

Towards Anytime Constraint Acquisition

Aicha Boukhari^{1,2}, Sébastien Bardin¹, Arnaud Gotlieb³, Nadjib Lazaar², and Grégoire Menguy¹

¹ CEA LIST, Université Paris Saclay, France
`{first.last}@cea.fr`

² LISN, Université Paris Saclay, France
`lazaar@lisen.fr`

³ SIMULA Research Laboratory, Oslo, Norway
`arnaud@simula.no`

Abstract. Constraint Acquisition (CA) aims at learning a target concept represented as a set of constraints through queries to an oracle. Classical CA approaches focus on identifying the constraints defining the concept. However, convergence is often unrealistic when the acquisition process is limited by time or query budgets. This paper introduces an anytime CA framework. Rather than requiring full reconstruction of the target concept, our approach focuses on iterative model refinement to produce increasingly accurate approximations. We propose Δ QUERYGEN, a query generation strategy designed to efficiently improve the current hypothesis. Integrated into the acquisition process, this strategy yields CANDY, an incremental approach that rapidly refines approximations under limited resources. Our approach is validated on program precondition inference, a recent real-world application of CA. Experimental results show that CANDY significantly improves approximation quality under tight time budgets, reaching comparable coverage levels up to 16 times faster than PRECA (a conjunctive CA baseline) and 6 times faster than DCA (a state-of-the-art disjunctive CA system), while reducing by 2 to 3 the average waiting time between positive queries on the hardest benchmarks.

Keywords: Constraint Acquisition · Anytime Learning · Program Verification · Precondition Inference

1 Introduction

Constraint Acquisition (CA) aims to assist non-expert users in modeling problems with constraints. It learns the constraints defining a target concept through queries to an oracle. In active CA, the learner does not rely on a predefined dataset of queries; instead, it generates them during the learning process to obtain informative feedback. Several interaction models have been proposed, based on different types of queries, such as membership queries [6], partial queries [5], Matchmaker-based interactions [7], or qualitative queries [3].

In this work, we focus on active CA based on membership queries in a black-box interaction model. A *membership query* consists in submitting a complete

assignment to an oracle, which answers whether that example belongs to the target concept or not, i.e., whether it is a solution or a non-solution. This setting naturally arises in program verification tasks, where constraints represent program preconditions. Given an input assignment, the oracle executes the program and checks whether the desired postcondition is met, thus providing the classification [6]. Membership queries are particularly suitable in this context because the execution of the program requires fully specified inputs, while other query types would require access to the internal structure of the program, which is not available in the black-box setting.

Problem. Despite significant progress in CA [14, 15, 5, 3], learning the exact target concept can remain computationally expensive and often does not converge within realistic time or query budgets. However, in many applications, an exact reconstruction is not strictly necessary. Instead, quickly obtaining a useful approximation of the target concept can already support downstream tasks. For instance, in precondition inference [16, 15, 14], a key task in program verification, a sound but more restrictive precondition can still ensure safe program execution. In such contexts, learning an over-constrained network of the target concept early in the acquisition process can already provide practical value. Existing approaches only partially address this need. PRECA [16] applies CONACQ [6] to precondition inference and can be stopped at any time, but when interrupted before convergence, it often produces an approximation that remains too imprecise to be useful. Conversely, DCA [15], which extends CA to disjunctive constraint networks and significantly improves convergence performance, does not exhibit anytime behavior: when interrupted early, it typically returns the empty concept (false), corresponding to the worst possible approximation. This paper aims to reconcile these two desirable properties: the ability to stop PRECA at any time and the inference efficiency of DCA. By adopting an anytime learning perspective [18], we seek to produce an increasingly precise constraint network throughout the acquisition process, obtaining useful approximations even under tight query or time budgets.

Contributions. This paper makes the following contributions:

1. We propose the first anytime approach to CA that focuses on rapidly generalizing the most specific network consistent with the information gathered so far. Our method maintains this network and progressively refines the concept learned during the acquisition process;
2. To accelerate this refinement, we introduce Δ QUERYGEN, a new query generation strategy that prioritizes queries based on a distance criterion. This strategy enables the learner to improve the current approximation more efficiently under limited time or query budgets;
3. We evaluate the resulting approach, CANDY, on program precondition inference benchmarks. The results show that it refines constraint networks significantly faster than the existing methods, achieving up to a $16\times$ speedup over the state-of-the-art results. CANDY also divides by 2 to 3 the waiting time between positive queries in the hardest cases compared to competitors.

2 Motivating Example

Consider the following program, where x_1 , x_2 , and x_3 are integer variables:

```
int compute(int x1, int x2, int x3) {
    assert(x1>0);
    if (x2 == 0) return x1 / x3;
    else return x1 / x2;
}
```

Precondition inference asks: *for which inputs does compute terminate without error?* Note that the input $(1, 0, 0)$ must be excluded, as it triggers a division by zero. For this function, the answer is $x_1 > 0 \wedge \neg(x_2 = 0 \wedge x_3 = 0)$ which is equivalent to $x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 \neq 0)$, that is an inherently disjunctive property. We take $D = [-4, 4]$, giving $9^3 = 729$ possible inputs, of which 320 satisfy the target precondition. The constraint basis is $B = \{x_i > 0, x_i \leq 0, x_i \geq 0, x_i < 0, x_i = 0, x_i \neq 0 \mid i \in 1..3\}$. To enable PRECA to learn disjunctive concepts, its basis must be augmented with all disjunctions of size up to 4 [16].

The anytime gap in existing approaches. Tables 1 and 2 reveal a fundamental weakness shared by both PRECA and DCA: neither produces a useful intermediate approximation under a tight budget.

PRECA can be stopped early, but the improvement is marginal in practice. After 18 queries, the learned concept still does not cover any solution. Then it progresses slowly: 16/320 in query 19, 112/320 in query 22, 256/320 in query 25. Meaningful generalization occurs only in the final queries, making PRECA poorly suited for settings where time or query budgets are constrained.

Table 1. Learned concept at each iteration of PRECA on the `compute` example.

Iter. Query	Class.	Learned concept	Time(s)	#Sol
1 (-4, -4, -4)	NO	<i>false</i>	0.238	0/320
⋮ ...	NO	<i>false</i>	...	0/320
19 (1, 1, 0)	YES	$x_1 > 0 \wedge x_2 > 0 \wedge x_3 = 0$	246.313	16/320
20 (1, 0, 1)	YES	$x_1 > 0 \wedge x_2 \geq 0 \wedge x_3 \geq 0 \wedge (x_2 > 0 \vee x_3 > 0) \wedge (x_2 = 0 \vee x_3 = 0)$	263.787	32/320
21 (1, 0, -4)	YES	$x_1 > 0 \wedge x_2 \geq 0 \wedge (x_2 > 0 \vee x_3 \neq 0) \wedge (x_2 = 0 \vee x_3 = 0)$	353.394	48/320
22 (1, 1, 1)	YES	$x_1 > 0 \wedge x_2 \geq 0 \wedge (x_2 > 0 \vee x_3 \neq 0) \wedge (x_2 = 0 \vee x_3 \geq 0)$	410.469	112/320
23 (1, -4, 1)	YES	$x_1 > 0 \wedge (x_2 > 0 \vee x_3 \neq 0) \wedge (x_2 \geq 0 \vee x_3 > 0) \wedge (x_2 = 0 \vee x_3 \geq 0)$	410.878	176/320
24 (1, -4, 0)	YES	$x_1 > 0 \wedge (x_2 = 0 \vee x_3 \geq 0) \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	411.287	192/320
25 (1, 1, -4)	YES	$x_1 > 0 \wedge (x_2 \geq 0 \vee x_3 \geq 0) \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	411.683	256/320
26 (1, 0, 0)	NO	$x_1 > 0 \wedge (x_2 \geq 0 \vee x_3 \geq 0) \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	412.029	256/320
27 (1, -4, -4)	YES	$x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	418.921	320/320

DCA exhibits a purely all-or-nothing behavior. Despite collecting positive evidence throughout, queries 2, 5, 10, and 12 are all classified *yes*, its learned concept remains *false* with 0/320 solutions covered until the very last query, where it jumps abruptly to 320/320. This is a structural limitation: DCA systematically rules out regions of the input space through negative queries and only

commits to a solution once the entire space has been partitioned. Interrupting it earlier does not produce any exploitable precondition.

Table 2. Learned concept at each iteration of DCA on the `compute` example.

Iter.	Query	Class.	Learned concept	Time(s)	#Sol
1	$(-4, -4, -4)$	NO	<i>false</i>	0.660	0/320
2	$(1, -4, -4)$	YES	<i>false</i>	0.922	0/320
3	$(-4, 1, -4)$	NO	<i>false</i>	1.174	0/320
4	$(-4, -4, 0)$	NO	<i>false</i>	1.637	0/320
5	$(1, 1, -4)$	YES	<i>false</i>	1.887	0/320
6	$(0, 1, -4)$	NO	<i>false</i>	2.165	0/320
7	$(-4, 1, 0)$	NO	<i>false</i>	2.431	0/320
8	$(0, -4, 0)$	NO	<i>false</i>	2.881	0/320
9	$(-4, -4, 1)$	NO	<i>false</i>	3.733	0/320
10	$(1, -4, 0)$	YES	<i>false</i>	3.996	0/320
11	$(0, -4, -4)$	NO	<i>false</i>	4.949	0/320
12	$(1, 0, -4)$	YES	<i>false</i>	5.210	0/320
13	$(-4, 0, 1)$	NO	<i>false</i>	5.441	0/320
$\vdots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
26	$(-4, 0, -4)$	NO	<i>false</i>	10.407	0/320
27	$(0, 0, -4)$	NO	$x > 0 \wedge (y \neq 0 \vee z \neq 0)$	10.655	320/320

Our proposal. CANDY breaks this pattern by design. Table 4 summarizes the key differences between the three approaches at the conceptual level. As shown in Table 3, CANDY already returns a non-trivial precondition covering 64/320 solutions after just 2 queries, inferring $x_1 > 0 \wedge x_2 < 0 \wedge x_3 < 0$. Coverage grows steadily with each positive query, reaching 288/320 by query 9 and converging to the exact target at query 13 — more than twice as fast as DCA and PRECA.

This behavior comes from two design choices. First, CANDY constructs the most specific network: each positive query immediately expands the set of covered inputs, so the approximation is informative from the very first positive query. Second, the Δ QUERYGEN heuristic actively steers the search toward regions likely to yield positive classifications, enabling rapid successive refinements rather than broad but uninformative exploration.

The contrast is stark: where DCA and PRECA offer nothing until near-convergence, CANDY delivers an increasingly precise and always-sound precondition at every step of the acquisition process.

3 Background

Constraint Acquisition (CA). Constraint Acquisition [6] is an interactive learning process between a *learner* and an oracle whose goal is to identify a

Table 3. Learned concept at each iteration of CANDY on the `compute` example.

Iter.	Query	Class.	Learned concept	Time (s)	#Sol
1	(-4, -4, -4)	NO	<i>false</i>	0.600	0/320
2	(1, -4, -4)	YES	$x_1 > 0 \wedge x_2 < 0 \wedge x_3 < 0$	0.877	64/320
3	(-4, 1, -4)	NO	$x_1 > 0 \wedge x_2 < 0 \wedge x_3 < 0$	4.466	64/320
4	(1, 0, -4)	YES	$x_1 > 0 \wedge x_2 \leq 0 \wedge x_3 < 0$	4.722	80/320
5	(1, -4, 1)	YES	$x_1 > 0 \wedge ((x_2 < 0 \wedge x_3 \neq 0) \vee (x_2 = 0 \wedge x_3 < 0))$	4.975	144/320
6	(0, -4, -4)	NO	$x_1 > 0 \wedge ((x_2 < 0 \wedge x_3 \neq 0) \vee (x_2 = 0 \wedge x_3 < 0))$	5.227	144/320
7	(1, -4, 0)	YES	$x_1 > 0 \wedge (x_2 < 0 \vee (x_2 = 0 \wedge x_3 < 0))$	5.468	160/320
8	(1, 1, -4)	YES	$x_1 > 0 \wedge (x_2 < 0 \vee x_3 < 0)$	6.227	224/320
9	(1, 1, 1)	YES	$x_1 > 0 \wedge (x_2 < 0 \vee x_3 < 0 \vee (x_2 > 0 \wedge x_3 > 0))$	6.480	288/320
10	(-4, 1, 1)	NO	$x_1 > 0 \wedge (x_2 < 0 \vee x_3 < 0 \vee (x_2 > 0 \wedge x_3 > 0))$	6.744	288/320
11	(0, 1, 1)	NO	$x_1 > 0 \wedge (x_2 < 0 \vee x_3 < 0 \vee (x_2 > 0 \wedge x_3 > 0))$	6.979	288/320
12	(1, 1, 0)	YES	$x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 < 0)$	7.226	304/320
13	(1, 0, 1)	YES	$x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	7.480	320/320
$\vdots$	$\dots$	NO	$x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	$\dots$	320/320
27	(-4, 0, 0)	NO	$x_1 > 0 \wedge (x_2 \neq 0 \vee x_3 \neq 0)$	11.049	320/320

Table 4. Conceptual comparison of CA approaches.

	Query type	Expressiveness	Anytime?
PRECA	Membership	Conjunctive	yes
DCA	Membership	Conj. + Disj.	no
CANDY	Membership	Conj. + Disj.	yes

target Constraint Network T , i.e., a set of constraints defined over a shared set of variables X with finite domains D . A *constraint* c on a subset of variables specifies the combinations of values that are allowed. An example $e \in D^{|X|}$ satisfies c if its projection onto the variables of c belongs to $\text{sol}(c)$. An example is a solution of a network C if it satisfies all constraints in C . The learner is given a constraint *language* Γ of bounded-arity relations from which constraints can be instantiated. The resulting *constraint basis* B is the set of all constraints that can be built from Γ on (X, D) . A network C is “representable” if $C \subseteq B$. We assume that the constraint basis B is *closed under negation*, which means that for each $c \in B$, its negation c' is also in B . The target concept can be viewed as a Boolean function over $D^{|X|}$. Learning proceeds through *membership queries*: the learner submits an example e to the oracle, which classifies it as positive (e^+) or negative (e^-). Given a labeled set of examples $E = E^+ \cup E^-$, a network C *agrees* with E if it accepts all positive examples and rejects all negative ones.

In classical CA, learning converges when B is exhausted, i.e., no informative query remains, and all networks consistent with E are solution-equivalent to the target concept — simultaneously recovering the true precondition when the target is $\vee$ -representable by B .

However, before convergence, the set of consistent networks induces two natural approximations of the target concept. The *most specific network* $\underline{T}$ is the most restrictive network in B consistent with E , whose solutions form an over-constrained network of the target concept. Conversely, the *most general* constraint network $\overline{T}$ is the least restrictive network consistent with E , defining an under-constrained network.⁴ When the process is interrupted before convergence, these networks characterize the current bounds on the target concept.

Precondition Inference using CA. Recent work [16, 15, 14] adapts CA for precondition inference, a key task in software verification: identifying a precondition P on the input of a function under which execution terminates and satisfies a postcondition Q [10].

In this context, PRECA replaces the human oracle with an automated one. The learner proposes concrete inputs and the oracle executes the program on them. Depending on the outcome, the oracle returns *yes* if the execution terminates and Q is fulfilled, *no* if Q is violated, or *ukn* if the program fails to terminate within the time budget. Following a conservative approach, *ukn* is treated as a negative example, ensuring that the inferred precondition only admits inputs guaranteed to satisfy Q . Although PRECA can be used at any time, stopping before convergence can produce approximations that are too coarse.

Disjunctive CA (DCA) [15] extends CA to learn constraint networks in a disjunctive form by leveraging the enumeration of *Maximal Satisfiable Subsets (MSS)*. An MSS is a satisfiable subset of the basis that becomes unsatisfiable if it is extended with any other constraint. We denote by MSS_B the set of MSSs of the basis B . Assuming a basis B closed under negation, each MSS defines a set of inputs that all share the same classification [15]. As a result, querying a single representative from an MSS suffices to classify all its elements, making DCA particularly efficient for learning disjunctive preconditions. A network C is $\vee$ -representable by B if each constraint in C is either in B or a disjunction of constraints from B (see Proposition 3). DCA queries one representative example per MSS M and, if the example is classified as negative or *ukn*, adds the negation of the entire MSS ($\neg M$) to the network.

4 The CANDY Framework

We present here CANDY, an incremental and anytime CA framework that combines two complementary components: CANY and Δ QUERYGEN. CANY extends DCA by focusing on membership queries that contribute to refining the most specific network ($\underline{T}$), progressively building it using positive examples rather than relying on acquisition based on negative-examples over the whole space.

⁴ Overconstrained concepts correspond to under-approximations in program analysis, which takes a solution-centric view (fewer solutions than expected). On the other hand, CA related-works [6] refer to them as over-approximations, adopting a constraints-centric view (more constraints than expected).

CANY. Algorithm 1 takes a basis closed under negation as input and enumerates MSSs, e.g., using the MARCO algorithm [12]. For each MSS, a representative example is selected and submitted to the oracle as a membership query. If the example is classified as positive, the current $\underline{T}$ is expanded by adding all MSS solutions as a new disjunct (line 4), progressively constructing and refining the boundary $\underline{T}$. Negative queries or *ukn* queries are ignored since $\underline{T}$ only contains inputs that satisfy the target concept and cannot be refined from negative examples. Contrary to DCA, which jumps directly from the trivial $\underline{T} = \text{false}$ to T at the end of the process, CANDY constructs $\underline{T}$ progressively. Thus, it produces intermediate approximations exploitable at any point if the acquisition is interrupted, embodying the principle of anytime algorithms [18]. Although CANDY already provides a practical anytime refinement, improving the accuracy of these intermediate $\underline{T}$ approximations remains crucial. To address this, we introduce a guided query generation strategy to accelerate the refinement process.

Δ QUERYGEN. We introduce Δ QUERYGEN, a query generation heuristic designed to accelerate the generalization of $\underline{T}$ by prioritizing positive queries. The guiding intuition is simple: *queries that satisfy a large number of constraints in common with known positive examples are more likely to be classified as positive.* This observation is empirically supported by an analysis of 40 precondition inference problems, in which both PRECA and DCA converged and generated all required queries. On average, 84% of the queries that shared the maximum number of constraints satisfied with a given positive example were themselves classified as positive. Consequently, Δ QUERYGEN prioritizes queries that maximize overlap with previously observed positive examples. This prioritization is formalized as an optimization problem through a dedicated metric Δ , which quantifies the potential informativeness of each candidate query.

Definition 1 (The Δ Metric). *Given two examples e and e' , and a basis B , we define $\Delta(e, e')$ as the number of constraints in B that are satisfied by one example and violated by the other: $\Delta(e, e') = |\{c \in B \mid (e \models c) \Leftrightarrow (e' \not\models c)\}|$*

Δ QUERYGEN (Algorithm 2) generates membership queries guided by a given MSS. The core idea is to produce queries that satisfy as many constraints as possible in common with the MSS. It follows from intuition that such queries are likely to be classified positively by the oracle. The algorithm takes as input a basis B closed under negation, the current **Map** formula representing hard constraints accumulated from previous queries, and an MSS M . Two sets of unit clauses are extracted from B : Ω , corresponding to the constraints in M , and Φ , the remaining constraints. A *weighted MaxSAT* solver is then used to select a candidate set of constraints *seed*, assigning high weight to Ω and low weight to Φ to prioritize constraints from the MSS. If the solver returns a model, an example e is extracted from the solutions of *seed*, and *Map* is updated with a *blocking down clause* (line 12) to prevent generating redundant queries later. The example e is then returned and submitted to the oracle as a membership query. If there is no satisfying assignment for *seed*, then a *shrink procedure* computes a minimal unsatisfiable subset (MUS) to ensure progress, and *Map* is updated

Algorithm 1: CANY

Input: basis $B = \{C_1, \dots, C_n\}$
Output: $\underline{T}$

```

1  $\underline{T} \leftarrow \perp$ 
2 forall  $M \in MSS_B$  do
3   | pick  $e \in sol(M)$ 
4   | if  $ask(e) = yes$  then  $\underline{T} \leftarrow \underline{T} \vee M$  ;
5 return  $\underline{T}$ 

```

Algorithm 2: Δ QUERYGEN

Input: Map Map , basis B , MSS M
Output: a query q

```

1 while true do
2   |  $\Omega \leftarrow \{a(c) \mid c \in M\}$ 
3   |  $\Phi \leftarrow \{a(c) \mid c \in B \setminus M\}$ 
4   |  $model \leftarrow \text{WEIGHTEDMAXSAT}(\Omega, \Phi)$ 
5   |  $hard = Map$ ,
6   |  $soft = \Omega$  with weight  $|M| + 1$ ,
7   |  $soft = \Phi$  with weight 1)
8   | if  $model \neq nil$  then
9     |  $seed \leftarrow \{c \in B \mid a(c) = 1 \text{ in } model\}$ 
10    | pick  $e \in sol(seed)$ 
11    | if  $e \neq nil$  then
12      |  $Map \leftarrow Map \wedge blockDown(seed)$ 
13      | return  $e$ 
14    | else
15      |  $MUS \leftarrow shrink(seed)$ 
16      |  $Map \leftarrow Map \wedge blockUp(MUS)$ 
17  | else return nil ;

```

Algorithm 3: CANDY

Input: A basis B closed under negation
Output: $\underline{T}$

```

1  $Map \leftarrow true$   $found \leftarrow false$ ;  $Q \leftarrow \emptyset$ ;
    $d_{min} \leftarrow nil$ ;  $\underline{T} \leftarrow \perp$ 
2 while  $Map$  is satisfiable do
3   | if  $\neg found$  then
4     | pick  $M \in MSS_B$ 
5     | pick  $e \in sol(M)$ 
6     | if  $Ask(e) = yes$  then
7       |  $found \leftarrow true$ 
8       |  $Map \leftarrow blockDown(M)$ 
9       |  $\underline{T} \leftarrow M$ ;
10    | else
11      |  $Map \leftarrow blockDown(M)$ 
12      | continue
13    |  $e' \leftarrow \Delta\text{QUERYGEN}(Map, B, M)$ 
14    | if  $e' = nil$  then break;
15    |  $M' \leftarrow \{c \in B \mid e' \models c\}$ 
16    | if  $Ask(e') = yes$  then
17      |  $\underline{T} \leftarrow \underline{T} \vee M'$ 
18      | add  $M'$  to the front of  $Q$ 
19    | else push  $M'$  to end of  $Q$  ;
20    |  $\delta \leftarrow \Delta(e, e')$ 
21    | if  $d_{min} = nil$  then  $d_{min} \leftarrow \delta$ ;
22    | if  $\delta > d_{min}$  then
23      | if  $Q = \emptyset$  then break;
24      |  $M \leftarrow$  poll first element of  $Q$ 
25      | pick  $e \in sol(M)$ 
26      |  $d_{min} \leftarrow nil$ 
27 return  $\underline{T}$ 

```

with a *blocking clause* (line 16). Concretely, SHRINK starts from *Seed* and removes each c whose deletion keeps the set unsatisfiable, returning a minimal core. These functions are defined as follows [13] :

$$\text{BLOCKDOWN}(M) = \bigvee_{i: c_i \notin M} x_i, \text{BLOCKUP}(M) = \bigvee_{i: c_i \in M} \neg x_i$$

This procedure iterates until a query is generated or no further queries exist. By guiding query selection with MSSs and weighted MaxSAT, Δ QUERYGEN prioritizes positive queries that expand the $\underline{T}$, enabling effective anytime learning under tight time or query budgets.

CANDY. Algorithm 3 presents CANDY, a smart combination of CANY and Δ QUERYGEN for fast anytime constraint acquisition. The input is a basis B closed under negation, enabling the enumeration of all maximal satisfiable subsets (MSSs). Note that the number of MSSs is not $2^{|B|}$ in general, since not all subsets of B are satisfiable; in practice, the number of MSSs is significantly smaller than $2^{|B|}$. The algorithm maintains a queue Q of discovered MSSs and iteratively generates membership queries as follows:

1. **Initialization:** The first MSS is sampled using an enumerator (MARCO [12] in our case). If a representative example $e \in sol(M)$ is classified as positive,

- $\underline{T}$ is initialized with M and M is added to the front of Q ; negative MSSs are added to the end. Blocking clauses prevent revisiting the same MSS.
2. **Query generation with Δ QUERYGEN:** For the current MSS M , Δ QUERYGEN produces an example e' that satisfies as many constraints in common with M as possible, increasing the probability of a positive classification.
 3. **Updating $\underline{T}$:** Positive examples expand $\underline{T}$ by adding the corresponding MSS M' as a new disjunct. The MSSs corresponding to the positive examples are added to the front of Q , while the negative ones go to the end. Blocking clauses ensure progress and prevent redundancy.
 4. **Queue management:** The algorithm tracks the Δ distances between queries; if the distance to the current MSS exceeds the observed minimum, the next MSS is selected from Q to guide the exploration efficiently.
 5. **Termination:** The loop continues until the Map becomes unsatisfiable or no further MSSs remain in the queue. At this point, $\underline{T}$ represents a disjunction of all positive MSSs encountered, providing the maximally specific concept available so far, i.e., an over-constrained version of the target concept. This anytime behavior ensures that a valid $\underline{T}$ is available even if the process is interrupted early.

Example 1. We illustrate CANDY with the `compute` program introduced in Section 2. Table 3 summarizes the evolution of $\underline{T}$ and the number of covered solutions at each step of CANDY acquisition process.

Step 1 — First positive example. MARCO first enumerates the negative example, $(-4, -4, -4)$, before returning the first positive one: $(1, -4, -4)$, yielding $M_1 = \{x_1 > 0, x_1 \geq 0, x_1 \neq 0, x_2 \leq 0, x_2 < 0, x_2 \neq 0, x_3 \leq 0, x_3 < 0, x_3 \neq 0\}$. $\underline{T}$ is initialized to M_1 , `BLOCKDOWN`(M_1) is added to *Map*, and Δ QUERYGEN is invoked for the first time.

Step 2 — First Δ QUERYGEN call. Δ QUERYGEN calls weighted MaxSAT with *Map* as hard constraints and the constraints of M_1 as high-weight soft clauses, so as to find the next example satisfying as many constraints of M_1 as possible. The solver returns $(-4, 1, -4)$, which maximizes overlap with M_1 at distance $\Delta = 2$. The oracle returns *no*, so its corresponding MSS is blocked down in *Map* and pushed to the end of the queue. Δ QUERYGEN calls weighted MaxSAT again with the updated *Map*. It returns $(1, 0, -4)$, also at distance $\Delta = 2$ from M_1 . The oracle returns *yes*, yielding $M_2 = \{x_1 > 0, x_1 \geq 0, x_1 \neq 0, x_2 \leq 0, x_2 \geq 0, x_2 = 0, x_3 \leq 0, x_3 < 0, x_3 \neq 0\}$. `BLOCKDOWN`(M_2) is added to *Map*, M_2 is pushed to the front of the queue, and $\underline{T}$ is updated to $M_1 \vee M_2$.

Step 3 — Exhausting minimum-distance examples from M_1 . Δ QUERYGEN continues maximizing overlap with M_1 until no more examples at minimum distance from $(1, -4, -4)$ remain; then moves to the next MSS in the queue.

Step 5 — Termination. Once all MSSs are exhausted CANDY terminates, returning: $\underline{T} = M_1 \vee M_2 \vee \dots \vee M_8 \quad \equiv \quad x > 0 \wedge (y \neq 0 \vee z \neq 0)$

Note that CANDY continues exploring all remaining MSSs even after reaching full coverage, to guarantee correctness and completeness.

This table illustrates the anytime behavior of CANDY: unlike DCA, which covers 0 solutions until the very last example, and PRECA, which covers 0 so-

lutions for the first 18 examples before reaching only 16 solutions at example 19, CANDY already covers 64 solutions out of 320 after only the second example, and reaches 320 solutions by example 13. This progressive refinement confirms that CANDY produces increasingly precise approximations throughout the acquisition process, making it significantly more informative than existing approaches under tight time or example budgets.

Theoretical Analysis. CANDY enumerates MSSs in the same way as DCA does, and so inherits all the theoretical guarantees and properties established for DCA. In contrast, CANDY introduces its own query generation mechanism (Δ QUERYGEN). We now prove that CANDY generates only informative queries, terminates, and is correct, while additionally exhibiting the *anytime* property, producing a progressively refined $\underline{T}$ network throughout the acquisition process.

Proposition 1 (Informativity). *Each query generated by Δ QUERYGEN is informative, i.e., non-redundant with respect to previously generated queries.*

Proof. Given a basis B closed under negation, at each iteration of Algorithm 3, the query e is generated by Δ QUERYGEN at line 13. The MaxSAT call returns a model under the current Map , from which the subset $seed$ is induced. If $seed$ is satisfiable, it is maximal by construction. Indeed, for any constraint $c \in B \setminus seed$, adding c to $seed$ would violate satisfiability; otherwise, the MaxSAT objective could be strictly improved by including c , contradicting optimality of the returned model. Since each MSS has a disjoint solution space under a basis closed under negation, every call to Δ QUERYGEN returns an example from a distinct MSS, and therefore each example introduces at least one new constraint not satisfied by prior queries. Consequently, each generated query reduces the version space, satisfying the definition of an informative query in the CA framework [6].

Proposition 2 (Termination). *The CANDY algorithm always terminates.*

Proof. CANDY enumerates MSSs of the basis B that is closed under negation. Such enumeration is guaranteed to terminate after a finite number of iterations, since the number of MSSs is finite given a finite basis closed under negation. At each iteration, CANDY either:

1. generates a query from the next MSS and updates the current $\underline{T}$, or
2. applies blocking clauses (downward or upward) to prevent repeating previously explored solutions.

Each update strictly reduces the remaining search space defined by Map , and no MSS or solution is revisited due to the blocking mechanism. Since both the number of MSSs and the number of examples within each MSS are finite, and the algorithm progresses monotonically by consuming MSSs or shrinking the search space, the loop in Algorithm 3 ends after a finite number of iterations.

Proposition 3 (Correctness). *Given a basis closed under negation B and a target concept T that is $\vee$ -representable by B , the CANDY algorithm returns a network $\underline{T}$ such that $\underline{T} \equiv T$.*

Proof. CANDY constructs $\underline{T}$ incrementally by enumerating MSSs M of B using MARCO. From Proposition 2 in [15], each MSS M represents a set of examples whose classification under T is uniform. Thus, querying any example $e \in \text{sol}(M)$ allows CANDY to correctly determine whether M should contribute to $\underline{T}$.

At each iteration:

1. If e is positive, M (or the induced $M' \subseteq B$ from Alg. 2) is added as a disjunct to $\underline{T}$.
2. If e is negative, blocking clauses are added to Map to prevent revisiting solutions that are inconsistent with T .

Since B is closed under negation, CANDY enumerates all MSSs in a finite number of steps. Each positive MSS is included exactly once in $\underline{T}$, while negative MSSs are blocked. Finally, when all MSSs are processed, $\underline{T}$ contains exactly the disjunction of all MSSs that are positive under T , which by definition corresponds to T itself. Therefore, the algorithm returns $\underline{T} \equiv T$, proving the correctness.

Proposition 4 (Anytime). *The CANDY algorithm exhibits an anytime behavior: at any point during execution, it maintains a valid approximation $\underline{T}$ of the target concept T .*

Proof. CANDY constructs $\underline{T}$ incrementally by processing MSSs and integrating the outputs of $\Delta\text{QUERYGEN}$:

1. Each positive query produced by $\Delta\text{QUERYGEN}$ extends $\underline{T}$ with a new disjunct $M' \subseteq B$. By Proposition 2 in [15], M' correctly covers a set of examples uniformly classified as positive under T , ensuring soundness.
2. Negative queries trigger blocking clauses in Map , preventing the algorithm from revisiting invalid solutions while preserving all previously learned positive MSSs in $\underline{T}$.

Because CANDY updates $\underline{T}$ after every positive query, the current $\underline{T}$ always represents a good, albeit partial approximation of T . Thus, even if the process is interrupted before convergence, the returned $\underline{T}$ can be used safely, guaranteeing a monotonic increase in the coverage of the positive examples. This establishes the anytime property.

5 Experimental Evaluation

We investigate the benefits of our approach via the following research questions:

- **RQ1 (Quality of the learned MS).** Does CANDY produce higher-quality $\underline{T}$ networks compared to CANY and state-of-the-art approaches? We assess this by measuring the coverage of the valid input space achieved over time.
- **RQ2 (Ablation study).** Does $\Delta\text{QUERYGEN}$ effectively guide the acquisition process toward faster coverage of the search space? We compare CANDY and CANY to show $\Delta\text{QUERYGEN}$ benefit on anytime constraint acquisition.

- **RQ3 (Time impact of $\Delta\text{QUERYGEN}$).** How does $\Delta\text{QUERYGEN}$ affect the acquisition process in terms of query generation speed? We evaluate its impact on query generation time and the time between two consecutive positive queries.

Experimental Setup. CANY and CANDY are implemented in Java. We evaluate them on realistic precondition inference tasks on C functions extracted from the `libc`, `Siemens`, `DSA`, and `mbedtls` cryptographic libraries introduced in [15], comparing CANDY to CANY, DCA, and PRECA. Our analysis focuses on the most challenging instances where DCA or PRECA do not converge within 30s (23 functions). For simpler instances (36 examples converging in less than 30s) all methods exhibit similar convergence behavior. All experiments were run on a 13th-Gen Intel Core i7-13800H with 62GB RAM, under Ubuntu 24.04.1 LTS and OpenJDK 21.0.11. SAT/Weighted-MaxSAT solving uses SAT4J 2.3.5 [4] (`maxsat` module 2.3.4); constraint solving uses Choco-solver 4.10.2 [17]; model counting uses PopCon [8]. Each tool was given a 1000s timeout per instance. As the acquisition process is fully deterministic, each instance was executed once, so we report exact per-instance values rather than averages. DCA, CANY and CANDY are all implemented within the PRECA tool, available at <https://github.com/binsec/preca/>; an archived artifact reproducing all results of this paper is available on Zenodo (<https://zenodo.org/records/21723102>). We partitioned the 23 benchmark examples into two groups (details in Table 5):

- **Bench 1** gathers instances that do not converge within 30 seconds but for which at least one tool covers more than 5% of the solution space;
- **Bench 2** includes very complex cases where no competitor converges within 1000 seconds nor exceeds 5% coverage.

Basis Construction. The same constraint language Γ is used across all benchmarks. Since B contains all constraints buildable from Γ over (X, D) , it adapts to each function: pointer constraints (validity, overlap) appear only with pointer variables, and integer constraints only over integer pairs. Its size $|B|$ per instance is reported in Table 5. Here $|B_\vee|$ is the basis used by PRECA, which — unable to learn disjunctions — augments B with disjunctive constraints; to avoid an unbounded blow-up, we cap disjunction size at 3, the maximum needed to express the target concepts of our benchmarks. Even so, $|B_\vee|$ reveals the combinatorial explosion of closing B under disjunctions, whereas $\vee$ -representability offers a principled alternative: Γ matches the target structure and disjunctions are inferred incrementally from queries.

Evaluation Metric. When convergence is not achieved within the allotted time, different approaches typically generate distinct sets of queries, leading to learned concepts that are not directly comparable through logical implications. To allow a meaningful comparison, we rely on model counting [9] to quantify the generality of the concept. More precisely, we measure the number of solutions covered by each learned concept. For this purpose, we use the POPCON model

Table 5. Benchmark characteristics.

	Function	#Vars	Possible inputs	$ B $	$ B_V $
Bench1	array_init_pair_symmetr_nopost	4	3.02×10^5	42	882
	find_first	4	2.82×10^4	34	6,018
	mbedtls_aes_set_dec	5	1.30×10^5	50	222,050
	mbedtls_aes_set_key_enc	5	1.30×10^5	50	222,050
	mbedtls_md	8	1.68×10^{10}	32	173,888
	mbedtls_md_get_size	5	7.20×10^4	44	130,328
	memccpy	4	1.61×10^4	26	2,626
	memmem	4	9.22×10^3	30	4,090
	mvswap	4	8.75×10^3	40	87,440
	strcat	2	3.60×10^1	16	576
	strsep	3	2.1×10^5	36	648
	strtok	3	1.00×10^3	36	7,176
	strtok_r	4	4.98×10^7	64	2,048
	strtol	3	1.80×10^2	22	1,562
	strxfrm	3	1.08×10^2	20	1,160
	swap	2	3.60×10^1	16	576
Bench2	mbedtls_aes_crypt_cbc	8	1.70×10^{13}	106	5,202
	mbedtls_md_finish	9	7.52×10^9	120	7,200
	mbedtls_md_hmac_finish	6	1.25×10^{10}	78	3,042
	mbedtls_md_hmac_reset	8	2.48×10^8	96	4,608
	mbedtls_md_hmac_update	7	6.69×10^{10}	102	171,802
	mbedtls_md_starts	8	4.29×10^8	96	4,608
	mbedtls_md_update	10	7.52×10^{10}	144	10,368

counter introduced in [8, 11]. This metric enables a fine-grained comparison of the precision and coverage of the concepts produced by the different approaches.

RQ1 results. Table 6 compares CANDY with PRECA and DCA on Bench 1. We observe that CANDY exhibits a clear anytime acquisition behavior, achieving faster coverage growth than PRECA and DCA. The gap is largest under short budgets. Here, CANDY is already able to infer general concepts. On bench 1, CANDY infers a concept $\underline{T}$ covering 20% of the ground truth 16 times faster than PRECA and 6 times faster than DCA. Moreover, CANDY also outperforms PRECA and DCA for higher thresholds. For instance, against PRECA, our CANDY proposal is the first to infer a concept $\underline{T}$ that covers 50% and 80% of the ground truth for 14/16 functions. More importantly, CANDY reaches these thresholds 11 times and 8 times faster than PRECA, respectively. These observations remain true for each threshold. CANDY also infers over-constrained concepts faster than DCA. It first infers a $\underline{T}$ that covers 50% of the ground truth for 16/16 functions, 3 times faster than DCA. CANDY also reaches the 80% threshold the first for 14/16 cases, being almost 2 times faster than DCA. Interestingly, we observe that CANDY and DCA are roughly equivalent for exact learning, CANDY being only 0.92 times slower than DCA. In Bench 2, no competitor can

Table 6. Comparison on Bench 1 for multiple coverage thresholds. #First is the number of functions (out of 16) where CANDY reaches the threshold first (including functions where the competitor never reaches it). Writing t_{tool} for the time taken by a tool to reach the threshold on a function, Speed-up is the arithmetic mean of the ratios $t_{\text{competitor}}/t_{\text{CANDY}}$, taken only over the functions where both tools reach the threshold.

	$\underline{T}$ coverage	20%	40%	50%	60%	80%	90%	100%
CANDY vs PRECA	#First	14/16	15/16	14/16	14/16	14/16	13/16	13/16
	Speed-up	$\times 16.28$	$\times 12.94$	$\times 11.69$	$\times 9.87$	$\times 8.06$	$\times 7.43$	$\times 5.87$
CANDY vs DCA	#First	16/16	16/16	16/16	16/16	14/16	12/16	5/16
	Speed-up	$\times 6.1$	$\times 3.73$	$\times 3.14$	$\times 2.69$	$\times 1.98$	$\times 1.72$	$\times 0.92$
CANDY vs CANY	#First	12/16	15/16	15/16	15/16	15/16	15/16	11/16
	Speed-up	$\times 2.04$	$\times 2.16$	$\times 2.19$	$\times 2.14$	$\times 1.92$	$\times 1.87$	$\times 1.1$

converge or reach coverage greater than 5%. Still, in Figure 1 we observe that CANDY infers a more general $\underline{T}$ than PRECA and DCA. Indeed, PRECA and DCA always return the empty concept — except for `mbedtlsls_aes_crypt_cbc`. On the other hand, CANDY always returns a non-empty concept.

Conclusion: CANDY provides improved anytime behavior compared to the state-of-the-art. CANDY learns higher-quality $\underline{T}$ networks within a given time budget, up to 16 times faster than PRECA and 6 times faster than DCA. It confirms that CANDY significantly accelerates the acquisition process and leads to more general $\underline{T}$ under tight time constraints.

RQ2 results (Ablation study). Table 6 compares CANDY and CANY on Bench 1. We observe that CANDY infers an over-constrained $\underline{T}$ approximately 2 times faster than CANY. Moreover, both show similar performance to achieve 100% coverage. On Bench 2, Figure 1 shows similar trends. On 4/7 examples, CANDY always infers more general concepts than CANY. For the remaining 3 examples, CANY is slightly better for small time budgets, but is quickly caught by CANDY. This confirms that $\Delta\text{QUERYGEN}$ significantly accelerates the acquisition process, leading to more general $\underline{T}$ under tight time constraints.

RQ3 results. Table 7 shows that CANDY generates queries slightly slower than other tools but achieves a shorter average time between positive queries, a trend amplified on Bench 2 with a mean time of 9 seconds, compared to roughly 21 seconds for PRECA and more than 30 seconds for DCA and CANY.

6 Related Work

Active CA builds on Angluin’s query-driven framework [1, 2], where a learner queries an oracle to identify a hidden concept. Matchmaker [7] exploits user feedback but requires users to name violated constraints, while CONACQ.2 [6] and QUACQ [5] use membership and partial queries, mainly for conjunctive

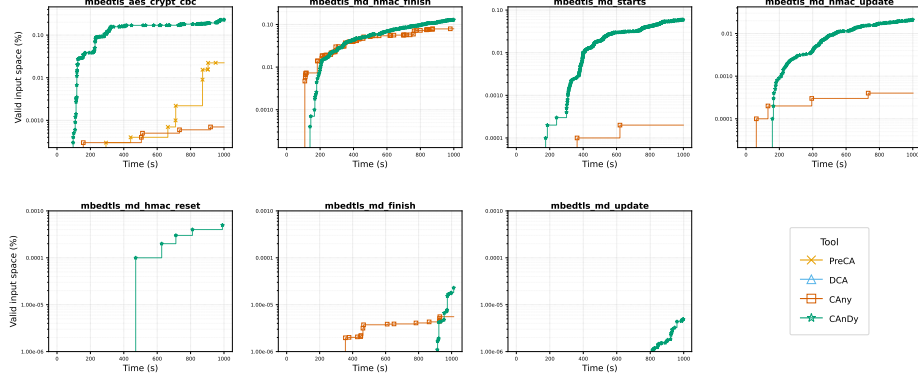


Fig. 1. Evolution of the learned $\underline{T}$ coverage (%) over time on Bench 2

Table 7. Average query generation time (all queries) and average waiting time between two successive positive queries, per tool.

Tool	Bench 1		Bench 2	
	Avg all (s)	Avg positive (s)	Avg all (s)	Avg positive (s)
PreCA	116.471	17.920	233.907	21.233
DCA	0.401	1.842	1.044	33.515
CAny	0.433	2.106	1.116	32.502
CAnDy	0.623	1.010	1.832	9.776

networks. DCA [15] handles disjunctive concepts by enumerating MSSs with MARCO [12], as we do. However, these methods target only the final constraint network. In contrast, we take an anytime perspective: rather than focusing on convergence, we continuously refine the approximation, guiding MSS exploration with a Weighted MaxSAT formulation that prioritizes promising regions. Unlike [13], which combines MARCO and MaxSAT to enumerate MUSes for infeasibility analysis, we use Weighted MaxSAT to steer query generation toward informative MSSs under limited resources.

7 Conclusion

In this work, we improve the anytime behavior of CA for precondition inference. We introduce CANDY, which leverages the Δ QUERYGEN heuristic to rapidly generate positive queries and generalize the $\underline{T}$ network. Unlike DCA and PRECA, CANDY delivers high-quality $\underline{T}$ approximations early: it reaches coverage levels up to $16\times$ faster, while reducing the average waiting time between positive queries by a factor of 2 to 3 on the hardest benchmarks. MSS-guided reasoning and query prioritization thus reconcile correctness with practical efficiency.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* **75**(2) (1987)
2. Angluin, D.: Queries revisited. *Theoretical Computer Science* **313** (Feb 2004)
3. Belaid, M., Belmecheri, N., Gotlieb, A., Lazaar, N., Spieker, H.: GEQCA: generic qualitative constraint acquisition. In: *Proc. of the AAAI Conference on Artificial Intelligence*. vol. 36. AAAI Press (2022)
4. Berre, D.L., Parrain, A.: The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation* **7** (2010)
5. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Walsh, T.: Constraint acquisition via partial queries. In: Rossi, F. (ed.) *IJCAI 2013, Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence*, Beijing, China, August 3-9, 2013 (2013)
6. Bessiere, C., Koriche, F., Lazaar, N., O’Sullivan, B.: Constraint Acquisition. *Artificial Intelligence* **244** (2017)
7. Freuder, E.C., Wallace, R.J.: Suggestion strategies for constraint-based match-maker agents. In: Freuder, E.C. (ed.) *Constraints & Agents: Collected Papers from the 1997 AAAI Workshop*, Providence, RI, USA, July 27, 1997. AAAI Press (1997)
8. Girol, G., Lacombe, G., Bardin, S.: Quantitative robustness for vulnerability assessment. In: *Proc. of the ACM on Programming Languages, Design and Implem. (PLDI)*. vol. 8 (Jun 2024)
9. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: *Handbook of Satisfiability - Second Edition* (2021)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of ACM* **12** (1969)
11. Lacombe, G., Bardin, S.: Attacker control and bug prioritization. In: *SEC ’25: Proceedings of the 34th USENIX Conference on Security Symposium (USENIX 2025)*. No. 234 (2025)
12. Liffiton, M.H., Malik, A.: Enumerating infeasibility: Finding multiple muses quickly. In: Gomes, C., Sellmann, M. (eds.) *Proc. of CPAIOR’2013: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer Berlin Heidelberg (2013)
13. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. *Constraints Journal* **21** (2016)
14. Menguy, G., Bardin, S., Gotlieb, A., Lazaar, N.: A query-based constraint acquisition approach for enhanced precision in program precondition inference. *Journal of Artificial Intelligence Research (JAIR)* **82** (2025)
15. Menguy, G., Bardin, S., Lazaar, N., Gotlieb, A.: Active Disjunctive Constraint Acquisition. In: *Proc. of the 20th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR’23)* (8 2023)
16. Menguy, G., Bardin, S., Lazaar, N., Gotlieb, A.: Automated program analysis: Revisiting precondition inference through constraint acquisition. In: Raedt, L.D. (ed.) *Proc. of the 31st Int. Joint Conf. on Artificial Intelligence, IJCAI-22* (7 2022)
17. Prud’homme, C., Fages, J.: CHOCO-solver: A java library for constraint programming. *Journal of Open Source Software* **7** (2022)
18. Zilberstein, S.: Using anytime algorithms in intelligent systems. *AI Magazine* **17**(3) (1996)