Project 1: Sparse Vectors, Sparse Matrices and modified ELL format

Dinh Thanh Phong Do NOMA: 1360 1700

4th of March, 2022

1 Complexity analysis of `operator+()` between 2 SparseVector

1.1 Time complexity

The time complexity of this operator can be decomposed in 2 parts. The first part consist of finding the length of the new `SparseVector` and the second part consist of copying the value of the `SparseVector& v1`.

In the first part, the implementation involve 2 counters (`count1` and `count2`) which will will help us to iterate in both array `this->rowidx` and `v1.rowidx`. In order to find this length, which is stored inside the variable `max`, we iterate inside a for loop by updating both counters and the variable `max` when both vectors are nonzero in the same row. $\min(nnz_1, nnz_2)$ iterations are needed and update for each iteration are in $\mathcal{O}(1)$ (beacause it consist of scalar assignment and addition operations). Thus, this first part is done in $\mathcal{O}(\min(nnz_1, nnz_2))$.

The second part, which is about copying the values, we need first to allocate memory for our vectors `rowidx` and `nzval` (denoted in the code as `rows` and `values`) which are of size `max`. Assuming allocation is done in $\mathcal{O}(1)$, these instruction are not significant in the analysis.
Afterward, we then update both array and it is done inside a for loop which do `max` iterations. Inside this second loop, only simple operations (assignment, addition, accessing a value in array) can be done in $\mathcal{O}(1)$. Thus this second part is done in $\mathcal{O}(\max)$.
Then, the call to the constructor for a deep copy of `SparseVector` is also done in $\mathcal{O}(\max)$ before the return statement.
To analyse further, we can look at the worst scenario where `max`, which represent the size of both array `rows` and `values`, is maximal. We can observe $\max \leq nnz_1 + nnz_2$ (the equality is hold when all zero rows for both vectors are completely different from each other) and thus, we can say the time complexity is:

$$\mathcal{O}(\max) \leq \mathcal{O}(nnz_1 + nnz_2)$$

1.2 Space complexity

The space complexity of this method can be determined by analysing allocated memory. The allocation of integer variable (ex: `count1` and `count2`) is negligeable compared to vectors thata are computed inside this function. Indeed, we allocate (and then free) 2 vectors (`rows` and `values`) of size lesser or equal to $(nnz_1 + nnz_2)$.
Thus, the space complexity is:

$$\mathcal{O}(nnz_1 + nnz_2)$$

2 Complexity analysis of operator*() between a SparseMatrix and a Vector

2.1 Time complexity

Let define first n_1,n_2 and nnz as the number of rows, column and nonzero inside a matrix. Let also define m as the maximum number of nonzero element on one column in the matrix.
The operator*() between a SparseMatrix and a Vector consist of a loop on which we iterate at most $M * n_2$ times.

We can also observe that if a matrix is without any nonzero element, $M * n_2 = n_1 * n_2 = nnz$. So the time complexity is at most $\mathcal{O}(n_1 * n_2)$.
However, we can find tighter time complexities depending on the structure of our matrix. Indeed, the counter `i` in the for loop is updated in order to skip the rest of a column in the `rowidx` matrix (given in the statement of the project) if it encounters the value -1. For example, if in the k-th row, l-th column, the value is equal to `-1`, `i` will be updated to look at the (k+1)-th row and 1st column of the same matrix `rowidx` for the next iteration inside the loop.
Let's look at 3 extreme scenarios to understand the complexity of the operator:

- **When the matrix `rowidx` only contains non "-1" elements on the first row**
  in this case, the operator perform $\mathcal{O}(m)$ for the first row, and $\mathcal{O}(1)$ for the $n_2 - 1$ other rows. The complexity becomes: $\mathcal{O}(m) + \mathcal{O}(n_2)$. And if $m \leq n_2$, it is performed by $\mathcal{O}(n_2)$
- **When the matrix `rowidx` only contains non "-1" element on the first column:**
  in this case, the operator perform 2 operations for each row and thus, the complexity become $\mathcal{O}(2m) \approx \mathcal{O}(m) \leq \mathcal{O}(n_1)$
- **When the matrix `rowidx` contains $nnz < n_1 * n_2$:**
  in this case, the complexity becomes $\mathcal{O}(nnz)$

2.2 Space complexity

The function only allocate memory when there is a call to the `Vector(int size)` constructor and this constructor, in this particular instance, allocate a space of size $m * n_2$ and we can deduce the complexity:

$$\mathcal{O}(m * n_2)$$

3 Description of memory state during the operator=()

This assignment operator first, if the `otherVector` has a different number of nonzero elements, we free memory previously taken by `rowidx` and `nzval` the delete operator. Thus, those pointers currently point toward nothing.
We modify the value of different scalar variables instance such as `mSize` which is a protected variable inherited from `AbstractVector` class, and `nnz` which is a member.
Afterward, we can compute the size needed for both array and allocate memory, with the help of the `new` operator to which pointers `rowidx` and `nzval` will point toward.
Finally, the values will be updated inside the for loop and the address of the object (`*this`) is returned, as asked inside the function signature.