Time complexity analysis of + between two `SparseVector`

Consider two `SparseVectors`. The first has nnz_1 non-zero values, and the second has nnz_2 non-zero values. The two vectors have respectively attributes `rowidx1`, `nzval1` and `rowidx2`, `nzval2`. We recall the pseudo-code of the algorithm :

```
1:  newRowIdx ← new empty array of size nnz_1 + nnz_2
2:  newNzVal ← new empty array of size nnz_1 + nnz_2
3:  newNnz ← 0
4:  Two pointers i ← 0, j ← 0
5:  while i ≤ nnz_1 and j ≤ nnz_2 do
6:      if rowidx1[i] < rowidx2[j] then
7:          newRowIdx[newNnz] ← rowidx1[i]
8:          newNzVal[newNnz] ← nzval1[i]
9:          i ← i + 1
10:         newNnz ← newNnz +1
11:     else if rowidx1[i] > rowidx2[j] then
12:         newRowIdx[newNnz] ← rowidx2[j]
13:         newNzVal[newNnz] ← nzval2[j]
14:         j ← j + 1
15:         newNnz ← newNnz +1
16:     else
17:         newRowIdx[newNnz] ← rowidx1[i]
18:         newNzVal[newNnz] ← nzval1[i] + nzval2[j]
19:         i ← i + 1
20:         j ← j + 1
21:         newNnz ← newNnz +1
22:     end if
23: end while
24: while i < nnz_1 do
25:     newRowIdx[newNnz] ← rowidx1[i]
26:     newNzVal[newNnz] ← nzval1[i]
27:     i ← i + 1
28:     newNnz ← newNnz +1
29: end while
30: while j < nnz_2 do
31:     newRowIdx[newNnz] ← rowidx2[j]
32:     newNzVal[newNnz] ← nzval2[j]
33:     j ← j + 1
34:     newNnz ← newNnz +1
35: end while
36: finalRowIdx ← new empty array of size newNnz
37: finalNzVal ← new empty array of size newNnz
38: Copy newNnz first elements of newRowIdx to finalRowIdx
39: Copy newNnz first elements of newNzVal to finalNzVal
40: Free newRowIdx and newNzVal
41: Initialize new SparseVector resVector with newNnz, finalRowIdx, finalNzVal
42: Free finalRowIdx and finalNzVal
        return resVector
```

From line 4 to line 23, the loop consists in two pointers i and j going from 0 to maximum either nnz_1 or nnz_2. In any case, thanks to the assumption made in the statement, one execution inside the loop is done in $\mathcal{O}(1)$. The worst case is when the two pointers go respectively to nnz_1 and nnz_2. So the total execution asymptotic time of this loop is $\mathcal{O}(nnz_1 + nnz_2)$. From line 24 to 29, the worst case is when the pointer i is still at 0. Also thanks to the assumption made in the statement, the total execution time of this loop is $\mathcal{O}(nnz_1)$. The exact same analysis can be made for the loop going from line 30 to 35, it has time complexity $\mathcal{O}(nnz_2)$. Finally, lines 38 and 39 are made in $\mathcal{O}(\texttt{newNnz})$ since copying m elements from an array to another is made in $\mathcal{O}(m)$. The same observation can be made for the intialization of the new `SparseVector` object in line 41. Thus the total execution time is made in $\mathcal{O}(nnz_1 + nnz_2) + \mathcal{O}(nn_1) + \mathcal{O}(nnz_2) + \mathcal{O}(\texttt{newNnz})$. Since $nnz_1 \leq nnz_1 + nnz_2$, $nnz_2 \leq nnz_1 + nnz_2$ and $\texttt{newNnz} \leq nnz_1 + nnz_2$, the total execution time is $\mathcal{O}(nnz_1 + nnz_2)$.

Space complexity analysis of `+` between two `SparseVector`

Memory is allocated in lines 1, 2, 36, 37 and 41 of the algorithm wer. In lines 1 and 2, $\mathcal{O}(nnz_1 + nnz_2)$ space is allocated. In lines 36, 37 and 41, $\mathcal{O}(\texttt{newNnz})$ space is allocated. Thus the total space allocated is $\mathcal{O}(nnz_1 + nnz_2) + \mathcal{O}(\texttt{newNnz})$. Since $\texttt{newNnz} \leq nnz_1 + nnz_2$, the total space allocated is $\mathcal{O}(nnz_1 + nnz_2)$. Note that in this analysis we do not consider when $O(1)$ space is allocated such as in line 4.

Time complexity analysis of `*` between a `SparseMatrix` and a `Vector`

Consider the `*` operator between a `SparseMatrix` with attributes m, n, M, `rowidx`, `nzval`, and a `Vector` of size n.

1: Initialize new zero `Vector` `res` of size m
2: **for** $i \in \{1, \dots, n\}$ **do**
3:     **for** $j \in \{1, \dots, M\}$ **do**
4:         $k \leftarrow Mi + j$
5:         **if** `rowidx`$[k] = -1$ **then**
6:             Break
7:         **end if**
8:         `res[rowidx[`k`]]` $\leftarrow$ `res[rowidx[`k`]]` $+$ `nzval[`k`]` $\times$ (i-th element of `v`)
9:     **end for**
10: **end for**

Note that in this analysis we consider that `rowidx` is a $n \times M$ matrix. The initialization in line 1 is made in $\mathcal{O}(m)$ time complexity since copying an array of size s in another is made in $\mathcal{O}(s)$. Then, from line 2 to 10, the two loops consist in browsing `rowidx` line by line, stopping each time that a -1 is reached. Thanks to the assumption made in the statement, each execution after having checked the value of `rowidx` is made in $\mathcal{O}(1)$. Suppose that $nnz \geq n$, then the time complexity from line 2 is $\mathcal{O}(nnz)$ since all non-zero values of a line is placed before the first -1 in `rowidx`. Now, suppose that $n > nnz$, then the algorithm still checks the values of `rowidx` at each line (which is also made in $\mathcal{O}(1)$). In this case, the time complexity is thus $\mathcal{O}(n)$. Thus the total time complexity is $\mathcal{O}(m) + \mathcal{O}(\max(n, nnz))$. If $m \geq \max(n, nnz)$, then the time complexity is $\mathcal{O}(m)$, otherwise it is $\mathcal{O}(\max(n, nnz))$. We can summarize this by saying that the time complexity is $\mathcal{O}(\max(m, n, nnz))$.

Space complexity analysis of `*` between a `SparseMatrix` and a `Vector`

The only space allocation is made in line 1. It allocates $\mathcal{O}(m)$ space to store the solution. Note that $\mathcal{O}(m) = \mathcal{O}(\max(m, n, nnz))$. Indeed if $m = \max(m, n, nnz)$, then it is direct that $\mathcal{O}(m) = \mathcal{O}(\max(m, n, nnz))$, otherwise $\mathcal{O}(m) = \mathcal{O}(\max(m, n, nnz)))$ since $m < \max(n, nnz)$.

Description of the memory states during the call of `=` of `SparseVector`

First, note that the variables `mSize` and `nnz` of `this` always point to the same place in memory, but their value might change. Concerning the variables `rowidx` and `nzval`, there are two cases. Either the value of `nnz` of `this` is the same as the value of `nnz` of `otherVector`, or it is not the case. In the first case, the places in memory to which `rowidx` and `nzval` of `this` point do not change, although new information from `otherVector` is copied in these places. In the second case (`nnz` are not the same), the places to which `rowidx` and `nzval` point are freed. New places in memory are then allocated, and `rowidx` and `nzval` now point to these new places. Again, information from `otherVector` is copied in these new places. In the case of `rowidx`, an illustration is given in Figure 1.

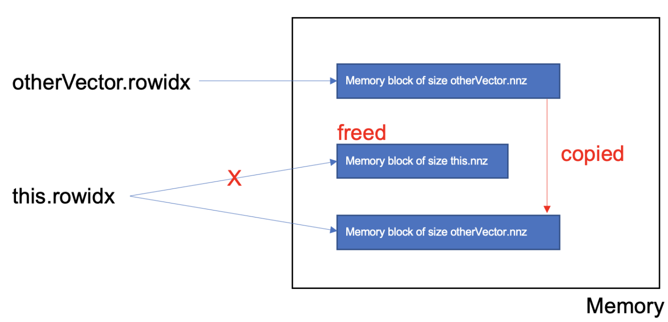

Figure 1: `=` operator memory modifications if `this.nnz` and `otherVector.nnz` have not identical values