Modeling Sparse Matrices

ACM Reference Format:
. 2019. Modeling Sparse Matrices. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MODELING SPARSE MATRIX FORMAT TRANSLATIONS

Translating a sparse matrix from one format to another may be necessary in a number of scenarios. For example, it is common practice to assemble a sparse matrix using a format that is efficient for incremental assembly and to translate the fully assembled matrix to a format that is efficient for use in a solver. In the context of ADCIRC++, we wish to compare the performance of a number of solvers in a number of different storm surge scenarios. Because each solver requires a different sparse matrix format, we must show that the same matrix is being used across solvers, regardless of format.

ADCIRC and ADCIRC++ both use the ELL format in conjuction with the ITPACKV solver. The format most commonly used in the solvers we wish to test (TODO: ITPACK, Pardiso, etc.) is the Compressed Sparse Row (CSR) format, and so we will model the ELL to CSR translation in this section. A similar approach is to be taken for other formats, including the Block Compressed Sparse Row (BSR) used in the cuSPARSE library, and the Compressed Sparse Column (CSC) format used in the Armadillo library.

```
1 kpos = 0
2 for i in range(nrows):
3     for k in range(maxnz):
4         idx = i * maxnz + k
5         if cols[idx] != -1:
6             A[kpos] = vals[idx]
7             JA[kpos] = cols[idx]
8             kpos = kpos + 1
9     IA[i + 1] = kpos
```

Algorithm 1: The ELL to CSR translation algorithm.

An ELL to CSR translation algorithm is provided by the SPARSEKIT library, and a Python version of this algorithm is found in Figure 1.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
pred genkpos [e: ELL, kpos: seq Int] {
    kpos[0] = 0
    #kpos = e.rows.mul[e.maxnz].add[1]
    all i: rowInds[e] {
        all k: indices[e.maxnz] {
            let it = i.mul[e.maxnz].add[k]{
                rowcols[e, i][k] != -1 ⇒ {
                    it < kpos.lastIdx ⇒
                        kpos[it.add[1]] = kpos[it].add[1]
                } else {
                    it < kpos.lastIdx ⇒
                        kpos[it.add[1]] = kpos[it]
                }
            }
        }
    }
}
pred ellcsr [e: ELL, c: CSR] {
    e.rows = c.rows
    e.cols = c.cols
    c.IA[0] = 0
    #c.IA = c.rows.add[1]
    some kpos: seq Int{
        genkpos[e, kpos]
        #c.A = kpos.last
        #c.JA = kpos.last
        all i: rowInds[e] {
            all k: indices[e.maxnz] {
                rowcols[e, i][k] != -1 ⇒
                let kp = kpos[i.mul[e.maxnz].add[k]] {
                    c.A[kp] = rowvals[e, i][k]
                    c.JA[kp] = rowcols[e, i][k]
                }
            }
            c.IA[i.add[1]] = kpos[i.add[1].mul[e.maxnz]]
        }
    }
}
assert transValid {
    all e: ELL, c: CSR, m: Matrix |
        repInv[e] and alpha[e, m] and ellcsr[e, c] ⇒
            repInv[c] and alpha[c, m]
}
check transValid
    for 4 Int, 7 seq, 1 Matrix, 1 ELL, 1 CSR, 2 Value
```

Figure 1: The ELL to CSR Model

The translation from ELL to CSR in Alloy is performed by quantifying the state of the algorithm at every step in the innermost loop. As seen in Algorithm 1, there are three variables used for indexing, i, k, and $kpos$, that need to be quantified. The values of i and k are easily generated in Alloy using an **all** statement, but the value of *kpos* is less straightforward. It can optionally increase by one on line 8, at the end of the innermost loop, depending on whether or not the current matrix value is nonzero. As a result, the value of *kpos* at line 9, where an IA value is set, may not be the same as the value of *kpos* at lines 6 and 7, where values of A and JA are set. Therefore, we model the value of *kpos* as a sequence of integers in which the value at index it is the value of *kpos* before line 8 and the value at index $it + 1$ is the value of *kpos* after line 8. This sequence is generated using the **genkpos** predicate found in Figure 1. With the sequence generated, the value of *kpos* at any location inside of the algorithm can now be determined using the values of i and k.

The **ellcsr** predicate generates a sequence of *kpos* values using the **genkpos** predicate[1] and performs the translation using the same indexing pattern used in the algorithm. Indeed, the value of *kpos* could be generated directly within the **ellcsr** predicate, but we found that the separation made it easier to reason about two key components separately: (1) the nondeterministic evolution of the *kpos* value and (2) the correct indexing of values.

For a translation to be considered valid, the final state must preserve the representation invariant of the target format as well as represent the same abstract matrix as the original format. Intermediate states, however, do not need to preserve the representation invariant. The Alloy assertion used to model this relationship is shown in Figure 1, which states that if the ELL matrix e is a valid representation of the abstract matrix m and the translation is applied, giving the CSR matrix c, then it follows that c is a valid CSR matrix that represents the abstract matrix m. The scope used for this check shows that the translation is valid for matrices up to 6x6.

REFERENCES

[1] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, “Alloy*: A general-purpose higher-order relational constraint solver,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 609–619.

[1]The *kpos* sequence is generated using **some**, which here is a higher-order quantification. Alloy cannot perform higher-order quantification, but is able to eliminate it in some cases, such as this one, by using skolemization. We expect there to be only a single possible sequence of *kpos* values for any given translation, and so the **one** quantifier should be used instead. This quantification cannot be eliminated using skolemization, and so we have verified this property using Alloy*, an extension of Alloy that permits higher-order quantification [1].