assists a non-expert user to model her problem as a constraint network. QUACQ generates queries as examples to be classified as positive or negative. One of the drawbacks of QUACQ is that generating queries can be time-consuming. In this paper we present TQ-GEN, a time-bounded query generator. TQ-GEN is able to generate a query in a bounded amount of time. We rewrite QUACQ to incorporate the TQ-GEN generator. This leads to a new algorithm called T-QUACQ. We propose several strategies to make T-QUACQ efficient. Our experimental analysis shows that thanks to the use of TQ-GEN, T-QUACQ dramatically improves the basic QUACQ in terms of time consumption, and sometimes also in terms of number of queries.

1 Introduction

Constraint programming (CP) has made considerable progress over the last forty years, becoming a powerful paradigm for modeling and solving combinatorial problems. However, modeling a problem as a constraint network still remains a challenging task that requires some expertise in the field. Several constraint acquisition systems have been introduced to support the uptake of constraint technology by non-experts. Freuder and Wallace proposed the matchmaker agent [7]. This agent interacts with the user while solving her problem. The user explains why she considers a proposed solution as a wrong one. Lallouet et al. proposed a system based on inductive logic programming with the use of the structure of the problem as a background knowledge [10]. Beldiceanu and Simonis have proposed MODELSEEKER, a system devoted to problems with regular structures and based on the global constraint catalog [2]. Bessiere et al. proposed CONACQ, which generates membership queries (i.e., complete examples) to be classified by the user [4, 6]. Shchekotykhin and Friedrich have extended CONACQ to allow the user to provide *arguments* as constraints to speed-up the convergence [12].

Bessiere et al. proposed QUACQ (for Quick Acquisition), an active learning system that is able to ask the user to classify partial queries [3,5]. QUACQ iteratively computes membership queries. If the user says *yes*, QUACQ reduces the search space by discarding all constraints violated by the positive example. When the answer is *no*, QUACQ finds the scope of one of the violated constraints of the target network in a number of

* This work was supported by Scholarship No.7587 of the EU METALIC non redundant program.

queries logarithmic in the size of the example. This key component of QUACQ allows it to always converge on the target set of constraints in a polynomial number of queries. Arcangioli et al. have proposed the MULTIACQ system as an attempt to make QUACQ more efficient in practice in terms of number of queries [1]. Instead of finding the scope of one constraint, MULTIACQ reports all the scopes of constraints of the target network violated by the negative example. Despite the good theoretical bound of QUACQ-like approaches in terms of number of queries, generating a membership query is NP-hard. It can then be too time-consuming when interacting with a human user. For instance, QUACQ can take more than 20 minutes to generate a query during the acquisition process of the sudoku constraint network.

In this paper, we introduce TQ-GEN, a time-bounded query generator. TQ-GEN generates queries in an amount of time not exceeding a waiting time upper bound. We incorporate the TQ-GEN generator into the QUACQ algorithm to reduce the time complexity of generating queries. This leads to a new version called T-QUACQ. Our theoretical and experimental analyses show that the bounded waiting time between queries in T-QUACQ is at the risk of reaching a *premature convergence* state and asking more queries. We then propose strategies to make T-QUACQ efficient. We experimentally evaluate the benefit of these strategies on several benchmark problems. The results show that T-QUACQ combined with a good strategy dramatically improves QUACQ not only in terms of time needed to generate queries but also in number of queries, while achieving the convergence state in most cases.

The rest of this paper is organized as follows. Section 2 presents the necessary background on constraint acquisition. Section 3 presents the algorithm TQ-GEN for time-bounded query generation. Section 4 describes how we use TQ-GEN in QUACQ to get the T-QUACQ algorithm. Section 5 analyzes the correctness of the algorithm. Experimental results and strategies to make T-QUACQ efficient are reported in Section 6. Section 7 concludes the paper

2 Background

The constraint acquisition process can be seen as an interplay between the user and the learner. User and learner need to share a *vocabulary* to communicate. We suppose this vocabulary is a set of n variables $X = \{x_1, \ldots, x_n\}$ and a domain $D = \{D(x_1), \ldots, D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values for x_i. A constraint c_Y is defined by a sequence Y of variables of X, called *the constraint scope*, and the relation c over D of *arity* $|Y|$. An assignment e_Y on a set of variables $Y \subseteq X$ *violates* a constraint c_Z (or e_Y is *rejected* by c_Z) if $Z \subseteq Y$ and the projection e_Z of e_Y on the variables in Z is not in c. A *constraint network* is a set C of constraints on the vocabulary (X,D). An assignment on X is a *solution* of C if and only if it does not violate any constraint in C. $sol(C)$ represents the set of solutions of C.

In addition to the vocabulary, the learner owns a *language* Γ of relations, from which it can build constraints on specified sets of variables. Adapting terms from machine learning, the *constraint bias*, denoted by B, is a set of constraints built from the constraint language Γ on the vocabulary (X,D), from which the learner builds the constraint network. We denote by $B[Y]$ the set of all constraints c_Z in B, where

$Z \subseteq Y$. The *target network* is a network C_T such that for any example $e \in D^X = \Pi_{x_i \in X} D(x_i)$, e is a solution of C_T if and only if e is a solution of the problem that the user has in mind.

A *membership query* $\text{ASK}(e)$ is a classification question asked to the user, where e is a *complete* assignment in D^X. The answer to $\text{ASK}(e)$ is *yes* if and only if $e \in sol(C_T)$. A *partial query* $\text{ASK}(e_Y)$, with $Y \subseteq X$, is a classification question asked to the user, where e_Y is a *partial* assignment in $D^Y = \Pi_{x_i \in Y} D(x_i)$. The answer to $\text{ASK}(e_Y)$ is *yes* if and only if e_Y does not violate any constraint in C_T. A classified assignment e_Y is called a positive or negative *example* depending on whether $\text{ASK}(e_Y)$ is *yes* or *no*. For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y.

We now define *convergence*, which is the constraint acquisition problem we are interested in. We are given a set E of (complete/partial) examples labeled by the user as positive or negative. We say that a constraint network C *agrees with* E if examples labeled as positive in E do not violate any constraint in C, and examples labeled as negative violate at least one constraint in C. The learning process has *converged* on the learned network $C_L \subseteq B$ if:

1. C_L agrees with E,
2. For any other network $C' \subseteq B$ agreeing with E, we have $sol(C') = sol(C_L)$.

We say that the learning process reached a *premature convergence* if only (1) is guaranteed. If there does not exist any $C_L \subseteq B$ such that C_L agrees with E, we say that we have *collapsed*. This can happen when $C_T \not\subseteq B$.

We finally define the class of biases that are *good* for a given time limit, that is, those biases on which bounding the query generation time does not hurt.

Definition 1 (τ-good). *Given a bias B on a vocabulary (X,D), given the maximum arity k of a constraint in B, and given τ a time limit, B is* τ-good *on (X,D) if and only if $\forall Y \subset X$ such that $|Y| = k$, $\forall C_i, C_j \subset B[Y]$, finding an assignment e on Y such that $e \in sol(C_i) \setminus sol(C_j)$, or proving that none exists, takes less than τ.*

3 Time-bounded Query Generation

To be able to exhibit its nice complexity in number of queries, QUACQ must be able to generate *non redundant* queries. A query is *non redundant* if, whatever the user's answer, it allows us to reduce the learner's version space (i.e., the subset of 2^B currently agreeing with all already classified examples). In the context of QUACQ, a query $\text{ASK}(e)$ is non redundant if e does not violate any constraint in the currently learned network C_L, and it violates at least one constraint of the current bias B in which we look for the missing constraints (i.e., $\kappa_B(e) \neq \emptyset$), We denote such an example e by $e \models (C_L \wedge \neg B)$. QUACQ has to solve an NP-hard problem to generate a non redundant query. Therefore, the user can be asked to wait a long time from a query to another.

We propose TQ-GEN, a query generator able to generate a query in a bounded amount of time (`time_bound`). We will see later that this bounded time is at the risk of reaching *premature convergence* and/or asking more queries than necessary. The

Algorithm 1: TQ-GEN

```
1  In α, τ, time_bound: parameters;
2  InOut ℓ: parameter; B: bias; C_L: learned network;
3  time ← 0 ;
4  while B ≠ ∅ and time < time_bound do
5  |  τ ← min(τ, time_bound − time) ;
6  |  ℓ ← max(ℓ, minArity(B)) ;
7  |  choose Y ⊆ X s.t. |Y| = ℓ ∧ B[Y] ≠ ∅ ;
8  |  e_Y ← solve(C_L[Y] ∧ ¬B[Y]) in t < τ ;
9  |  if e_Y ≠ nil then return e_Y ;
10 |  else
11 |  |  if t < τ then
12 |  |  |  C_L ← C_L ∪ B[Y] ; B ← B \ B[Y] ;
13 |  |  else ℓ ← ⌊α · ℓ⌋ ;
14 |  |  time ← time + t ;
15 return nil ;
```

idea behind TQ-GEN is that instead of looking for an assignment e on X such that $e \models (C_L \wedge \neg B)$, we look for a partial assignment e_Y such that $e_Y \models (C_L[Y] \wedge \neg B[Y])$, for some set $Y \subseteq X$.

3.1 Description of TQ-GEN

The algorithm TQ-GEN (see Algorithm 1) takes as input the set of variables X, a reduction factor $\alpha \in]0,1[$, a solving timeout τ, a time limit to generate a query `time_bound`, an expected query size ℓ, a current bias of constraints B, and a current learned network C_L. We start by initializing the counter `time` of the time TQ-GEN has already consumed in its main loop. In line 5, we set τ so that the next execution of the main loop cannot exceed `time_bound`. In line 7, we choose a subset Y of size ℓ. To be able to generate a non redundant query on Y, it is required that $B[Y]$ is not empty. To guarantee that such an Y exists, we need ℓ to never be smaller than the smallest arity in B (line 6). In line 8, TQ-GEN tries to generate a query on Y of size ℓ in a time less than τ. If such a query is found in less than τ, we return it in line 9. Otherwise, either $C_L[Y] \wedge \neg B[Y]$ is unsatisfiable and τ is sufficient to prove it, or $C_L[Y] \wedge \neg B[Y]$ is too hard to be solved in τ. If $C_L[Y] \wedge \neg B[Y]$ is unsatisfiable, the constraints in $B[Y]$ are redundant to C_L and they can be removed from B (line 12). These constraints have to be put in C_L to avoid generating later a query violating one of these redundant constraints, but they are useless in terms of the set of solutions represented by C_L. They can safely be removed from C_L at the end of the learning process. If $C_L[Y] \wedge \neg B[Y]$ is too hard, we reduce the expected query size ℓ with a factor α (line 13). In line 14, the time spent to try to generate a query is recorded in order to ensure that TQ-GEN will never exceed the allocated time `time_bound` (see line 4). The last attempt shall not exceed the remaining time (i.e., `time_bound` − `time`) (line 5).

Algorithm 2: T-QUACQ

```
1  C_L ← ∅ ;
2  initialize(α, τ, time_bound, ℓ) ;
3  while true do
4      e ← TQ-GEN (α,τ,time_bound,ℓ,B,C_L) ;
5      if e = nil then
6          if B = ∅ then return "convergence on C_L" ;
7          return "premature convergence on C_L" ;
8      if Ask(e) = yes then
9          B ← B \ κ_B(e) ;
10         adjust(ℓ, yes) ;
11     else
12         c ← FindC(e,FindScope(e,∅,X,false)) ;
13         if c ≠ nil then C_L ← C_L ∪ {c} ;
14         else return "collapse" ;
15         adjust(ℓ, no);
```

4 Using the TQ-GEN Algorithm in QUACQ

In this section, we present T-QUACQ (Algorithm 2), an integration of TQ-GEN into QUACQ. T-QUACQ differs from the basic version presented in [3] at the shaded lines (i.e., lines 2, 4, 7, 10 and 15).[1]

T-QUACQ initializes the constraint network C_L to the empty set (line 1). In line 2, the parameters of TQ-GEN are initialized such that $\alpha \in]0..1[$ and $\ell \in [\texttt{minArity}(B), |X|]$. In line 4, we call TQ-GEN to generate a query in bounded time. If no query exists (i.e., $B = \emptyset$), then the algorithm reaches a convergence state (line 6). If a query exists and TQ-GEN is not able to return it in the allocated time, T-QUACQ reaches a premature convergence (line 7). Otherwise, we propose the example e to the user, who will answer by yes or no (line 8). If the answer is yes, we can remove from B the set $\kappa_B(e)$ of all constraints in B that reject e (line 9). We can also adjust the expected size of the next query following a given strategy (line 10). This function is discussed later in section 6.3. If the answer is no, we are sure that e violates at least one constraint of the target network C_T. We then act exactly as QUACQ by calling the function `FindScope` to discover the scope of one of these violated constraints and `FindC` to select which constraint with the given scope is violated by e (line 12). If a constraint c is returned, we know that it belongs to the target network C_T, we then add it to the learned network C_L (line 13). If no constraint is returned (line 14), this is a condition for collapsing as we could not find in B a constraint rejecting one of the negative examples. Functions `FindScope` and `FindC` are used exactly as they appear in [3]. When the answer is no, we can also adjust the expected size of the next query following a given strategy (line 15).

[1] QUACQ also contains a line for returning "collapse" when detecting an inconsistent learned network. This line has been dropped from T-QUACQ because we allow it to learn a target network without solutions.

5 Theoretical Analysis

In this section we analyze the correctness of TQ-GEN and T-QUACQ. The role of TQ-GEN is to return a query that is non redundant with all queries already asked to the user.

Proposition 1 (Soundness). TQ-GEN *is sound.*

Proof. The only place where TQ-GEN returns a query is line 9. By construction, e_Y is an assignment which is solution of $C_L[Y]$ and that violates at least one constraint from $B[Y]$ (line 8). Thus, $\kappa_B(e_Y) \neq \emptyset$, and by definition e_Y is a non redundant query. □

Proposition 2 (Termination). *Given a bias B on the vocabulary (X,D), if* `time_bound` $< \infty$ *or if B is τ-good on (X,D), then* TQ-GEN *terminates.*

Proof. If `time_bound` $< \infty$, it is trivial. Suppose now that B is τ-good on (X,D), `time_bound` $= \infty$, and TQ-GEN never goes through line 9 (which would terminate TQ-GEN). At each execution of its main loop, TQ-GEN executes either line 12 or line 13. ℓ decreases strictly at each execution of line 13. Hence, after a finite number of times, ℓ will be less than or equal to the maximum arity in B. As B is τ-good, the cutoff τ will no longer be reached in line 8, and the next executions of the loop will all go through line 12. Thanks to line 6, ℓ cannot be less than the smallest arity in B. Thus, the set Y chosen in line 7 is guaranteed to have a non empty $B[Y]$. As a result, B strictly decreases in size at each execution of line 12, B will eventually be empty, and TQ-GEN will terminate. □

We now show that under some conditions TQ-GEN cannot miss a non redundant query, if one exists.

Proposition 3 (Completeness). *If the bias B is τ-good, and* `time_bound` $> (|B| + \lceil log_\alpha(\frac{k}{n})\rceil) \cdot \tau$*, with $n = |X|$ and k the maximum arity in B, then* TQ-GEN *is complete.*

Proof. TQ-GEN finishes by either returning a query in line 9 or nil in line 15. If a query is returned, we are done as TQ-GEN is sound (Proposition 1). Suppose nil is returned in line 15. According to the assumption on `time_bound` and the fact that each execution of the main loop of TQ-GEN takes at most τ seconds, we know that TQ-GEN has enough time to execute $|B| + \lceil log_\alpha(\frac{k}{n})\rceil$ times its main loop before returning nil in line 15. In each of these executions, line 12 or line 13 is executed. Each time line 13 is executed, ℓ is reduced by multiplying it by the factor $\alpha \in]0..1[$. As ℓ cannot be greater than n when entering TQ-GEN, after $\lceil log_\alpha(\frac{k}{n})\rceil$ executions, we are guaranteed that $\ell \leq n \cdot \alpha^{\lceil log_\alpha(\frac{k}{n})\rceil} \leq n \cdot \frac{k}{n} = k$. As B is τ-good, TQ-GEN will be able to solve the formula $C_L[Y] \wedge \neg B[Y]$ in less than τ seconds for all Y, as $|Y| = \ell \leq k$. As a result, TQ-GEN has enough time for $|B|$ executions of the loop before reaching the `time_bound` limit. Thanks to line 7, we know that the set Y has a non empty $B[Y]$. Thus, line 12 removes at least one constraint from B, and B will be emptied before the time limit. Therefore, we have converged, and there does not exist any non redundant query. □

Theorem 1. *If $C_T \subseteq B$,* T-QUACQ *is guaranteed to reach (premature) convergence. If in addition B is τ-good and* `time_bound` $> (|B| + \lceil log_\alpha(\frac{k}{n})\rceil) \cdot \tau$*, with $n = |X|$ and k the maximum arity in B, then* T-QUACQ *converges.*

Proof. (Sketch.) We first prove premature convergence. Let E be the set of all examples generated during the execution of T-QUACQ and C_L be the returned network. If C_L does not agree with E this means that there exists $e_Y \in E$ such that e_Y is positive and $e_Y \not\models C_L$, or e_Y is negative and $e_Y \models C_L$. As `FindScope` and `FindC` are sound, we only consider examples classified in line 8 of T-QUACQ. Suppose first that in line 8, e_Y is positive (e_Y^+). By construction, e_Y has been generated by satisfying $C_L[Y]$ (line 4), that is, $\nexists c_Z \in C_L[Y] \mid e_Y \not\models c_Z$ at the time of generating e_Y. As line 9 removes from B all constraints rejecting e_Y, we are guaranteed that C_L agrees with $\{e_Y^+\}$ at the end of T-QUACQ. Suppose now that e_Y is negative (e_Y^-). As $C_T \subseteq B$, `FindC` returns a constraint c rejecting e_Y (line 12) and c is added to C_L in line 13. Thus, C_L agrees with $\{e_Y^-\}$ at the end.

We now prove that T-QUACQ converges when B is τ-good and `time_bound` $> (|B| + \lceil log_\alpha(\frac{k}{n})\rceil) \cdot \tau$. By Proposition 3 we know that under this assumption, TQ-GEN always returns a non redundant query if one exists. As a result, TQ-GEN returns nil only when B has been emptied of all its redundant constraints in line 9, which means that T-QUACQ has converged on C_L. □

6 Experiments

In this section, we experimentally analyze our new algorithms. We first describe the benchmark instances. Second, we evaluate the validity of the time-bounded query generation by comparing a baseline version of T-QUACQ to the QUACQ algorithm. This baseline version allows us to observe the fundamental characteristics of the approach. Based on these observations, we discuss possible strategies and parameter settings that may make our approach more efficient. The only parameter we will keep fixed in all our experiments is `time_bound`, that we set to 1 seconds, as we consider it as an acceptable waiting time for a human user [9]. All tests were performed using the Choco solver[2] version 4.0.4 with a simulation run time cutoff of 3 hours, 2 Gb of Java VM allowed memory on an Intel(R) Xeon(R) @ 3.40GHz.

6.1 Benchmarks

We used four benchmarks from the original QUACQ paper [3] (Random, Sudoku, Golomb ruler, and Zebra), and two additional ones (Latin square, Graceful graphs).

Random. We generated binary random target networks with 50 variables, domains of size 10, and m binary constraints. The binary constraints are selected from the language $\Gamma = \{=, \neq, \leqslant, \geqslant, <, >\}$. We have launched our experiments with $m = 12$, and $m = 122$.

Sudoku. The sudoku logic puzzle with 9×9 grid must be filled with numbers from 1 to 9 in such a way that all the rows, all the columns, and the 9 non overlapping 3×3

[2] www.choco-solver.org

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. We use a bias of 19,440 binary constraints taken from the language $\Gamma = \{=, \neq \leqslant , \geqslant, <, >\}$.

Golomb ruler. (prob006 in [8]) The problem is to find a ruler where the distance between any two marks is different from that between any other two marks. Golomb ruler is encoded as a target network with n variables corresponding to the n marks. For our experiments, we selected the 8, 12, 16 and 20 marks ruler instances with bias of 660, 3,698, 12,552, and 32,150 constraints, respectively, generated using the language $\Gamma = \{=_0, \neq_0, \leqslant, \geqslant, < , >, \shortparallel^{zt}_{xy} , \nmid^{zt}_{xy}\}$ where $=_0$ and $\neq_0$ respectively denote the unary constraints "equal zero" and "not equal zero", and $\shortparallel^{zt}_{xy}$ and $\nmid^{zt}_{xy}$ respectively denote the distance constraints $|x - y| = |z - t|$ and $|x - y| \neq |z - t|$.

Latin square. A Latin square is an $n \times n$ array filled with n different Latin letters, each occurring exactly once in each row and exactly once in each column. We have taken $n = 10$ and the target network is built with 900 binary $\neq$ constraints on rows and columns. We use a bias of 29,700 constraints built from the language $\Gamma = \{=, \neq, \leqslant, \geqslant, <, >\}$.

Zebra. Lewis Carroll's zebra problem has a single solution. The target network has 25 variables of domain size 5 with 5 cliques of "$\neq$" constraints and 14 additional constraints given in the description of the problem. We use a bias of 3,250 unary and binary constraints taken from a language with 20 basic arithmetic and distance constraint.

Graceful graphs. (prob053 in [8]) A labeling f of the n nodes of a graph with q edges is graceful if f assigns each node a unique label from 0,1, . . . ,q and when each edge (x, y) is labeled with $|f(x) - f(y)|$, the edge labels are all different. The target network has node-variables $x_1, x_2, \ldots x_n$, each with domain $\{0,1, \ldots ,q\}$, and edge-variables $e_1, e_2, \ldots e_q$, with domain $\{1, 2, \ldots , q\}$. The constraints are: $x_i \neq x_j$ for all pairs of nodes, $e_i \neq e_j$ for all pairs of edges, and $e_k = |x_i - x_j|$ if edge e_k joins nodes i and j. The constraints of B were built from the language $\Gamma = \{\neq , =, \shortparallel^{z}_{xy} , \nmid^{z}_{xy}\}$ where $\shortparallel^{z}_{xy}$ and $\nmid^{z}_{xy}$ denote respectively the distance constraints $z = |x - y|$ and $z \neq |x - y|$. We used three instances that accept a graceful labeling [11]: $GG(K_4 \times P_2)$, $GG(K_5 \times P_2)$, and $GG(K_4 \times P_3)$, whose number of variables is 24, 35, and 38 respectively, and bias size is 12,696, 40,460, and 52,022 respectively.

For all our benchmarks, the bias contains all the constraints that can be generated from the relations in the given language. That is, for a commutative relation c (resp. non-commutative relation c') of arity r, the bias contains all possible constraints c_Y (resp. c'_Y), where Y is a subset (resp. an ordered subset) of X of size r.

6.2 Baseline version of T-QUACQ

The purpose of our first experiment is to validate the approach of time-bounded query generation and to understand the basics of its behavior. We defined a baseline version of T-QUACQ, called T-QUACQ.0, that we compare with QUACQ. T-QUACQ, presented in Algorithm 2, is parameterized with `time_bound`, α, τ and ℓ used by function $initialize$, and what function $adjust$ does.

Once `time_bound` has been fixed, as said above, to 1 seconds, there remains to specify the other parameters and function $adjust$. In T-QUACQ.0, to remain as close as possible to the original QUACQ, we set ℓ to $|X|$ and the function $adjust$ at lines 10 and

Table 1: T-QUACQ.0 versus QUACQ (`time_bound` $= 1s$)

CSP	Algorithm	$(\alpha, \tau$ (in $ms))$	$\#q$	$totT$ (in $seconds$)	$\#Conv$	$\%Conv$
sudoku 9×9	QUACQ	-	9,053	2,810	-	100%
	T-QUACQ.0	$(0.5, 0.001)$	12	14	0	1%
		$(0.5, 0.024)$	9,132	37	10	100%
		$(0.5, 5)$	9,612	62	10	100%
		$(0.5, 900)$	9,557	41	5	94%
$GG(K_5 \times P_2)$	QUACQ	-	4,898	3,144	-	100%
	T-QUACQ.0	$(0.5, 0.001)$	11	62	0	1%
		$(0.5, 0.024)$	7,495	56	0	93%
		$(0.5, 5)$	5,610	43	10	100%
		$(0.5, 900)$	1,888	40	0	41%

15 of Algorithm 2 simply resets ℓ to $|X|$. The impact of the parameters α and τ will be discussed later. For this first comparison between T-QUACQ.0 and QUACQ, we use two CSP instances: sudoku and $GG(K_5 \times P_2)$. They are good candidates for this analysis because QUACQ can be very time-consuming to generate queries on them.

Table 1 reports the comparison of QUACQ and our baseline version T-QUACQ.0 on the sudoku and $GG(K_5 \times P_2)$ instances. The performance of T-QUACQ.0 is averaged over ten runs on each instance. In this first experiment, we have arbitrarily set α to 0.5. $\#q$ denotes the total number of asked queries, $totT$ denotes the total time of the learning process, $\#Conv$ denotes the number of runs of T-QUACQ.0 in which it reached convergence, and $\%Conv$ denotes the average of the convergence rate over the ten runs. We estimated the convergence rate by the formula $100.\frac{|C_T|-\#missingto(C_L)}{|C_T|}$, where $\#missingto(C_L)$ is the number of constraints that have to be added to the learned network C_L to make it equivalent to the target network C_T.

From Table 1, we observe that when $\tau = 5ms$, for both instances, T-QUACQ.0 is able to converge on the target network, as QUACQ (obviously) does. The interesting information is that T-QUACQ.0 does this in a total cpu time for generating all queries that is significantly lower than the time needed by QUACQ. QUACQ needs 46 minutes to converge on the instance of sudoku and 52 minutes to converge on the instance of graceful graphs, whereas T-QUACQ.0 converges in 1 minute or less on both instances.

Let us focus a bit more on how the two algorithms spend their time. Figure 1 reports the waiting time from one query to another needed by QUACQ and T-QUACQ.0 to learn $GG(K_5 \times P_2)$. We selected a fragment of 100 queries near to the end of the learning process for each algorithm. On the one hand, we see that T-QUACQ.0 never exceeds the bound of `time_bound` $= 1$ second between two queries, thanks to its TQ-GEN time bounded generator. On the other hand, we observe that in these 100 queries close to convergence, generating a query in QUACQ is time consuming because it requires solving a hard CSP. There are three negative queries (that is, queries followed by small queries of `FindScope` and `FindC`) requiring from 20 to 50 seconds to be generated, and many positive queries (that is, not followed by small queries) requiring from 20 to 200 seconds to be generated.

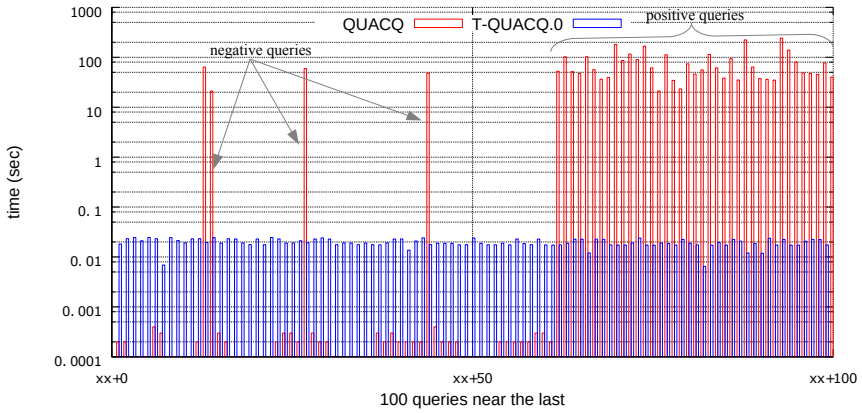

Fig. 1: Time to generate queries on $GG(K_5 \times P_2)$ (T-QUACQ.0 versus QUACQ).

Once the approach has been validated by this first experiment, we tried to understand the behavior of T-QUACQ.0 when pushing τ to the limits of the range $0..$`time_bound`. We instantiated τ to a very small value: $0.001ms$, and a very large value, close to `time_bound`: $900ms$. Results are reported in Table 1.

When τ takes the large value of $900ms$, T-QUACQ.0 fails to converge (the convergence rate is 94% in sudoku, and 41% in $GG(K_5 \times P_2)$). The explanation is that τ is so close to `time_bound` that if TQ-GEN fails to produce a query of size $|X|$ in $900ms$, there remains only $100ms$ to produce a query of size $\alpha \cdot |X|$. In case TQ-GEN cannot make it, T-QUACQ.0 returns premature convergence because the time limit has been reached.

When τ takes its small value $0.001ms$, T-QUACQ.0 fails to converge (the convergence rate is 1% on both instances). The explanation in this case is that τ is so small that the bias is not τ-good. That is, the solver at line 8 of TQ-GEN fails to terminate even for the smallest sub-problems of two variables. Thus, TQ-GEN will spend time looping through lines 8, 13, and 6 until reaching `time_bound`.

After having tried these extreme values for τ, let us now use Theorem 1 to theoretically determine the values of τ that guarantee convergence. (Remember that α is set to 0.5.) According to Theorem 1, τ must be less than $1/(19440+log_{0.5}(2/81)) = 0.05ms$ on sudoku and less than $1/(40,460+log_{0.5}(3/35)) = 0.0247ms$ on $GG(K_5 \times P_2)$. We launched an experiment with $\tau = 0.024$ms, which meets the theoretical bound for both sudoku and $GG(K_5 \times P_2)$. The results are reported in Table 1. T-QUACQ.0 converges on sudoku but returns premature convergence on $GG(K_5 \times P_2)$ with a convergence rate equal to 93%. On sudoku, the bias is τ-good when $\tau = 0.024$ms, so the two conditions for convergence of Theorem 1 are met. On $GG(K_5 \times P_2)$, $\tau = 0.024$ms is too small for ensuring τ-goodness because the bias contains ternary constraints. Thus, the first condition for convergence of Theorem 1 is violated and T-QUACQ.0 fails to converge.

Our last observation on Table 1 is related to the number of queries. We consider only the cases where the learning process has converged, that is, sudoku with $(\alpha,\tau) = (0.5,0.024)$ and $(\alpha,\tau) = (0.5,5)$, and $GG(K_5 \times P_2)$ with $(\alpha,\tau) = (0.5,5)$. We observe that T-QUACQ.0 respectively asks 1%, 4%, and 14% more queries than QUACQ.

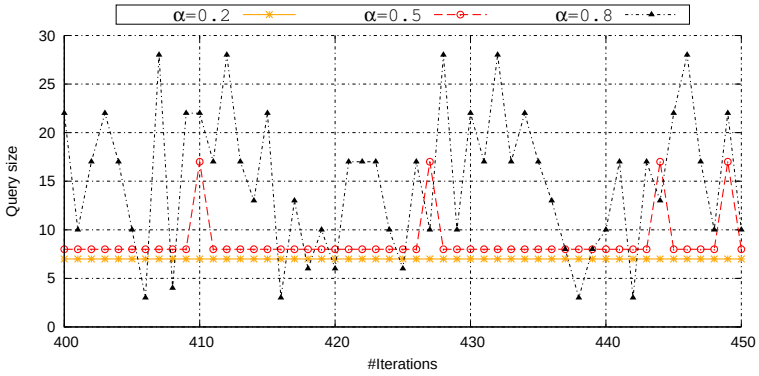

Fig. 2: Size of queries generated by TQ-GEN on $GG(K_5 \times P_2)$ with $\tau = 5ms$.

To understand why T-QUACQ.0 asks more queries than QUACQ on $GG(K_5 \times P_2)$, we launched T-QUACQ.0 with different values of α. (τ is kept fixed to $5ms$.) Interestingly, we observed that the number of queries varies significantly with α. For α taking values 0.2, 0.5, and 0.8, T-QUACQ.0 requires respectively, $6{,}654$, $5{,}610$, and 5000 queries to converge. We then measured the size of queries in T-QUACQ.0 with these three values of α. Figure 2 reports the size of queries asked by T-QUACQ.0 to learn $GG(K_5 \times P_2)$ with α equal to 0.2, 0.5, and 0.8. We make a zoom on the $400th$ to $450th$ iterations of T-QUACQ.0. We observe that the larger α, the greater the size of the query returned by TQ-GEN and the smaller the number of queries. When α is small, this often leads to queries of very small size. For $\alpha = 0.2$, all queries have size $\lfloor \alpha \cdot 35 \rfloor = 7$ (because $GG(K_5 \times P_2)$ has 35 variables). For $\alpha = 0.5$, a few queries have size $\lfloor \alpha \cdot 35 \rfloor = 17$ but almost all have size $\lfloor \alpha^2 \cdot 35 \rfloor = 8$. Generating queries of small size can be beneficial at the beginning of the learning process, when queries are often negative, because function `FindScope` will quickly find the right scope of the missing constraint inside a small subset Y. But at the end of the learning process, when most queries are positive, a short query leads to very few constraints removed from B in line 9 of Algorithm 2. Hence, convergence is slow in terms of number of queries. This is what happened in Table 1 on $GG(K_5 \times P_2)$ with $\alpha = 0.5$ and $\tau = 5ms$. These observations led us to propose more flexible ways to adjust the query size during the learning process.

6.3 Strategies and Settings

Following our first observations on our baseline version T-QUACQ.0, we expect that there is room for improvement by making the use of the query size ℓ less brute-force (reset to $|X|$ after each query generation in T-QUACQ.0). We propose here to adjust it in a more smooth way, to let T-QUACQ concentrate on the size of query that is the most beneficial at a given point of the learning process.

We propose the following $adjust$ function (see Algorithm 3). Given a query generated by TQ-GEN, if the answer is *yes*, $adjust$ increases ℓ by a factor α, and if the answer is *no*, $adjust$ decreases ℓ by a factor α. The intuition behind such adaptation

Algorithm 3: $adjust$ function of T-QUACQ.1

```
1 In ℓ, answer, InOut ℓ
2 if answer = yes then
3 |   ℓ ← min(⌈ℓ/α⌉, |X|);
4 else
5 |   ℓ ← ⌊α · ℓ⌋
6 return ℓ
```

of the query size is that when we are in a zone of many no answers (early learning stage), short negative queries lead to less queries needed by `FindScope` to find where the culprit constraint is, whereas in a zone of yes answers (late learning stage), larger positive queries lead to the removal of more constraints from B, and thus faster convergence. T-QUACQ using this version of the function $adjust$ is called T-QUACQ.1 in the following.

We expect that the efficiency of T-QUACQ.1 will depend on the initialization of the parameters α and τ in function $initialize$ (line 2 of Algorithm 2). Concerning the parameter ℓ, we observed that its initial value has negligible impact as it is used only once at the start of the learning process. We thus set function $initialize$ to always initialize ℓ to $|X|$.

Concerning α and τ, to find the most promising values of these parameters, we made an experiment on graceful graphs. On these problems, our base version T-QUACQ.0 performed worse than QUACQ. The results of T-QUACQ.1 are shown in sub-figures (a), (b), and (c) of Figure 3. The x-axis and the y-axis are respectively labeled by α ranging from 0.1 to 0.9, and $log_{10}(\tau)$ in μs (that is, each value of y corresponds to $10^y \mu s$) ranging from $10\mu s$ to $1s$. Darker color indicates higher number of queries. The number in each cell indicates the convergence rate $\%Conv$.

Let us first analyze the convergence rate of T-QUACQ.1. We observe the same results as already seen with our baseline version, that is, premature convergence when τ is too small ($10\mu s$) or very large ($1s$), When τ does not take extreme values, we observe convergence in many cases. The range of values of τ that lead to frequent convergence (in fact convergence for all values of α except 0.9) goes from $[1ms, 100ms]$ on the small instance to $[10ms, 100ms]$ on the larger. Concerning α, we observe that its value does not have any impact on convergence except the very large value 0.9, which leads T-QUACQ.1 to return premature convergence on the harder instances even for values of τ that give convergence with all other values of α. (See $\alpha = 0.9$ in sub-figures (b) and (c) of Figure 3.) This is explained by the fact that with such a large α, finding the right size ℓ of the query to generate can require too many tries (when τ is reached) and lead to exhaust the time bound

Let us now study the impact of τ and α on the number of queries asked by T-QUACQ.1. We restrict our analysis to the cases where T-QUACQ.1 has converged. We observe that when T-QUACQ.1 converges, the larger τ, the lower number of query (see sub-figures (a), (b), and (c) of Figure 3). Concerning the impact of α on the number of queries, we observe that, the greater α (except 0.9 which leads to premature convergence), the lower the number of queries. The reason is that a large α leads to a smooth adjust-

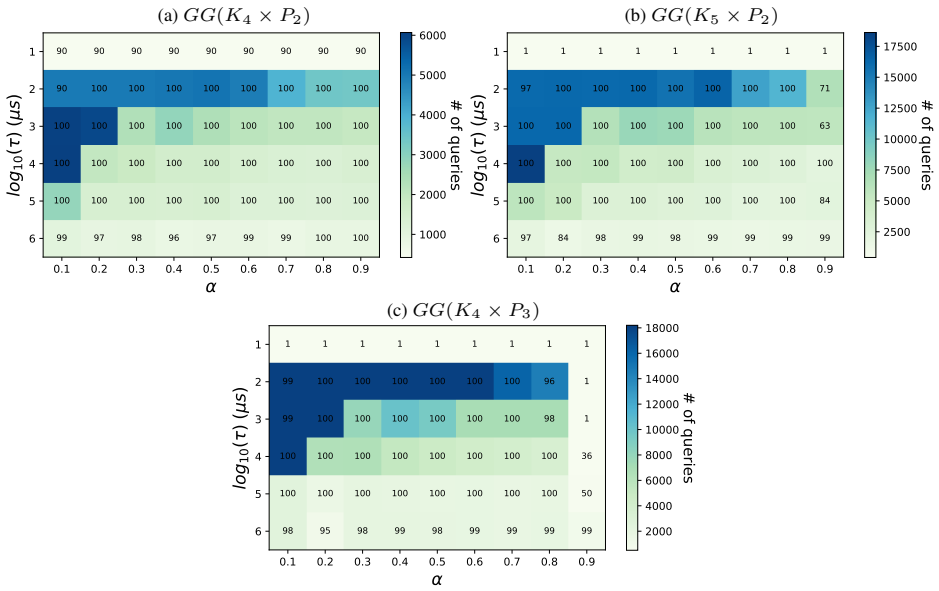

Fig. 3: Number of queries and convergence rate performed by T-QUACQ.1 on graceful graphs. Darker color in color bar indicates higher number of queries, and the number in each cell of the map indicates the convergence rate.

ment of the size of the queries, depending on the computing time allowed by τ and the positive/negative classification of previous examples. This especially has the effect that T-QUACQ.1 generates large queries at the end of the learning process, which lead to faster convergence, as seen with T-QUACQ.0. In the following we choose 0.8 as a default value for α.

To validate the observations made on graceful graphs, and in order to select the most promising value of τ, we extended our experimentation to Golomb rulers. Golomb rulers have the nice property that the basic model does not only contain binary constraints. It also contains ternary and quaternary constraints, which makes query generation more difficult. We used four instances of Golomb rulers of size $n = 8$, 12, 16, and 20. We set α to 0.8, and vary τ from 10μs to 1s. We added the value $\tau = 50ms$ (that is, $log_{10}(50ms) = 4.7$) inside the interval $[10ms, 100ms]$ as these values were looking the most promising for convergence in our previous experiment. The results of T-QUACQ.1 on those problems are shown in Figure 4, where the x-axis and the y-axis are respectively labeled by $log_{10}(\tau)$ in μs, and the problem size n.

We first analyze the impact of τ on the convergence rate. The results in Figure 4 show us that the larger the problem size, the greater the value of τ for convergence. We observe that T-QUACQ.1 converges for no instance at $\tau = 10\mu s$, 1 instance at $100\mu s$, 2 instances at $1ms$, and 3 instances from $50ms$ to $1s$. For $n = 20$, convergence is never reached, but $\tau = 10ms$ and $\tau = 50ms$ give the best results. If we combine these results with those obtained on graceful graphs, it leads us to the conclusion that the best value for τ is $50ms$. Let us now analyze the impact of τ on the number of

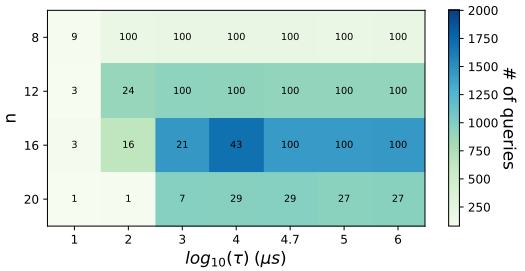

Fig. 4: Number of queries and convergence rate performed by T-QUACQ.1 on Golomb rulers.

queries when T-QUACQ.1 converges. We observe that for all the instances, the number of queries required for convergence is almost the same regardless of the value of τ. In the following we set τ to $50ms$.

We finally validate this optimized version of T-QUACQ.1 on other benchmark problems. Table 2 reports the results of QUACQ and of T-QUACQ.1 with the parameters $\alpha = 0.8$ and $\tau = 50ms$. $totT$ is the total time of the learning process, $MT(q)$ the maximum waiting time between two queries, and $\#q$ the total number of asked queries.

The first important observation is that T-QUACQ.1 has converged for all instances presented in Table 2. Second, what we saw with the baseline version T-QUACQ.0 remains true with T-QUACQ.1: Time to generate queries is short, almost always orders of magnitude shorter than with QUACQ. Finally, the good surprise comes from the number of queries. Compared to T-QUACQ.0, the number of queries in T-QUACQ.1 drops significantly thanks to the smooth adjustment of the size of the queries. The number of queries in T-QUACQ.1 is even smaller than the number of queries in QUACQ on all but two instances, despite QUACQ is free to use as much time as it needs to generate a query.

7 Conclusion

We have proposed TQ-GEN, a query generator that is able to generate a query in a bounded amount of time, and then to satisfy users tolerable waiting time. TQ-GEN is able to adjust the size of the query to generate so that the query can be generated within the time bound. We have also described T-QUACQ, a QUACQ-like algorithm that uses TQ-GEN to generate queries. Our theoretical analysis shows that the bounded waiting time between queries is at the risk of reaching a *premature convergence*. We have then proposed strategies to better adapt query size. Our experiments have shown that T-QUACQ combined with a good strategy dramatically improves QUACQ in terms of time needed to generate queries and also in number of queries, while still reaching convergence.

References

1. Arcangioli, R., Bessiere, C., Lazaar, N.: Multiple constraint acquisition. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016. pp.

Table 2: T-QUACQ.1 versus QUACQ, $\alpha = 0.8$, $\tau = 50ms$

Benchmark ($\|X\|$, $\|D\|$, $\|C\|$)	**Algorithm**	$totT$ (in $seconds$)	$MT(q)$ (in $seconds$)	$\#q$
Zebra (25, 5, 64)	QUACQ	1.29	0.13	706
	T-QUACQ.1	1.34	0.11	547
rand-50-10-12 (50, 10, 12)	QUACQ	204	5.01	253
	T-QUACQ.1	13	0.36	325
rand-50-10-122 (50, 10, 122)	QUACQ	88	1.68	1,217
	T-QUACQ.1	21	0.22	1,222
$GG(K_4 \times P_2)$ (24, 16, 164)	QUACQ	976	13	1,989
	T-QUACQ.1	47	0.39	1,273
$GG(K_5 \times P_2)$ (35, 25, 370)	QUACQ	3,144	512	4,898
	T-QUACQ.1	110	0.65	2,317
$GG(K_4 \times P_3)$ (38, 26, 417)	QUACQ	7,206	367	5,796
	T-QUACQ.1	150	0.89	2,883
Sudoku 9×9 (81, 9, 810)	QUACQ	2,810	1,355	9,053
	T-QUACQ.1	69	0.33	6,873
Latin-Square (100, 10, 900)	QUACQ	7,200	1,234	12,204
	T-QUACQ.1	120	0.56	7,711
Golomb-ruler-12 (12, 110, 2,270)	QUACQ	11,972	2,808	2,445
	T-QUACQ.1	1,184	0.94	916

698–704. New York, NY (2016)
2. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP 2012. Lecture Notes in Computer Science, vol. 7514, pp. 141–157. Springer, Québec City, QC, Canada (2012)
3. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Walsh, T.: Constraint acquisition via partial queries. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013. pp. 475–481. Beijing, China (2013)
4. Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M.: Query-driven constraint acquisition. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007. pp. 50–55. Hyderabad, India (2007)
5. Bessiere, C., Daoudi, A., Hebrard, E., Katsirelos, G., Lazaar, N., Mechqrane, Y., Narodytska, N., Quimper, C., Walsh, T.: New approaches to constraint acquisition. In: Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, Lecture Notes in Computer Science, vol. 10101, pp. 51–76. Springer (2016)
6. Bessiere, C., Lazaar, N., Koriche, F., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence p. In Press (2017)
7. Freuder, E.C., Wallace, R.J.: Suggestion strategies for constraint-based matchmaker agents. International Journal on Artificial Intelligence Tools 11(1), 3–18 (2002)
8. Jefferson, C., Akgun, O.: CSPLib: A problem library for constraints. `http://www.csplib.org` (1999)
9. Lallemand, C., Gronier, G.: Enhancing user experience during waiting time in hci: Contributions of cognitive psychology. In: Proceedings of the Designing Interactive Systems Conference. pp. 751–760. DIS '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2317956.2318069`
10. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010. pp. 45–52. Arras, France (2010)
11. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. In: Proceedings of the 9th International Conference Principles and Practice of Constraint Programming, CP 2003. Lecture Notes in Computer Science, vol. 2833, pp. 930–934. Springer, Kinsale, Ireland (2003)
12. Shchekotykhin, K.M., Friedrich, G.: Argumentation based constraint acquisition. In: Proceedings of the Ninth IEEE International Conference on Data Mining, ICDM 2009. pp. 476–482. Miami, FL (2009)