Translation Sections

ACM Reference Format:
. 2019. Translation Sections. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 3 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. Due to differences in solution technique in combination with floating point error propagation, the results generated by various solvers may not be exactly equivalent. Therefore, in order to ensure that the same problem is being solved in each test case, a comparison of solutions is not sufficient. Rather, we must show that the same matrix–i.e., the same system of equations–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.

An ELL to CSR translation algorithm is provided by the SPARSEKIT [1] library, and a Pythonic listing of this algorithm is found in Figure 1. The algorithm works by iterating over each stored column of each row in the ELL data structure. If it is non-zero, the current value and its column index are copied in to the next available location in the corresponding CSR arrays, and the ending IA index for that row is incremented.

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

```
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.

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

[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 [2].

```
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

this check shows that the translation is valid for matrices up to 6×6.

1.1 In-Place Translation

In SPARSEKIT [1], the ELL to CSR translation is a Fortran subroutine that receives five arrays as input: two arrays, COEF and JCOEF, representing the ELL format, and three arrays, A, JA, and IA, representing the CSR format. The target CSR arrays must be allocated before calling the subroutine, and must contain enough memory to store the sparse matrix. The only way to ensure that there is enough memory allocated is to either count the number of non-zero values in the ELL matrix before calling the subroutine or naively allocate enough memory to hold a densely populated ELL matrix (one in which there are no padding values). This presents two possible performance issues: (1) counting the number of non-zero values requires looping through the entire ELL matrix, and (2) depending on the size and sparsity of the matrix, allocating enough memory to perform the translation may not be possible on certain hardware.

An in-place translation from ELL to CSR could sidestep these issues, provided there is no requirement that the original ELL matrix be valid after the translation occurs. During the translation, all padding values are removed from COEF and JCOEF to produce A and JA, respectively, and so we can infer that in the case where there are no padding values present, the arrays will be equivalent. Therefore, we know that the maximum amount of memory required to store A and JA is equal to the amount required to store COEF and JCOEF. So if an in-place translation is possible, the number of non-zero values is not required and the only memory allocation required is for the IA array, the size of which is always one more than the number of rows in the matrix.

```
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             vals[kpos] = vals[idx]
7             cols[kpos] = cols[idx]
8             kpos = kpos + 1
9     IA[$i + 1$] = kpos
```

Algorithm 2: The in-place ELL to CSR translation algorithm.

Algorithm 2 shows the in-place translation. It is equivalent to Algorithm 1, except for lines 7 and 8 which are modified to perform the translation in-place. In order to determine if this algorithm performs a valid translation, we must show that once a value has been written to an index, that index is not read from in subsequent iterations. In this algorithm, the values used to index into the arrays, *kpos* and *idx*, are always increasing. At every iteration, the value of *idx* increases by exactly one and the value of *kpos* optionally increases by one. Therefore, we can conclude that *kpos* indices being used to overwrite existing values will always be less than or equal to *kpos* indices being used to read existing values. This means that once a value is written to an index, that index will never again be used by the algorithm to read that value, and so an in-place translation is possible.

This property can be formally verified using Alloy, as shown in Figure 2. The assertion used here states that for valid ELL matrices up to size 15×15, the value of *kpos* is less

```
assert inPlace {
    all e: ELL, kpos: seq Int{
        repInv[e] and genkpos[e, kpos] ⇒ {
            all i: rowInds[e] {
                all k: indices[e.maxnz] {
                    let idx = i.mul[e.maxnz].add[k] |
                        kpos[idx] ≤ idx
                }
            }
        }
    }
}
check inPlace
    for 5 Int, 15 seq, 0 Matrix, 1 ELL, 0 CSR, 2 Value
```

Figure 2: Alloy assertion that the ELL to CSR translation can be performed in-place.

than or equal to the value of *idx* at every iteration of the translation algorithm. Indeed, the assertion holds and so we have formally verified that within scope, the algorithm can be performed in-place. While visual inspection and careful reasoning about the algorithm led us to the same conclusion, the benefits of using Alloy here are twofold: (1) a formal verification provides confidence that our reasoning is sound, and (2) this type of modeling gives valuable insight into how to reason about the types of algorithms commonly found in scientific and engineering software, many of which may not be as easy to analyze analytically.

REFERENCES

[1] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[2] 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.