BASIC COMPUTING OF SOLUTION DENSITY

Reporter: Jinling Xing

April 17, 2018

Polytechnique Montréal

Outlines

1 INTRODUCTION

Introduction

Basic computing of **Solution density**, the number of supports & currTable for each pair of (x, a) in the compactTable constraint of Oscar.

2 COMPACT TABLE

2.1 currTable

T	x	y	z
0	a	a	a
1	a	a	b
2	a	b	c
3	b	a	a
	a	c	b
4	a	b	b
5	b	a	b
6	b	b	a
7	b	b	b

(a) The indexed tuples

currTable	1	1	1	1	1	1	1	1
supports$[x, a]$	1	1	1	0	1	0	0	0
supports$[x, b]$	0	0	0	1	0	1	1	1
supports$[y, a]$	1	1	0	1	0	1	0	0
supports$[y, b]$	0	0	1	0	1	0	1	1
supports$[y, d]$	0	0	0	0	0	0	0	0
supports$[z, a]$	1	0	0	1	0	0	1	0
supports$[z, b]$	0	1	0	0	1	1	0	1
supports$[z, c]$	0	0	1	0	0	0	0	0

(b) The corresponding bit-sets

Fig. 2: Illustration of the data structures after the initialization of $\langle x, y, z \rangle \in T$. The tuple (a, c, b) will not be indexed and d will be removed from $dom(y)$.

For example, the solution density of **supports[x,a] & currTable** is the **number of ones** after computing, at this case the result is 4.

2.2 Algorithm of simplified version of CT

Algorithm 1. Class ConstraintCT

```
Method updateTable()
    foreach variable x ∈ scp do
        mask ← 0
        if |Δx| < |dom(x)| then                              // Incremental Update
            foreach value a ∈ Δx do
                mask ← mask | supports[x, a]    // Use supports*[x, a] in CT*
            mask ← ~mask                                      // ~: bitwise negation
        else                                                  // Reset-based Update
            foreach value a ∈ dom(x) do
                mask ← mask | supports[x, a]                  // | : bitwise or
        currTable ← currTable & mask                          // & : bitwise and
Method filterDomains()
    foreach variable x ∈ scp do
        foreach value a ∈ dom(x) do
            if currTable & supports[x, a] = 0 then
                dom(x) ← dom(x) \ {a}
Method enforceGAC()
    updateTable()
    filterDomains()
```

Modify the **Method** *filterDomains()* and use the **Method** *nbOfOnes()*

3 CT WITH SOLUTION DENSITY

Count the number of elements in intersection between currTable and bs(bitset to intersect) and return number of elements in intersection.

```
/**
  * count the number of ones of supports and currTable
  */
def nbOfOnes(bs: BitSet): Int = {
  var count = 0
  var i : Int = nNonZero /*bit length of nWords*/
  while(i >0){
    i -= 1
    val offset = nonZeroIdx(i)
    count += java.lang.Long.bitCount(words(offset) & bs.words(offset))
    }
  count
}
```

sd is a 2-dimension array: Array[Array[]], I used ArrayBuffer to store the counts in sd.

```
//sd is the solution density
val sd = Array.tabulate(x.length)(i => Array.tabulate(spans(i))(v => new ArrayBuffer[Int]()))

/* Create the final support bitSets and remove any value that is not supported */
for {
  varIndex <- variableValueSupports.indices
  valueIndex <- variableValueSupports(varIndex).indices
} {
  if (varValueSupports(varIndex)(valueIndex).nonEmpty) {
    variableValueSupports(varIndex)(valueIndex) = new
        validTuples.BitSet(varValueSupports(varIndex)(valueIndex))
    val count = validTuples.nbOfOnes(variableValueSupports(varIndex)(valueIndex))

    sd(varIndex)(valueIndex) += count //sd is the solution density
    println(" var index " + varIndex + " value " + valueIndex + " count " + count )
  }else {
    /* This variable-value does not have any support, it can be removed */
    x(varIndex).removeValue(valueIndex)
  }
}
```

Find the max count value of each variable. We can get a 3*6 2-dimension array. Variables: val x = Array.fill(5)(CPIntVar(0 to 5)). When the size of the sd>=1, we will escape the empty ArrayBuffer.

```
// Print two dimensional array and find the max
for {
  varIndex <- variableValueSupports.indices
} {
  var maxOfCount = 0
  for {valueIndex <- variableValueSupports(varIndex).indices} {

    if(sd(varIndex)(valueIndex).size >= 1) {
      val c = sd(varIndex)(valueIndex).head
      //print(" " + c )
      maxOfCount = java.lang.Math.max(maxOfCount, c)
      print(" " + sd(varIndex)(valueIndex))
    }
  }
  print(" Max " + maxOfCount + "\n")
}
```

4. EXPERIMENT

The origin table is on the left, which is a Array[Array[]] format table, including 3 variables and the domain of each variable is between 0-5. The output of count is on the right, for each X[i] and the value in its domain.

```
val tableGround: Array[Array[Int]] = Array(
  Array(1, 2, 3),
  Array(2, 3, 4),
  Array(1, 1, 1),
  Array(0, 1, 5),
  Array(1, 2, 5),
  Array(5, 5, 5)
)
```

Figure: Origin table

```
var index 0 value 0 count 1
var index 0 value 1 count 3
var index 0 value 2 count 1
var index 0 value 5 count 1
var index 1 value 1 count 2
var index 1 value 2 count 2
var index 1 value 3 count 1
var index 1 value 5 count 1
var index 2 value 1 count 1
var index 2 value 3 count 1
var index 2 value 4 count 1
var index 2 value 5 count 3
```

Figure: The count of each (x,a)

Table: The count Array[Array[]] format matrix

Variable	Count				MaxCount
X[0]	1	3	1	1	3
X[1]	2	2	1	1	2
X[2]	1	1	1	3	3

```
ArrayBuffer(1) ArrayBuffer(3) ArrayBuffer(1) ArrayBuffer(1) Max 3
ArrayBuffer(2) ArrayBuffer(2) ArrayBuffer(1) ArrayBuffer(1) Max 2
ArrayBuffer(1) ArrayBuffer(1) ArrayBuffer(1) ArrayBuffer(3) Max 3
```

5. Conclusion

- In this project, I want to compute the basic **solution density**, the number of ones in of each pair(x,a) after calculating currTable & Supports(x,a).
- The modification of **Method** *filterDomains()* and use the **Method** *nbOfOnes()* to count the number of ones in currTable & Supports(x,a).
- Construct the 2-dimension array **"sd"** to store the **"counts"** and find the **"max"** of the count.
- Future work: Based on the maximum value and its index to modify the **search**.

Questions?