Paralleling and Accelerating Arc Consistency Enforcement with Recurrent Tensor Computations

Abstract. We propose a new arc consistency enforcement paradigm that transforms arc consistency enforcement into recurrent tensor operations. In each iteration of the recurrence, all involved processes can be fully parallelized with tensor operations. And the number of iterations is quite small. Based on these benefits, the resulting algorithm fully leverages the power of parallelization and GPU, and therefore is extremely efficient on large and densely connected constraint networks.

Keywords: Arc consistency · Tensor Computation · Parallelization · CSP.

1 Introduction

In most existing architectures, arc consistency enforcement is designed as a sequential process on the constraint network, where the enforcement is first conducted on the current local parts of the network and then is propagated to the connected distant parts and so on. This paradigm of arc consistency enforcement generally requires a *propagation* queue and a *revision* process, where the propagation queue maintains a list of variables or constraints to be revised and the revision fetches elements and removes the involved values that violate the arc consistency rule.

This paradigm can be optimized to be much more incremental that remove many redundant operations between the two consecutive enforcements. But the *propagation* manner is inherently sequential and can be difficult to parallelize. The drawback is obvious. On large or densely connected constraint networks, the chain of propagation conducted on the entire network can be quite long and time-consuming.

To overcome this issue, we propose a new paradigm that transforms the traditional sequential propagation-based arc consistency enforcement into a group of parallelized tensor operations. We first reformalize the arc consistency enforcement as a recurrent process and then prove how the results of this recurrent process are equivalent to traditional arc consistency enforcement. The benefit of this new paradigm is that in each iteration of the recurrent process, the involved operations are independent and can be fully parallelized with tensor operations. And the number of recurrent steps is dramatically less in comparison to the number of propagation steps in the traditional paradigm. As a result, the proposed algorithm based on this paradigm can fully leverage the power of parallel computing and can be extremely efficient on large or densely connected constraint networks.

2 Notation

$dom(x)$ denotes the domain of the variable x. The tuple (x, a) denotes the assignment $x = a$. Then the domain of all variables in a CSP can be represented as $D = \{(x, a)|x \in Vars, a \in dom(x)\}$, where $Vars$ is the set of all variables. $c_{xy}|_{(x,a)}$ denotes the set of all supports of (x, a) on the constraint c_{xy}, i.e. $c_{xy}|_{(x,a)} = \{\tau[y]|\tau \in rel(c_{xy}) \wedge \tau[x] = a\}$. C_x denotes the set of all constraints involving the variable x.

3 Recurrent Arc Consistency (RAC) Enforcement

Given a CSP with the domain of all variables D as defined previously, according to the definition of arc consistency, for any $D_{\bar{ac}} \subseteq D$,

$$\begin{aligned}
&D_{\bar{ac}} \text{ is arc consistent} \\
\Leftrightarrow &\forall (x, a) \in D_{\bar{ac}}((x, a) \text{ is arc consistent}) \\
\Leftrightarrow &\forall (x, a) \in D_{\bar{ac}} \forall c_{xy} \in C_x((x, a) \text{ has valid supports on } c_{xy}) \\
\Leftrightarrow &\forall (x, a) \in D_{\bar{ac}} \forall c_{xy} \in C_x(c_{xy}|_{(x,a)} \cap D_{\bar{ac}} \neq \emptyset).
\end{aligned}$$

Generally, there can be more than one $D_{\bar{ac}} \subseteq D$ that is arc consistent for a CSP. Let $D_{\bar{AC}}$ be the set of all $D_{\bar{ac}}$ that is arc consistent and $D_{ac} = \bigcup_{D_{\bar{ac}} \in D_{\bar{AC}}} D_{\bar{ac}}$. Then, $D_{ac} \in D_{\bar{AC}}$, and D_{ac} is the result of the arc consistency enforcement. There are many algorithms proposed to compute D_{ac}. In contrast with all these algorithms, we reformulate the computation of D_{ac} (arc consistency enforcement) with completely recurrent tensor operations, which can fully leverage the power of parallel computations.

Let $D_{\widetilde{ac}} = D \backslash D_{ac}$. According to the definition of arc consistency, we have

Lemma 1. *For any $(x, a) \in D$, if there exists $c_{xy} \in C_x$ and $D'_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$ such that $c_{xy}|_{(x,a)} \subseteq D'_{\widetilde{ac}}$, then $(x, a) \in D_{\widetilde{ac}}$.*

We prove Lemma 1 in Appendix A. Let $D''_{\widetilde{ac}} = D'_{\widetilde{ac}} \cup \{(x, a)\}$. Then $D''_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$ as $D'_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$ and $(x, a) \in D_{\widetilde{ac}}$. By setting $D'_{\widetilde{ac}} = D''_{\widetilde{ac}}$, we can iteratively enlarge the set $D'_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$. Based on this idea, we design the recurrent way of collecting (x, a) with no valid support as follows.

$$\begin{cases} D^{(0)}_{\widetilde{ac}} = \emptyset, k = 0 \\ D^{(k)}_{\widetilde{ac}} = D^{(k-1)}_{\widetilde{ac}} \cup \{(x, a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\}, k \in \mathbb{Z}_+. \end{cases} \tag{1}$$

The collection of $\{(x, a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\}$ in each iteration can be potentially parallelized. Next, we show that the recurrence in Equation 1 will always end with $D^{(K)}_{\widetilde{ac}} = D_{\widetilde{ac}}$.

Proposition 1. *According to the computation of $D^{(k)}_{\widetilde{ac}}$ in Eq.1,*

1. $\forall k \in \mathbb{Z}_+(D^{(k)}_{\widetilde{ac}} \subseteq D_{\widetilde{ac}})$;

2. *there always exists* K *such that,*
   (a) $\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(K)}_{\widetilde{ac}}\}\backslash D^{(K)}_{\widetilde{ac}} = \emptyset$;
   (b) $D^{(K)}_{\widetilde{ac}} = D_{\widetilde{ac}}$.

We prove Proposition 1 in Appendix B. Therefore, Eq.1 acts as an iterative manner to enforce arc consistency. When $D^{(K)}_{\widetilde{ac}} = D^{(K+1)}_{\widetilde{ac}}$, the iteration ends with $D_{ac} = D\backslash D^{(K)}_{\widetilde{ac}}$. More interestingly, Eq.1 indicates the availability of paralleling the arc consistency enforcement as we can collect the element in $\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\}$ in each iteration simultaneously. We make the best use of parallelization by designing tensor operations.

Before that, there is an interesting insight that indicates a possible optimization of increment among iterations of the recurrent process.

Proposition 2. *Let* $V^{(k)}_{\widetilde{ac}} = D^{(k)}_{\widetilde{ac}}\backslash D^{(k-1)}_{\widetilde{ac}}$, *then*

1. $\forall (x,a) \in V^{(k)}_{\widetilde{ac}} \forall c_{xy} \in C_x (c_{xy}|_{(x,a)} \backslash D^{(k-2)}_{\widetilde{ac}} \neq \emptyset)$;
2. $\forall (x,a) \in V^{(k)}_{\widetilde{ac}} \exists c_{xy} \in C_x (c_{xy}|_{(x,a)} \backslash D^{(k-2)}_{\widetilde{ac}} \subseteq V^{(k-1)}_{\widetilde{ac}})$.

We prove Proposition 2 in Appendix C. Proposition 2 shows that all (x,a) to be removed in the current iteration are caused by the loss of the supports removed in the previous iteration. So, we can make the whole enforcement process as in Equation 1 incremental by maintaining the set of variables with a changed domain and then only checking and removing inconsistent (x,a) by only testing variables in this set.

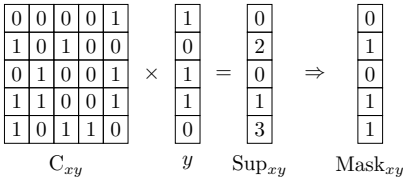

Fig. 1. A variable y is represented as a 1d array indexed by the values in $dom(y)$. In this array, $y[a] = 1$ represents y has the value a, and $y[a] = 0$ represents not. Similarly, a constraint C_{xy} is represented as a 2d array. $Sup_{xy}[a]$ represents the number of collected supports of (x,a) on the constraint C_{xy}.

4 RAC Enforcement with Tensor Accelerating (RTAC)

In this section, we design our algorithm based on the recurrent arc consistency paradigm and accelerating each iteration, i.e. collecting $\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\}$, with tensor parallelization.

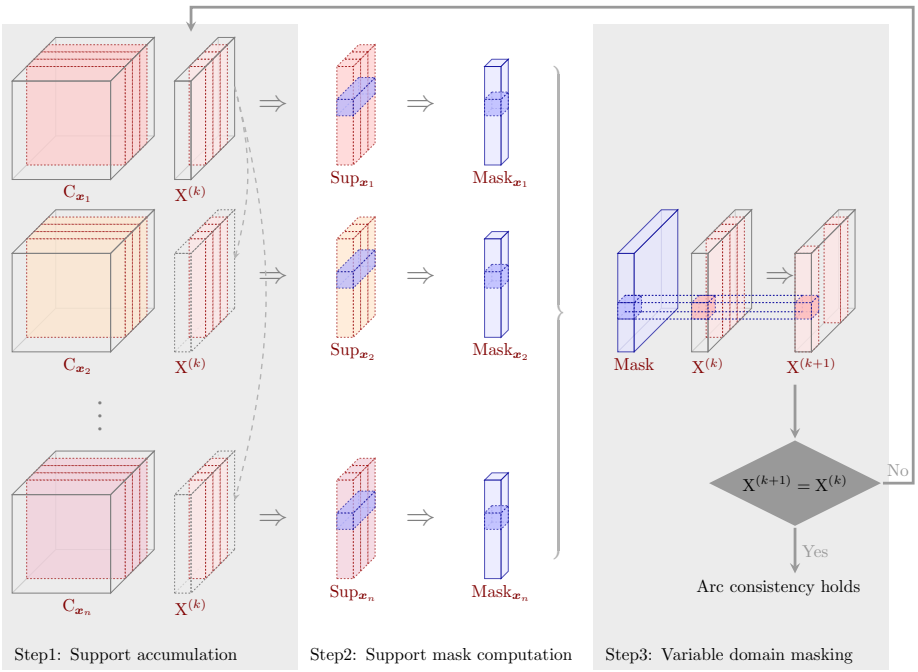

Fig. 2. The illustration of arc consistency enforcement with tensor parallzation.

We first introduce the basic idea of support collection with tensor computation with the simplest single constraint case as in Figure 1. Then, the whole support collecting on all constraints and variables is achieved by stacking multi-dimensional tensors and can be computed simultaneously as Step 1 in Figure 2. The whole recurrent tensor arc consistency (RTAC) enforcement pipeline is as Figure 2, and the corresponding pseudocode is as Algorithm 1. Algorithm 1 only relies on a small group of basic tensor operations as follows where we use T to represent a given tensor:

- T.**sum**(*dim*): Returns the sum of each row of T in the given dimension *dim*.
- T.**any**(): Tests if any element in T evaluates to True.
- T.**nonzero**(): Return the indices of all non-zero elements of T.
- T.**dim_expand**(*dim*): Returns a tensor with a dimension of size one inserted to T at the specified *dim*.
- T.**dim_reduct**(*dim*): Returns a tensor with the specified *dim* of size one of T removed.
- **where**(*condition*, x, y): Return a tensor of elements selected from either x or y, depending on *condition*.

These operations are all well provided in most tensor computing/deep learning frameworks e.g., Pytorch, Jax, TensorFlow, Numpy, etc. For ease of implementation, we build our program upon the popular deep learning framework

PyTorch with the sacrifice of some computation efficiency. The actual amount of code is quite little thanks to the well-developed tensor computing frameworks. We give all codes in the Appendix E.

```
Def tensorAC(Vars, @changed):
    #Vals^pre = Vars.sum(1);
    while |@changed| ≠ 0 do
        Vars=tensorRevise(Vars, @changed);
        #Vals = Vars.sum(1);
        if (#Vals == zero^n).any() then
            throw inconsistency;
        @changed = (#Vals ≠ #Vals^pre).nonzero();
        #Vals^pre = #Vals;
    return Vars;
Def tensorRevise(Vars, @changed):
    Cons = Cons[*, @changed, *, *];
    Vars = Vars[@changed, *].dim_expand(2);
    supp = (Cons × Vars).dim_reduct(-1);
    supp = where(supp > 1, one^nnd[*, 0: |@changed|, *], supp);
    Vars = where(supp.sum(1) ≠ |@changed|, zero^nd, Vars);
    return Vars;
```

Algorithm 1: Arc consistency enforcement

5 Experiments

5.1 Configuration

To ensure a fair comparison as much as possible, we implement the state-of-the-art sequential arc consistency enforcement algorithm AC3 with Python + JIT since the pure Python program is slow. Then, we implement RTAC with Python + PyTorch. For the hardware part, we use CPU: I9-10900K and GPU: RTX3090.

5.2 Benchmark

We use the randomly generated binary constraint satisfaction problems since they can easily be generated with different scales, i.e. different number of variables, constraints, etc, to be used to conduct our ablation studies. The constraint network topology is generated randomly with manually setting constraint density. Specifically, for a number of n variables and a given constraint density d. There will be $\frac{n\times(n-1)}{2}$ pair of variables, and each pair of them is assigned with a constraint with the possibility of d. We generate a total of 25 random

CSPs with the number of variables $\{100, 250, 500, 750, 1000\}$ and the densities $\{0.1, 0.25, 0.5, 0.75, 1.0\}$.

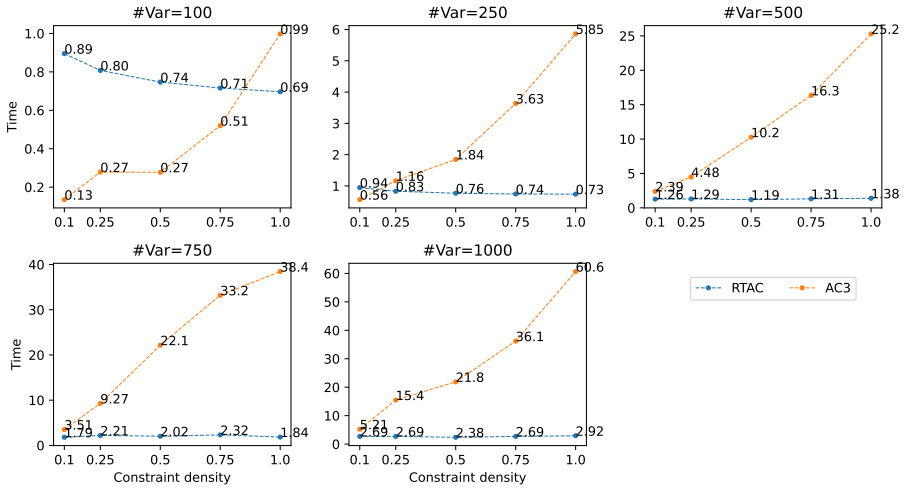

Fig. 3. Running time (ms) of one assignment in backtrack search. The results are an average of 50K assignments.

5.3 Result

Comparing the efficiency of our proposed RTAC and the traditional sequential algorithms can be tricky for the following reasons: First, the arc consistency enforcements of two algorithms do not run on the same device. The former runs on GPU and the latter runs on CPU. Thus their efficiency is partially decided by the performance of the hardware. Second, although RTAC can be built from scratch to ensure the best efficiency at every detailed implementation, we built it upon the deep learning framework for ease of implementation. Therefore its performance is mostly decided by the efficiency of the selected framework.

As noticed in the above concerns, the comparisons between RTAC and AC3 can be less rigorous. But the empirical results in Fig.3 and Tab.1 indicate the following two guarantees: First, the increase of time-consuming of RTAC is even unnoticeable when increasing the number of variables or the density of constraints; Second, RTAC has great potential on large and densely connected CSPs.

6 Conclusion

We propose a new arc consistency enforcement paradigm and theoretically prove the equivalence of its results with the definition of arc consistency. The induced

Table 1. A statistics of the number of revisions (denoted by #Revision) in AC3 and the number of recurrences (denoted by #Recurrence) in RTAC. The results are an average of 50K assignments.

#Variable	Density	#Revision	#Recurrence
100	0.10	307.6	4.509
100	0.25	626.4	4.103
100	0.50	965.2	3.752
100	0.75	1612.4	3.573
100	1.00	2714.1	3.462
250	0.10	1152.0	4.804
250	0.25	2532.6	4.167
250	0.50	4629.6	3.794
250	0.75	7881.9	3.617
250	1.00	12405.6	3.441
500	0.10	3250.9	4.620
500	0.25	7619.8	4.126
500	0.50	18793.8	3.952
500	0.75	28218.4	3.728
500	1.00	42557.7	3.455
750	0.10	6195.7	4.766
750	0.25	13768.6	4.020
750	0.50	36220.6	3.940
750	0.75	61171.7	3.703
750	1.00	71509.8	3.597
1000	0.10	8322.2	4.831
1000	0.25	24544.3	4.381
1000	0.50	39707.7	4.048
1000	0.75	65446.2	3.755
1000	1.00	107680.5	3.556

algorithm RTAC fully leverages the power of parallelization and GPU, showing its efficiency on large and densely connected constraint networks.

A Proof of Lemma 1

Proof. $c_{xy}|_{(x,a)} \subseteq D'_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$, So $c_{xy}|_{(x,a)} \cap D_{ac} = \emptyset$, which means (x,a) has no valid support on c_{xy}, and therefore (x,a) is not arc consistent. Hence, $(x,a) \notin D_{ac}$ and $(x,a) \in D_{\widetilde{ac}}$.

B Proof of Proposition 1

Proof. (1) When $k = 0, D^{(0)}_{\widetilde{ac}} = \emptyset \subseteq D_{\widetilde{ac}}$. Suppose when $k = l$, $D^{(l)}_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$. According to Prop.1, $\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(l)}_{\widetilde{ac}}\} \subseteq D_{\widetilde{ac}}$. Then, when $k = l+1$, $D^{(l+1)}_{\widetilde{ac}} = D^{(l)}_{\widetilde{ac}} \cup \{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(l)}_{\widetilde{ac}}\} \subseteq D_{ac}$. So, $\forall k \in \mathbb{Z}_+, D^{(k)}_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$.

(2.a) If there is no such K, in other words, for any $k \in \mathbb{Z}_+$, $\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k)}_{\widetilde{ac}}\} \backslash D^{(k)}_{\widetilde{ac}} \neq \emptyset$, then $D^{(0)}_{\widetilde{ac}} \subset D^{(1)}_{\widetilde{ac}} \subset \cdots \subset D^{(+\infty)}_{\widetilde{ac}}$. So $\lim_{k\to+\infty} |D^{(k)}_{\widetilde{ac}}| = +\infty$, which is inconsistent with the fact that $D^{(k)}_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$, thus reaching conflicts.

(2.b) As $D^{(k)}_{\widetilde{ac}} \subseteq D_{\widetilde{ac}}$ for any k, we have

$$
\begin{aligned}
&\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(K)}_{\widetilde{ac}}\} \backslash D^{(K)}_{\widetilde{ac}} = \emptyset \\
\Leftrightarrow &\{(x,a)|\exists y, c_{xy}|_{(x,a)} \subseteq D^{(K)}_{\widetilde{ac}}\} \subseteq D^{(K)}_{\widetilde{ac}} \\
\Leftrightarrow &\forall (x,a)(\exists c_{xy} \in C_x(c_{xy}|_{(x,a)} \subseteq D^{(K)}_{\widetilde{ac}}) \rightarrow (x,a) \in D^{(K)}_{\widetilde{ac}}) \\
\Leftrightarrow &\forall (x,a)((x,a) \in (D \backslash D^{(K)}_{\widetilde{ac}}) \rightarrow \forall c_{xy} \in C_x(c_{xy}|_{(x,a)} \cap (D \backslash D^{(K)}_{\widetilde{ac}}) \neq \emptyset)) \\
\Leftrightarrow &\forall (x,a) \in D \backslash D^{(K)}_{\widetilde{ac}} \forall c_{xy} \in C_x((x,a) \text{ has valid supports on } c_{xy}) \\
\Leftrightarrow &D \backslash D^{(K)}_{\widetilde{ac}} \text{ is arc consistent} \\
\Rightarrow &D \backslash D^{(K)}_{\widetilde{ac}} \subseteq D_{ac} = D \backslash D_{\widetilde{ac}} \\
\Leftrightarrow &D_{\widetilde{ac}} \subseteq D^{(K)}_{\widetilde{ac}} \\
\Leftrightarrow &D_{\widetilde{ac}} = D^{(K)}_{\widetilde{ac}}
\end{aligned}
$$

C Proof of Proposition 2

Proof. (1) For any $(x,a) \in V^{(k)}_{\widetilde{ac}}$, if there exists $c_{xy} \in C_x$ with $c_{xy}|_{(x,a)} \backslash D^{(k-2)}_{\widetilde{ac}} = \emptyset$, aka, $\exists y, c_{xy}|_{(x,a)} \subseteq D^{(k-2)}_{\widetilde{ac}}$, then $(x,a) \in D^{(k-1)}_{\widetilde{ac}}$ according to Eq.1, thus reaching conflicts.

(2)

$$
\begin{aligned}
V^{(k)}_{\widetilde{ac}} &= D^{(k)}_{\widetilde{ac}} \backslash D^{(k-1)}_{\widetilde{ac}} \\
&= D^{(k-1)}_{\widetilde{ac}} \cup \{(x,a)|\exists c_{xy} \in C_x, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\} \backslash D^{(k-1)}_{\widetilde{ac}} \\
&\subseteq \{(x,a)|\exists c_{xy} \in C_x, c_{xy}|_{(x,a)} \subseteq D^{(k-1)}_{\widetilde{ac}}\} \\
&= \{(x,a)|\exists c_{xy} \in C_x, c_{xy}|_{(x,a)} \subseteq D^{(k-2)}_{\widetilde{ac}} \cup V^{(k-1)}_{\widetilde{ac}}\} \\
&= \{(x,a)|\exists c_{xy} \in C_x, c_{xy}|_{(x,a)} \backslash D^{(k-2)}_{\widetilde{ac}} \subseteq V^{(k-1)}_{\widetilde{ac}}\}
\end{aligned}
$$

Hence, $\forall(x,a) \in V_{\widetilde{ac}}^{(k)} \exists c_{xy} \in C_x(c_{xy}|_{(x,a)} \backslash D_{\widetilde{ac}}^{(k-2)} \subseteq V_{\widetilde{ac}}^{(k-1)})$.

D Pseudocode of Backtrack search

```
1  Def main():
2      init();
3      tensorAC(Vars, [0 : |Vars|]);
4      dfs(0, Vars);
5  Def dfs(level, Vars):
6      if level == n then
7          find answer;
8      idx = heuristics();
9      for Val ∈ Var[idx].nonzero() do
10         Vars = assign(idx, Val, Vars);
11         Vars = tensorAC(Vars, [idx]);
12         if dfs (level+1, Vars) then
13             return True;
14     return False;
15 Def init():
16     Prepare Cons ∈ {0,1}^{n×n×d×d};
17     Prepare Vars ∈ {0,1}^{n×d};
18     Prepare zero^n ∈ 0^n;
19     Prepare zero^{nd} ∈ 0^{n×d};
20     Prepare one^{nnd} ∈ 1^{n×n×d};
21     Prepare I^{nn} = Identity matrix;
22 Def assign(idx, Val, Vars):
23     I^{nn}[idx][idx] = 0;
24     Vars = I^{nn} × Vars;
25     I^{nn}[idx][idx] = 1;
26     Vars[idx][Val] = 1;
27     return Vars;
```

Algorithm 2: Backtrack search

E Source Code of RTAC

The amount of source code of RTAC is dramatically small when implemented upon the deep learning libraries. Below is the code of RTAC built upon PyTorch.

```
import torch

class ACEnforcer:
    def __init__(self, cons_map, n_vars, n_dom):
        self.cons_map = cons_map
        self.n_mask0 = torch.zeros(n_vars).to(device)
        self.nnd_mask1 = torch.ones((n_vars, n_vars, n_dom))
            .to(device)
        self.nd_mask0 = torch.zeros((n_vars, n_dom)).to(
            device)

    def ac_enforcer(self, vars_map, changed_idx):
        n_idx = changed_idx.shape[0]
        vars_map_pre = vars_map.sum(1)
        while n_idx != 0:
            nkd = torch.matmul(self.cons_map[:, changed_idx,
                :, :], vars_map[changed_idx, :].unsqueeze
                (2)).squeeze(-1)
            nd = torch.where(nkd > 1, self.nnd_mask1[:, :
                n_idx, :], nkd).sum(1)
            vars_map = torch.where(nd != n_idx, self.
                nd_mask0, vars_map)
            vars_map_sum = vars_map.sum(1)
            if (vars_map_sum == self.n_mask0).any():
                return None
            changed_idx = (vars_map_sum != vars_map_pre).
                nonzero(as_tuple=True)[0]
            n_idx = changed_idx.shape[0]
            vars_map_pre = vars_map_sum
        return vars_map
```

Listing 1.1. The code of RTAC implemented with PyTorch.