

Parallel Constraint Acquisition

Nadjib Lazaar

▶ **To cite this version:**

Nadjib Lazaar. Parallel Constraint Acquisition. AAAI 2021 - 35th Conference on Artificial Intelligence, Sep 2021, virtually, Canada. pp.3860-3867, 10.1609/aaai.v35i5.16504 . lirmm-03687351

HAL Id: lirmm-03687351
https://hal-lirmm.ccsd.cnrs.fr/lirmm-03687351

Submitted on 3 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI-21)

Parallel Constraint Acquisition

Nadjib Lazaar
LIRMM, University of Montpellier, CNRS, Montpellier, France
lazaar@lirmm.fr

Abstract

Constraint acquisition systems assist the non-expert user in modelling her problem as a constraint network. QUACQ is a sequential constraint acquisition algorithm that generates queries as (partial) examples to be classified as positive or negative. The drawbacks are that the user may need to answer a great number of such examples, within a significant waiting time between two examples, to learn all the constraints. In this paper, we propose PACQ, a portfolio-based parallel constraint acquisition system. The design of PACQ benefits from having several users sharing the same target problem. Moreover, each user is involved in a particular acquisition session, opened in parallel to improve the overall performance of the whole system. We prove the correctness of PACQ and we give an experimental evaluation that shows that our approach improves on QUACQ.

Introduction

Constraint programming (CP) has made considerable progress over the last forty years, becoming a powerful paradigm for modelling and solving combinatorial problems. Several parallel algorithms have been proposed to solve a problem as a constraint network and they are grouped under categories: distributed CSPs; parallel propagation; parallel search; portfolio algorithms; problem decomposition (Régin and Malapert 2018). However, modelling 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 (Freuder and Wallace 1998). 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 (Lallouet et al. 2010). Beldiceanu and Simonis proposed MODELSEEKER, a system devoted to problems with regular structures and based on the global constraint catalog (Beldiceanu and Simonis 2012). Bessiere et al. proposed CONACQ, which generates membership queries (i.e., complete examples) to be classified by the user (Bessiere et al. 2017). Shchekotykhin and Friedrich extended CONACQ to allow the user to provide *arguments* as constraints to speed-up the convergence (Shchekotykhin and Friedrich 2009).

Bessiere et al. proposed QUACQ (for Quick Acquisition), an active learning system that is able to ask the user to classify partial queries (Bessiere et al. 2020, 2013). 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 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. However, even that good theoretical bound can be hard to put in practice. Generating a complete example is NP-hard and the total number of examples to classify can be large. For instance, QUACQ can take more than 20 minutes to generate a complete example during the acquisition process of the Sudoku constraint network and it requires the user to classify more than $9K$ examples.

In this paper, we introduce a first ever approach to combine CP modelling through constraint acquisition with parallelism. We present PACQ, a portfolio-based parallel constraint acquisition system. PACQ learns constraint network by exchanging with several users in different acquisition sessions. Users have in mind the same target problem without knowing how to model it as a constraint network. PACQ benefits from the main limitations of the sequential QUACQ system: (i) large number of examples to classify by the user; and (ii) significant waiting time between two queries. We experimentally evaluate the benefit of using parallelism in constraint acquisition on several problems. The results show that the number of queries increases under an upper-bound using PACQ and that PACQ dramatically improves the sequential version of QUACQ in terms of queries asked per user and in terms of CPU time needed to generate queries.

Background

The constraint acquisition process can be seen as an interplay between the user and the learner. In our context, we have N users (U_1 to U_N) involved in N different acquisition sessions (A_1 to A_N) with one user per session and under a memory-shared model. The learner and the users need to share some common knowledge to communicate.

We suppose this common knowledge, called the *vocabulary*, is a tuple 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_S is defined by the sequence of variables S, a sub-sequence of X (i.e. $S \preceq X$), called *the constraint scope*, and the relation c over $\mathbb{Z}$ specifying which sequences of $|S|$ values are allowed for the variables S. A *constraint network* is a set C of constraints on the vocabulary (X, D). An assignment $e_Y \in D^Y$, where $D^Y = \Pi_{x_i \in Y} D(x_i)$, is called a partial assignment when $Y \prec X$ and a complete assignment when $Y = X$. An assignment e_Y on a set of variables $Y \preceq X$ *is rejected by* a constraint c_S (or e_Y *violates* c_S) if $S \preceq Y$ and the projection $e_Y[S]$ of e_Y on the variables in S is not in c. If e_Y does not violate c_S, it *satisfies* it. An assignment e_Y on Y *is accepted by* C if and only if it satisfies all constraint in C. An assignment on X that is accepted by C is a *solution* of C. We write $sol(C)$ for the set of solutions of C, and $C[Y]$ for the set of constraints from C whose scope is included in Y.

In addition to the vocabulary, the learner owns a *language* Γ of bounded arity relations from which it can build constraints on specified sets of variables. Adapting terms from machine learning, the *constraint basis*, denoted by B, is a set of constraints built from the language Γ on the vocabulary (X, D) from which the learner builds a constraint network.

Given a prefixed vocabulary (X, D), a *concept* is a Boolean function f over D^X, that is, a map that assigns to each assignement e a value in $\{0, 1\}$. A *representation* of a concept f is a constraint network C for which $f^{-1}(1) = sol(C)$. A *target concept* is a concept f_T that returns 1 for e if and only if e is a solution of the problem that the N users share and have in mind. The *target network* is a network T such that $T \subseteq B$ and T is a representation of f_T. Then we say that the target concept f_T is representable by B.

A *membership query* $ASK_{U_i}(e)$ takes as input a *complete* assignment e in D^X and asks the user U_i to classify it. The answer to $ASK_{U_i}(e)$ is *yes* if and only if $e \in sol(T)$. A *partial query* $ASK_{U_i}(e_Y)$, with $Y \subseteq X$, takes as input a *partial* assignment e_Y in D^Y and asks the user U_i to classify it. The answer to $ASK_{U_i}(e_Y)$ is *yes* if and only if e_Y does not violate any constraint in T. It is important to observe that "$ASK_{U_i}(e_Y)$=*yes* " does not mean that e_Y extends to a solution of T, which would put an NP-complete problem on the shoulders of the user. For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y. A classified assignment e_Y is called positive or negative *example* depending on whether $ASK_{U_i}(e_Y)$ is *yes* or *no*. Knowing that (i) any extension of a negative example is a negative example and any shortening of a positive example is a positive example; and (ii) under a memory-shared model: the ASK function checks first if the query is not a redundant one w.r.t. another acquisition session where the classification can be deduced.

We now define *convergence*, which is the constraint acquisition problem we are interested in. Given a set E of (partial) examples labelled by the user *yes* or *no*, we say that a network C agrees with E if C accepts all examples labelled *yes* in E and does not accept those labelled *no*. The learning process has *converged* on the network $L \subseteq B$ if **(i)** L agrees with E and **(ii)** for every other network $L' \subseteq B$ agreeing with E, we have $sol(L') = sol(L)$. We are thus guaranteed that $sol(L) = sol(T)$. We say that the learning process reached a *premature convergence* if only **(i)** is guaranteed.

PACQ: Portfolio-Based Parallel Constraint Acquisition

We propose PACQ, a portfolio-based parallel constraint acquisition system. PACQ takes as input a basis B on a vocabulary (X, D) shared with N users. It asks (partial) queries of the N users until it has converged on a constraint network L equivalent to the target network T. The rationale behind PACQ is to learn a constraint network using N parallel acquisition sessions. That is, we have a portfolio of acquisition sessions aiming to acquire simultaneously different parts of the problem using a shared-memory model.

Description of PACQ

PACQ (see Algorithm 1) shares between acquisition sessions the basis B and the network L (line 2). PACQ initializes the network L it will learn to empty set (line 3). At line 4, PACQ makes a set-partition of the basis B into N subsets $(B_1 \ldots B_N)$ using `split` function (i.e., $B_i \neq \varnothing$, $B_i \cap B_j = \varnothing, \forall i, j$ and $\bigcup_{i=1}^{N} B_i = B$). We will see later that there are multiple ways to design the `split` function. Then, PACQ opens in parallel N `Acq_session` (line 6) and it converges on L once all sessions closed (line 7).

`Acq_session` starts by calling `GenerateExample` function that generates an example e on X satisfying the constraints of L, but violating at least one constraint from B_i (line 2). Bear in mind that, from a session to another, generating an example on a different B_i allows us to have sessions with different and complementary viewpoints on the acquisition process as a whole. If there does not exist any example e accepted by L and rejected by B_i, then all constraints in B_i are implied by L and we can safely remove them from B (line 3). Then, the current session is closed (line 3). If an example e is returned by `GenerateExample`, e is classified as positive or negative by the user U_i (line 4). If the answer is *yes*, we can remove from B the set $\kappa_B(e)$ (line 5). If the answer is *no*, we are sure that e violates at least one constraint of the target network T. We then call the function `FindScope` to discover the scope scp of these violated constraints (line 7), and the procedure `FindC` will learn (that is, put in L) at least one constraint of T whose scope is in scp (line 9). Function `FindScope` and procedure `FindC` ask queries to the corresponding user U_i and they are used exactly as they appear in, respectively, (Bessiere et al. 2013) and (Bessiere et al. 2020). The unique portion of the algorithm that cannot be parallelized is the call of `FindC` within connected scopes. Here, the procedure `FindC` has a unique permit access. For instance, if two `Acq_session` A_i and A_j return simultaneously at line 7 the scopes scp_1 and scp_2, such that $scp_1 \cap scp_2 \neq \varnothing$ (i.e., connected scopes), only one session acquires an access to `FindC` on scp_1 or scp_2 (line 8) and the second one must wait for its release at line 10. In case we have two sessions looking simultaneously for constraints on scp, only one session will have access to `FindC`

whereafter $\kappa_B(e[scp]) = \varnothing$.

Algorithm 1: PACQ

```
   In  : A basis B, Number of Users N
   Out : A learned network L
1  begin
2      shared B; L;
3      L ← ∅;
4      split (B, N);        // split B into N parts
5      foreach i ∈ 1..N do in parallel
6          Acq_session (U_i);
7      return "convergence on L"
```

Theoretical Analysis

We first show that a parallel acquisition using PACQ (algorithm 1) is a correct algorithm to learn a constraint network representing the target problem over B. We prove that PACQ is sound, complete, and it terminates.

Proposition 1 (Soundness) *Given a basis B, a target network $T \subseteq B$ and N users, the network L returned by* PACQ *is such that $sol(T) \subseteq sol(L)$.*

Proof. Suppose there exists $e \in sol(T) \setminus sol(L)$. Hence, there exists at least a constraint $c_Y \in L$ rejecting e and learned by PACQ within a `Acq_session` A_i of user U_i. The only place where we can add c_Y to L is (algo:2-line:9) with `FindC` on Y scope that is returned by `FindScope` at (algo:2-line:7). `FindC` represents the portion of PACQ that is not parallelized and the access is conditioned by the fact that no previous call occurred on Y (i.e., $\kappa_B(e[Y]) \neq \varnothing$). We know from (Bessiere et al. 2013, 2020) that `FindScope` and `FindC` functions are sound. The learned constraint c_Y is one of the target network T. Therefore, adding a constraint to L cannot reject a tuple accepted by T. □

Proposition 2 (Completeness) *Given a basis B, a target network $T \subseteq B$ and N users, the network L returned by* PACQ *is such that $sol(L) \subseteq sol(T)$.*

Proof. Suppose there exists $e \in sol(L) \setminus sol(T)$ when PACQ terminates. Hence, there exists a constraint c_Y from B that rejects e. Knowing that at (algo:1-line:4) we have a set-partition of B, it exists a `Acq_session` A_i where $c_Y \in B_i$. The only way for PACQ to terminate is to have all `Acq_session` closed. This means that within A_i session and at (algo:2-line:2), `GenerateExample` was not able to generate an example e' accepted by L and rejected by B_i. c_Y was in B_i before starting PACQ ($c_Y \in T$) and it is not in B_i when PACQ terminates. Constraints can be removed in `FindC`/`FindScope` functions and at (algo:2-line:3 and 5). We know from (Bessiere et al. 2013, 2020) that `FindC`/`FindScope` cannot remove a constraint that rejects an example accepted by L. A constraint c removed from (algo:2-line:3 and 5) cannot be c_Y because e violtates c_Y and is accepted by L. Therefore, c_Y cannot reject an example accepted by L, which proves that $sol(L) \subseteq sol(T)$. □

Proposition 3 (Termination) *Given a basis B, a target network $T \subseteq B$ and N users,* PACQ *terminates.*

Proof. The termination of PACQ immediately follows the closure of the N `Acq_session`. Let us consider a given A_i session. An example is generated such that it satisfies L and violates B_i. If no such example exists, B_i is reduced to empty and A_i session is closed. Otherwise, `GenerateExample` at (algo:2-line:2) returns an example e to submit to the user U_i. B decreases in size by removing $\kappa_B(e)$ when user U_i says *yes* (algo:2-line:5). If the user U_i says *no*, B also decreases in size by learning at least one constraint from B (algo:2-line:9). Let us suppose now that the user U_i says *no* on e because of a unique constraint to learn c_Y that is rejected by e. Suppose that the same example is generated at the same time in $k-1$ sessions. Then Y scope is returned by the `FindScope` calls in the k `Acq_session`. We know that for connected scopes, `FindC` has a single permit access. Thus, we have only one user U_j calling `FindC` $_{U_j}$, adding c_Y to L and removing c_Y from B. Afterwards, the other $k-1$ sessions will not have access to `FindC` on Y as $\kappa_B(e[Y])$ is reduced to empty after the first call. Therefore, at each execution of the loop, we have at least one B_i that strictly decreases in size. As B_i represent finite-size subsets coming from a set-partition of B, we have termination. □

Theorem 1 (Correctness) *Given a basis B, a target network $T \subseteq B$ and N users,* PACQ *returns a network L such that $sol(L) = sol(T)$.*

Proof. Correctness immediately follows from Propositions 1, 2, and 3. □

Algorithm 2: `Acq_session` (U_i)

```
1  while true do
2      e ← GenerateExample (L, B_i);
3      if e = nil then B ← B \ B_i;   break ;
4      if ASK_{U_i}(e) = yes then
5          B ← B \ κ_B(e);
6      else
7          scp ← FindScope_{U_i}(e, ∅, X);
8          acquire(scp);
9          if κ_B(e[scp]) ≠ ∅ then FindC_{U_i}(e, scp, L);
10         release(scp);
```

Strategies and Settings

PACQ can be improved by making the use of `GenerateExample` and `split` functions less brute-force, and by adapting it to a particular context (e.g., distributed CSPs).

`GenerateExample`. We can speed up the example generation by using well-known variable heuristic selectors (e.g., `minDom`, `domOverWdeg`, `impact`,...), or by using a dedicated one like `bdeg` heuristic (Tsouros and Stergiou 2020). `bdeg` selects the variable involved in a maximum number of constraints present in $B_i \setminus L$. Knowing that each session is reasoning on a particular B_i and based on preliminary comparisons, `bdeg` heuristic provides a good diversification.

`split`. In our study, we have investigated five set-partitions of B based on different background knowledge:
• `Scope`: put in the same B_i all constraints of a given scope;
• `Negation`: put in B_i a constraint and its negation;
• `Language`: put in B_i constraints of the same relation;
• `Graph`: B_i's are connected components;
• `Rule`: put in B_i constraints satisfying a set of rules.

The preliminary tests show that a B partition using background knowledge boosts the acquisition process and that the same findings are observed with the five set-partitions. We focus our analysis on the `Rule` based set-partition.

The `split` function based on `Rule` groups in the same B_i's constraints satisfying a set of rules adapted to constraint acquisition. For instance, if we know that $c_1 \wedge c_2 \rightarrow c_3$, then putting the three constraints c_1, c_2, c_3 in the same B_i can speed up the generation of examples. Building a complete set of rules is often too expensive, both in time and space as it requires generating a set of rules potentially exponential in space (all combinations of constraints that imply another one). However, it is possible to compute approximations by bounding the number of constraints in the body of a rule. Here, we only considered the rule that contain two constraints in the body `rule` : $c_i \wedge c_j \rightarrow c_k$. That is, `split` performs a random partition of triplets $(c_i, c_j, c_k) \in B^3$ where (c_i, c_j, c_k) satisfies `rule`. The rationale behind `Rule` partition is to group in the same session a certain percentage of redundancies that B contains. " Doing so, (i) we facilitate the task of `GenerateExample` to find an assignment satisfying L and rejecting at least one constraint in B, and (ii) avoiding parallel sessions to learn redundancies.

PACQ for Distributed CSPs. In some cases, parallel acquisition can be subject to privacy and/or security requirements with information that should not be shared between sessions. PACQ can easily be adapted to act in a distributed context by (i) taking into account the visibility of each agent (i.e., set of variables) in the `split` function (algo:1-line:4); and (ii) for a given session, generating examples on its own learned network L_i (algo:2-line:2).

PACQ versions. With the different strategies and settings in hand, we evaluate the four following versions of PACQ:
• **PACQ.0** using a *random* variable ordering and a *random* set-partition of B into N subsets.
• **PACQ.1** using `bdeg` heuristic;
• **PACQ.2** using `bdeg` heuristic and `Rule` based `split`;
• **PACQ.3** a revised version of PACQ.2 for distributed CSPs.

Experimental Evaluation

In this section, we experimentally evaluate our portfolio-based parallel constraint acquisition system. As finding an assignment satisfying the constraints of L and violating at least one constraint from B is an NP-complete problem, we use a time limit, denoted by `TL`, once reached, the acquisition process returns a *premature convergence* on L. The only parameter we will keep fixed in all our experiments is `TL`, that we set to 5 seconds as it corresponds to an acceptable waiting time for a human user (Lallemand and Gronier 2012). The implementation of PACQ were carried out in Java using `Choco solver 4.10.2`.[1] The code is publicly available at (gite.lirmm.fr/constraint-acquisition-team). All tests were conducted on an HPC node of 28 CPU cores and 128Gb of RAM. Each core is an Intel(R) Xeon(R) CPU E5-2640 v4 @2.40GHz. Our evaluation aims to answer the following five research questions:

- **RQ1**: *How effective is an acquisition in a parallel configuration?*
- **RQ2**: *How to make* PACQ *more effective?*
- **RQ3**: *Is* PACQ *achieving a good level of load balancing between sessions?*
- **RQ4**: *How does* PACQ *scale with the number of sessions?*
- **RQ5**: *How effective is* PACQ *on distributed CSPs?*

Benchmark Problems

Random. We generated binary random target networks with 50 variables, domains of size 10, and 122 binary arithmetic constraints, denoted by `rand_122`. PACQ is initialized with the basis B containing the complete graph of $12,250$ binary arithmetic constraints.

Purdey. The problem has a single solution. Four families have stopped by Purdey's general store, each to buy a different item and paying differently. The problem is how can we match each family with the item they bought and how they paid for it. The target network has 12 variables with domains of size 4 and 27 binary arithmetic constraints. We initialized PACQ with a basis of 950 binary constraints.

Zebra. Lewis Carroll's zebra problem has a single solution. The target network is formulated using 25 variables of 5 values with 5 cliques of $\neq$ constraints and 14 additional constraints given in the description of the problem. PACQ is initialized with a basis B of $4,950$ unary and binary (arithmetic and distance) constraints.

Queens. (prob054 in CSPLib[2]) The problem is to place n queens on an $n \times n$ chessboard such that the placement of no queen constitutes an attack on any other. The target network is formulated using n variables of n values and $3n*(n-1)/2$ binary constraints with 3 constraints between each pair of variables ($\neq$, `out_diag1` and `out_diag2`). We take the instance of 30 queens. PACQ is initialized with a basis B of $4,350$ binary constraints.

Sudoku. The Sudoku logic puzzle is a 9×9 grid. It must be filled in such a way that all the rows, all the columns and the 9 non overlapping 3×3 squares contain the numbers 1 to 9. We run experiments also on a variant of Sudoku problem, the Jigsaw Sudoku (`jsudoku`) displayed in figure 1. Instead of having 3×3 squares, we have irregular shapes. The two target networks of `sudoku` and `jsudoku` have 81 variables of 9 values and, respectively, 810 and 811

[1] github.com/chocoteam/choco-solver
[2] www.csplib.org/Problems/prob054/

binary $\neq$ constraints on rows, columns and shapes. PACQ is initialized with B of $19,440$ binary arithmetic constraints.

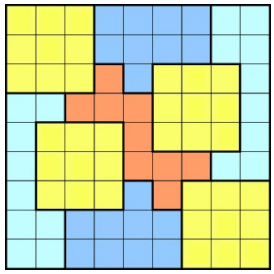


Figure 1: Jigsaw Sudoku logic puzzle instance.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. For this problem, we use 100 variables with domains of size 10 and 900 binary $\neq$ constraints on rows and columns. PACQ is initialized with $|B| = 29,700$ constraints.

Meeting Scheduling. (prob046 in CSPLib[3]) The Meeting Scheduling problem (`msp`) consists of n meetings and m attendees. Each meeting is of a given duration and location, and has a set of attendees. Thus, each attendee has a set of meetings that must attend. We take the instances of 40 meetings, presented in the CSPLib (`msp_19` to `msp_27`). The target networks contain from 75 to 125 constraints (attendee and time-arrival constraints). PACQ is initialized with $|B| = 4,680$ constraints.

Results

Table 1 reports the performance of PACQ.0 averaged over ten runs on each instance. We report the number of users $\#\mathtt{U}$; the size of the learned network $|L|$; the total number of asked queries $\#\mathtt{Q}$ in all sessions; the averaged, min and max number of queries asked per session ($\#\mathtt{q}, \mathtt{min}, \mathtt{max}$); time $\mathtt{T_A}$ needed to learn `L`; time of the learning process until convergence $\mathtt{T_C}$; the acquisition rate $\%\mathtt{A} = |T| - |L'|/|T|$, where L' are the constraints that have to be added to L to make it equivalent to T; and the convergence rate $\%\mathtt{C} = (\mathtt{B_{init}} - \mathtt{B_{final}})/\mathtt{B_{init}}$, where $\mathtt{B_{init}}$ and $\mathtt{B_{final}}$ are, respectively, the initial and the final size of the the basis B. $\mathtt{T_A}$ and $\mathtt{T_C}$ are CPU times of the last closed session. That is, we report the maximum ($\mathtt{T_A}, \mathtt{T_C}$) times needed for a given session. We report results with 1, $n/10$ and n users, where n corresponds to the number of variables of the given instance. We denote by PACQ.x$_{[n]}$ the call of PACQ with n users. Note that PACQ.0$_{[1]}$ is equivalent to a sequential acquisition under QUACQ.

[RQ1]: PACQ.0 **effectiveness.** From table 1, we observe that parallel acquisition under PACQ reduces the number of asked queries per user $\#\mathtt{q}$ and increases the total number of queries $\#\mathtt{Q}$. The number of queries per user $\#\mathtt{q}$ is reduced by a factor ranging between 2 and 60. However, the total number of queries $\#\mathtt{Q}$ is increased by a factor ranging between 2 and 4, which is far below the theoretical bound represented by the factor $\#\mathtt{U}$ (Amdahl's Law (Amdahl 1967)). For instance, on `latin` QUACQ asks more than 11K queries to a unique user, where PACQ.0$_{[100]}$ asks 192 queries per user (reduction factor of 60) with a total number of queries of 19K (growth factor of 2). Two reasons explain the growth of $\#\mathtt{Q}$. The first reason is related to the cost of learning a constraint (i.e., learning-ratio) that increases in a parallel configuration. For instance, we need 5 queries on average to learn a constraint of `queens` using QUACQ, where we need 12 queries under PACQ.0$_{[30]}$. The increase of the learning-ratio is due to the fact that several sessions can visit simultaneously the same part of the search space looking for the same constraint to learn where only one session succeeds at the end. The second reason that explains the growth of $\#\mathtt{Q}$ is learning implied constraints. If $c_1 \wedge c_2 \Rightarrow c_3$, parallel sessions can learn c_1, c_2 and the implied constraint c_3, where in a sequential configuration, c_3 can be removed. For instance, QUACQ converges on `zebra` with a constraint network of size 50. Within 25 parallel sessions, we learn a constraint network of size 70 including 20 implied constraints.

The second observation that we can draw is that a parallel acquisition speeds up the convergence. For instance, QUACQ is not able to learn the whole constraint network of the `queens` instance and it returns a premature convergence state. Here, generating $e \in sol(L \wedge \neg B)$ is hard enough that it requires more than a `TL` of $5s$. Whereas, PACQ.0$_{[3]}$ converges in $6.15s$ with B split between 3 sessions, which makes example generation much easier. Let us take another example with `sudoku` instance. QUACQ asks more than $8K$ queries to learn 76% of the target network. Than it reaches a state where finding an example in $sol(L \wedge \neg B)$ is too hard to be returned in less than $5s$. Within 8 sessions, PACQ.0 asks $1,378$ queries per user to reach an acquisition rate of 94% (*min, max* sessions of, resp., $1,123$ and $2,043$ queries). Whereas, PACQ.0$_{[81]}$ asks less than 200 queries per user to learn the whole target network and reaches a convergence rate of 99% ($[min, max] = [107, 346]$).

[RQ2]: Strategies and Settings. Table 2 reports the performance of PACQ.0, 1 and 2 averaged on ten runs on each instance. As table 1, we report the size of the learned network $|L|$; total number of queries $\#\mathtt{Q}$; queries per user, min and max ($\#\mathtt{q}, \mathtt{min}, \mathtt{max}$); acquisition time $\mathtt{T_A}$; convergence time $\mathtt{T_C}$; acquisition rate $\%\mathtt{A}$; convergence rate $\%\mathtt{C}$. We report results of $10n$ users, where n is the number of variables of the given instance.
In terms of queries, we observe a slight difference between the three versions. However, PACQ.1 and 2 outperform the basic setting of PACQ in terms of time and convergence with the use of a dedicated heuristic `bdeg` and the `Rule` split. For instance, we need 11 seconds to acquire the `jsudoku` instance, where PACQ.1 and 2 acquire it in one second. On the same instance, PACQ.0 reaches ($\%\mathtt{C} = 98\%$), where PACQ.1 and PACQ.2 slightly improve it by both reaching 99%.

[RQ3]: Workload Balancing. From table 1 and 2, we observe that PACQ.2 provides well-balanced sessions in terms of queries with values close to the mean comparing

[3] www.csplib.org/Problems/prob046/

#U	$\|L\|$	#Q	(#q, min, max)	(T_A, T_C)	%A	%C
rand_122:						
1	116	1K	1K	(4, 9)	100	100
5	116	4K	(898, 781, 973)	(4, 5)	100	100
50	117	4K	(90, 18, 475)	(0, 2)	100	100
purdey:						
1	17	159	159	(0, 0)	100	100
2	15	154	(77, 70, 84)	(0, 0)	100	100
12	30	372	(31, 20, 44)	(0, 0)	100	100
zebra:						
1	50	601	601	(1, 1)	100	100
3	61	852	(284,276 , 340)	(0, 0)	100	100
25	70	1K	(53, 41, 108)	(0, 0)	100	100
queens:						
1	1295	6K	6K	(-, -)	99	99
3	1305	8K	(2K, 2K, 2K)	(4, 6)	100	100
30	1305	16K	(534, 470, 577)	(2, 5)	100	100
sudoku:						
1	621	8K	8K	(-, -)	76	96
8	769	11K	(1K, 1K, 2K)	(-, -)	94	99
81	801	15K	(195,100 , 346)	(2, -)	100	99
jsudoku:						
1	586	7K	7K	(-, -)	72	95
8	746	9K	(1K, 1K, 1K)	(-, -)	92	98
81	791	12K	(155, 107 , 188)	(10, -)	100	99
latin:						
1	818	11K	11K	(-, -)	90	98
10	879	13K	(1K, 820, 1K)	(-, -)	97	99
100	898	19K	(192, 65, 1K)	(10, -)	100	99
msp_27:						
1	92	1K	1K	(-, -)	36	63
4	134	1K	(390, 202, 410)	(-, -)	53	77
40	223	2K	(71, 20, 79)	(-, -)	89	89

Table 1: PACQ.0 results

P.	$\|L\|$	#Q	(#q, min, max)	(T_A, T_C)	%A	%C
rand_122:						
0	116	9K	(18, 8, 115)	(0, 1)	100	100
1	116	8K	(17, 9, 117)	(0, 0)	100	100
2	115	9K	(18, 8, 114)	(0, 0)	100	100
purdey:						
0	41	1K	(9,4, 25)	(0, 0)	100	100
1	48	1K	(10, 4, 21)	(0, 0)	100	100
2	47	1K	(11, 4, 20)	(0, 0)	100	100
zebra:						
0	94	3K	(14, 5, 41)	(0, 2)	100	100
1	88	3K	(14, 5, 27)	(0, 0)	100	100
2	88	3K	(14, 5, 27)	(0, 0)	100	100
queens:						
0	1305	61K	(204, 23, 275)	(9, 9)	100	100
1	1305	63K	(210, 144, 272)	(7, 8)	100	100
2	1305	61K	(206, 146, 294)	(5, 8)	100	100
sudoku:						
0	798	18K	(23, 6, 143)	(10, -)	100	98
1	805	22K	(28, 6, 142)	(5, -)	100	99
2	806	21K	(27, 6, 67)	(3, -)	100	99
jsudoku:						
0	794	15K	(19, 9, 194)	(11, -)	100	98
1	817	22K	(28, 9, 182)	(1, -)	100	99
2	816	22K	(28, 9, 62)	(1, -)	100	99
latin:						
0	899	23K	(23, 7, 326)	(4, -)	100	98
1	900	27K	(27, 7, 254)	(1, -)	100	99
2	900	29K	(29, 7, 66)	(1, -)	100	99
msp_27:						
0	197	5K	(13, 5, 76)	(0, -)	78	97
1	168	10K	(27, 5, 69)	(0, -)	67	99
2	170	11K	(29, 5, 43)	(0, -)	67	99

Table 2: PACQ (P.) 0, 1 and 2 results with $10n$ users.

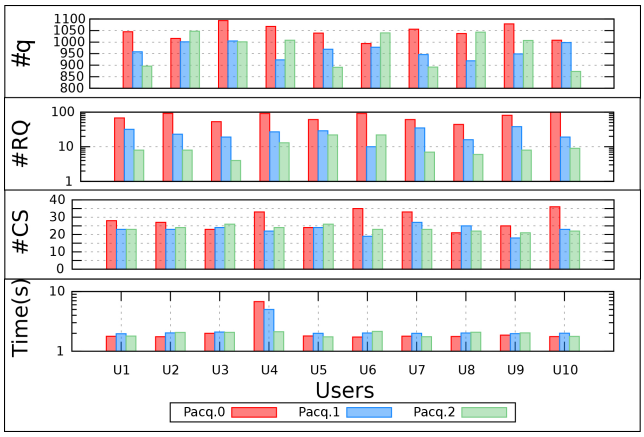


Figure 2: PACQ acting on 30-queens with 10 users

to PACQ.0 and 1. Thanks to the dedicated background knowledge partition based on Rule making sessions homogeneous comparing to a random partition used in PACQ.0 and 1. Let us take a closer look at latin instance. PACQ.0 and 1 with 1K users asks less than 30 queries per user with a minimum session of 7 queries and $\approx$ 10 sessions exceeding 200 queries. However, PACQ.2 asks queries per user in a tight interval of $[7, 66]$ with a standard deviation of 13.

In order to strengthen our previous observations, we run PACQ.0, 1 and 2 on queens with 10 sessions A_1 to A_{10}. For each session, we report in figure 2 the number of queries (#q), the number of redundant queries (#RQ), the number of connected scopes (#CS) and time in seconds of each session A_i. Note that RQ and CS allow us to estimate the degree of diversification between the different sessions.

The instance of queens has a target network of $1,305$ binary constraints. For the three versions, the averaged number of constraints acquired by each session is 130 with a standard deviation of 15 constraints for PACQ.0 and less than 8 constraints for PACQ.1 and 2. That is, PACQ provides a well-balanced workload of sessions. It follows that the same observation can be made on #Q, #RQ and #CS. Comparing the three versions, we observe that PACQ.1 (using bdeg heuristic) outperforms PACQ.0 by reducing the number of redundant queries to 50% and the number of connected scopes to 90%. We also observe that using Rule based split in PACQ.2 further improves the performance. In terms of CPU time, we observe an overload for A_4 session under PACQ.0 and PACQ.1. That is, the acquisition process terminates after

the close of session A_4 ($7s$ for PACQ.0 and $7s$ for PACQ.1). Whereas, PACQ.2 is ensuring an excellent level of load balancing with sessions of $\approx 1s$.

[RQ4]: Scalability. From table 1, we selected the three instances where QUACQ needs to ask more than $7K$ queries to acquire the corresponding target network (i.e., `sudoku`, `jsudoku` and `latin`). We run PACQ.2 on the three instances by varying the number of users up to $1K$. Figure 3 reports the total number of queries (#Q), the number of queries asked per user (#q), the learning-ratio R (i.e., the number of queries needed to learn a constraint $\mathtt{R} = \#\mathtt{Q}/|\mathtt{T}|$), the number of redundant queries (#RQ) and the number of connected scopes (#CS). Figure 3 shows that when the number of users grows, #Q follows an $(a - bx^{-c})$ scale with $a = 25K$, $b = 18K$ and $c = 0.74$, which means that when we get more and more users, the total number of queries gets closer and closer to the bound $a = 25K$, which is far below the Amdahl's Law theoretical bound (Amdahl 1967) represented by number of queries asked by QUACQ ($> 8K$) multiplied by #U. The second observation is that when the number of users grows, the number of queries asked per user #q follows a negative power function scale $(a\ x^{-b})$ with $a = 11K$ and $b = 0.85$, which means that when we get more and more users, number of queries asked per user gets closer and closer to 0 (horizontal asymptotes of negative power functions). This is very good news as it means that learning problems in parallel will scale well. For instance, QUACQ asks more than $8K$ to one user to learn `sudoku` instance. PACQ with, respectively, 2, 10, 100 and 1000 parallel sessions, asks to the same user, respectively, $3K$, $1K$, 163 and 13. Also, the learning-ratio R scales well when the number of users grows. The number of queries asked to learn one constraint follows a logarithmic scale bounded above by 35 queries per constraint ($c\ log(x) + d$ with $c = 3$ and $d = 8$). Following the conclusions drawn in **RQ3** and thanks to `bdeg` heuristic and `Rule` based `split`, we observe that #RQ and #CS per user decreases when the number of users grows. That is, PACQ.2 ensures an excellent level of workload balancing between sessions up to $1K$.

[RQ5]: PACQ for distributed CSP. For our last experiment, we illustrate the use of PACQ on distributed CSP with `msp` problem. In `msp`, each attendee comes with her own constraints and shares meetings (i.e., variables) with the other attendees. This means that the problem can be acquired in a fully-distributed scheme by having a session per attendee. Figure 4 reports %A and %C rates performed by QUACQ, PACQ.2$_{[v]}$ and PACQ.3$_{[v]}$ (where $v \in \{9, 13, 14, 17\}$ is the number of attendees) on the 9 `msp` instances. Darker color indicates higher convergence rate. The number in each cell indicates the acquisition rate. The sequential version using QUACQ is not able to learn and to converge on the 9 instances. PACQ.2 is not converging. However, PACQ.3 learns and converges on the 9 instances. For instance, on `msp_21` QUACQ learns 58% of the target network and returns a premature convergence of 62% in 10 seconds. PACQ.2 learns the instance without converging (%C = 84%) in $8s$. Then, the distributed version with PACQ.3 converges

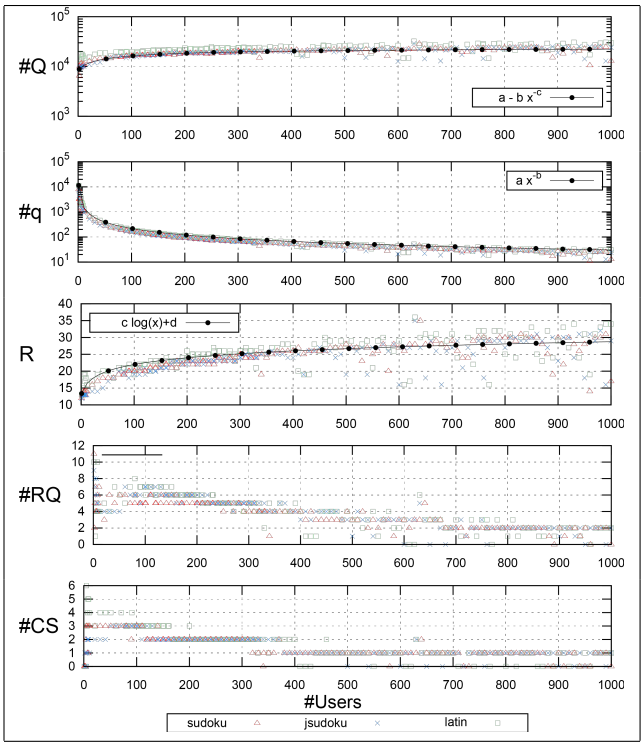


Figure 3: Scalability of PACQ.2 up to $1K$ users

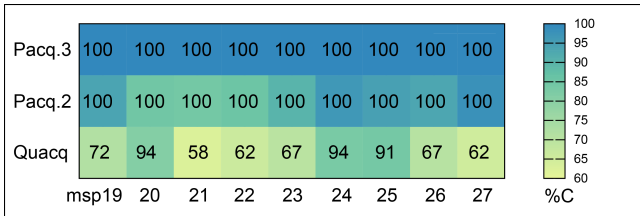


Figure 4: %A and %C comparison (QUACQ, PACQ.2 and 3)

in $0.38s$. This is explained by the fact that in PACQ.3 we have sessions of small size in terms of variables. Again on `msp_21`, PACQ.3 opens 13 sessions of 5 variables, where PACQ.2 opens 13 sessions of 40 variables. That is, generating examples on 5 variables is easier than generating examples on 40 variables and thus, avoid premature convergence.

Conclusion

In this paper we proposed a parallel constraint acquisition system PACQ, where numerous users answer in parallel queries in order to learn all the constraints. PACQ is a parallel extension of QUACQ and preserves the fundamentals of its active learning system and its soudness, correctness and completeness properties. Performed experiments showed that (i) PACQ ensures an excellent level of load balancing; (ii) the total number of queries increases under an upper-bound; (iii) when the number of users grows, the number of queries asked per user gets closer and closer to zero.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952215. This work also received support from University of Montpellier and I-Site MUSE under CAR-UM2020/2021 project. We thank Nassim Belmecheri and Yahia Lebbah who provided insight and expertise that greatly assisted the work. We would also like to show our gratitude to the "anonymous" reviewers for their insightful suggestions and careful reading.

References

Amdahl, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C. doi:10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Beldiceanu, N.; and Simonis, H. 2012. A Model Seeker: Extracting Global Constraint Models from Positive Examples. In Milano, M., ed., *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, 141–157. Springer. doi:10.1007/978-3-642-33558-7_13. URL https://doi.org/10.1007/978-3-642-33558-7_13.

Bessiere, C.; Carbonnel, C.; Dries, A.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.; Stergiou, K.; Tsouros, D. C.; and Walsh, T. 2020. Partial Queries for Constraint Acquisition. *CoRR* abs/2003.06649. URL https://arxiv.org/abs/2003.06649.

Bessiere, C.; Coletta, R.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.; and Walsh, T. 2013. Constraint Acquisition via Partial Queries. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 475–481.

Bessiere, C.; Koriche, F.; Lazaar, N.; and O'Sullivan, B. 2017. Constraint acquisition. *Artif. Intell.* 244: 315–342. doi:10.1016/j.artint.2015.08.001. URL https://doi.org/10.1016/j.artint.2015.08.001.

Freuder, E. C.; and Wallace, R. J. 1998. Suggestion Strategies for Constraint-Based Matchmaker Agents. In Maher, M. J.; and Puget, J., eds., *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, 192–204. Springer. doi:10.1007/3-540-49481-2_15. URL https://doi.org/10.1007/3-540-49481-2_15.

Lallemand, C.; and Gronier, G. 2012. Enhancing User eXperience During Waiting Time in HCI: Contributions of Cognitive Psychology. In *Proceedings of the Designing Interactive Systems Conference*, DIS '12, 751–760. New York, NY, USA: ACM. ISBN 978-1-4503-1210-3. doi:10.1145/2317956.2318069. URL http://doi.acm.org/10.1145/2317956.2318069.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On Learning Constraint Problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, 45–52. IEEE Computer Society. doi:10.1109/ICTAI.2010.16. URL https://doi.org/10.1109/ICTAI.2010.16.

Régin, J.; and Malapert, A. 2018. Parallel Constraint Programming. In Hamadi, Y.; and Sais, L., eds., *Handbook of Parallel Constraint Reasoning*, 337–379. Springer. doi:10.1007/978-3-319-63516-3_9. URL https://doi.org/10.1007/978-3-319-63516-3_9.

Shchekotykhin, K. M.; and Friedrich, G. 2009. Argumentation Based Constraint Acquisition. In Wang, W.; Kargupta, H.; Ranka, S.; Yu, P. S.; and Wu, X., eds., *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, 476–482. IEEE Computer Society. doi:10.1109/ICDM.2009.62. URL https://doi.org/10.1109/ICDM.2009.62.

Tsouros, D. C.; and Stergiou, K. 2020. Efficient multiple constraint acquisition. *Constraints* doi:10.1007/s10601-020-09311-4. URL https://doi.org/10.1007/s10601-020-09311-4.