GRAPHBLAS

# Maximum Matching in LAGraph

## A guide to the algorithm and its implementation

Christina Koutsou
ECE, Aristotle University of Thessaloniki
github: @kchristin22

# 1 About

Many graph algorithms are usually described periphrastically, instead of realizing the underlying linear algebra operations performed. LAGraph [5] is an open-source selection of graph algorithms implemented using GraphBLAS [3], a parallel and efficient framework for sparse matrix operations on an extended algebra of semirings. The semiring operations can be chosen from an extensive list (f.i. addition, min, max, multiplication, first value etc.) and the usage of descriptors, such as considering only the structure of a matrix and not its values, further increases flexibility and performance.

One group of algorithms that LAGraph currently misses is the one that concerns bipartite graphs. These types of graphs are very useful for solving problems between two types or groups of objects, clearly showing their relationship with each other. Hence, bipartite graphs excel at solving matching problems, such as assigning tasks to workers, and these types of problems are frequently found in economics, biology, transportation, and many more fields. One of the most popular challenges is the maximum matching problem which refers to finding the maximum number of independent pairs between sets. This project aims to implement the paper "Distributed-Memory Algorithms for Maximum Cardinality Matching in Bipartite Graphs" by Dr. Ariful Azad and Dr. Aydın Buluç [1] using GraphBLAS, as this algorithm favors parallelism and already makes use of linear algebra abstractions.

# 2 High-level explanation of the algorithm

Most maximum matching algorithms rely on Depth-First Search (DFS), meaning following one path at a time. This method is sequential and, thus, can be very slow for large matrices. Breadth-First Search, however, is highly parallelizable, as it provides the capability of exploring many paths simultaneously. But the problem that arises from this is how can we ensure that there are no items with the same match, or, in other words, how can we ensure that the paths are disjoint? Let's have a closer look at the algorithm and how that is achieved.

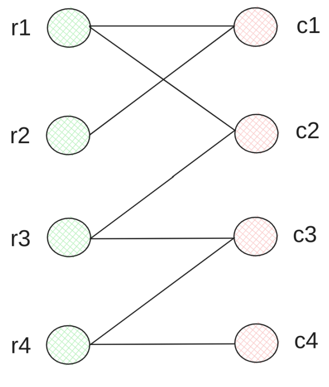

Figure 2.1: *Bipartite graph, initial matching*

Consider this example of a bipartite graph. The graph consists of two sets, the rows and the columns, as denoted in the adjacency matrix. The matches are recorded from the perspective of the column. At each iteration, till there are no paths left, we begin from the unmatched columns and perform a single level of BFS. For each row, the parent with the minimum id is chosen, and every child that stems from the BFS is marked as visited, in order not to be considered in other paths later on. This way, the paths remain independent. In the example, all rows are unmatched in the beginning, so we end up with the matches shown in bold in Figure 2.2.

Since there are still paths to be explored, the algorithm continues with the next iteration. This time, the only unmatched column is c4, so the result of the single BFS step includes only r4. However, r4 is already matched, hence we dive deeper in this path and get redirected to the mate of r4, c3. The only unvisited node that originates from its BFS is r3, which is again matched. We continue traversing the path in the same fashion until we find an unvisited and unmatched row- in our case, r2. This path is now marked as finished and since there are no other paths at the moment, the algorithm moves to updating the mates of the columns and rows respectively.

The unmatched row of each path is matched to its parent, which is the first of its parents encountered depth-wise, before marked as visited, and should be the last column visited in this path. Afterwards, the rows previously matched with these columns are matched with their parents, etc. As a result, in the presented example, we end up with the match shown in Figure 2.3 instead.

Essentially, the algorithm explores alternate paths by reversing an already traversed path to see if one of the already matched columns encountered in the path has at least one free children to be matched with instead. If so, it concedes its previous match to another previously matched parent or to the unmatched root of the path.

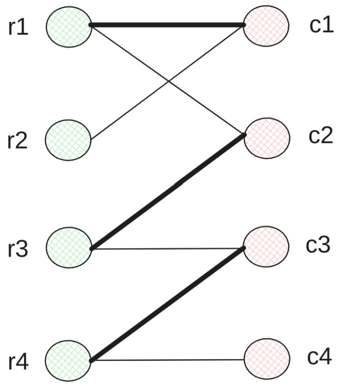

Figure 2.2: *Matching after first iteration*

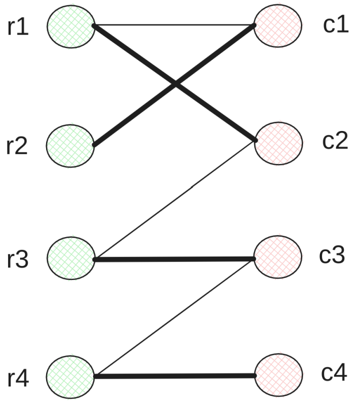

Figure 2.3: *Alternate and final matching*

# 3 Implementation

## 3.1 Helper functions

GraphBLAS is an API designed to handle the mathematical characteristics of a graph and seeks to convert graph operations into linear algebra expressions. Even though the pseudocode already incorporates this type of operations, the helper functions defined in a mathematical way can be complicated to implement.

To begin with, there are two different kinds of SET(y,x) operations used in this algorithm. The first kind is used to update and augment the parents of the visited rows, the path and the mate vectors, while the latter is used to set the parents of the R frontier to their mates and inside the augment function, when setting $u_r$ and $u_c'$. In GraphBLAS, these two operations are both equivalent to assignments but with different masks. For the update and augment operation, the mask is the same as the input's structure in order to only update entries present in both input and output and insert entries in the output, while the rest of the output's positions stay unaffected. In the second case of SET, the mask matches the output's structure as the purpose is to only update its current entries, without adding new ones.

The SELECT operation is more straightforward, relying on an assignment with the equivalent mask and descriptor.

The semiring is also similar with that described in the paper, using two User-Defined Operations, as the VERTEX type introduced in the paper is not a standard C variable type and GraphBLAS cannot handle it directly.

INVERT is performed by extracting the tuples of a vector and building a new one with indices and values swapped, using `GrB_Vector_extractTuples` and `GrB_Vector_build` respectively. In some cases, `GrB_Vector_build` is needed to handle duplicates in values that would become duplicate indices in the inverted version. Similarly to the paper, the binary operation used for the duplicates is FIRST, which is an extension to the specification [4] as it violates the requirement of this operation being associative. The output vector must be cleared before using `GrB_Vector_build`. SuiteSparse's version of GraphBLAS [6] offers an extension to the API called `GxB_Vector_unpack_CSC`, which provides a faster way to obtain vector data in CSC format, leaving the vector empty afterwards. To mimic this behavior when using `GrB_Vector_extractTuples`, the vector is cleared after the operation is performed. The reverse operation, `GxB_Vector_pack_CSC`, does not allow any duplicate indices to be present, so, in those cases, `GrB_Vector_build` is used instead. `GxB_Vector_pack_CSC` takes ownership of the provided data, while `GrB_Vector_build` copies them. Thus, when the tuple lists need to be retained, `GrB_Vector_build` is preferred. `GxB_Vector_pack_CSC` has no requirement of emptying the vector priorly and discards previously stored data.

The PRUNE operation would typically involve three INVERT and one NAND operation. The two vectors to be compared are inverted to find common values, which are then discarded, and the output is inverted again to obtain the final result. However, the implementation modifies this procedure slightly to reduce the required work (See 3.2).

| Name | Operation | GraphBLAS equivalent |
|---|---|---|
| SET | Update and Augment: for each i∈x: y[i] = x[i] | GrB_Vector_assign(y, x, NULL, x, GrB_ALL, n, GrB_DESC_S)) |
| | Assign: for each i∈y: y[i] = x[i] | GrB_Vector_assign(y, y, NULL, x, GrB_ALL, n, GrB_DESC_S) |
| SELECT | for each i∈x: if (y[i] not empty) z[i] = x[i] | GrB_Vector_assign(z, y, NULL, x, GrB_ALL, n, GrB_DESC_RS) |
| | for each i∈x: if (y[i] empty) z[i] = x[i] | GrB_Vector_assign(z, y, NULL, x, GrB_ALL, n, GrB_DESC_RSC) |
| INVERT | [I,X] → [X,I] (No duplicates and input vector not needed afterwards) | GxB_Vector_unpack_CSC(x, (GrB_Index **)&I, (void **)&X, &Ibytes, &Xbytes, NULL, &nvals, &jumbled, NULL)<br>GxB_Vector_pack_CSC(y, (GrB_Index **)&X, (void **)&I, Xbytes, Ibytes, NULL, nvals, true, NULL) |
| | [I,X] → [X,I] (Possible duplicates existing or input vector needed afterwards) | GxB_Vector_unpack_CSC(x, (GrB_Index **)&I, (void **)&X, &Ibytes, &Xbytes, NULL, &nvals, &jumbled, NULL)<br>GrB_Vector_clear(y)<br>GrB_Vector_build_UINT64(y, X, I, nvals, GrB_FIRST_UINT64) |

Table 3.1: *Helper functions*

## 3.2 Algorithm walk-through

```
procedure MCM-DIST($\mathbf{A}$, $mate_c$, $mate_r$)
    repeat do
        $\pi_r \leftarrow$ -1 GrB_Vector_clear(parentsR);
        $path_c \leftarrow$ -1
        $f_c \leftarrow$ an empty sparse vector of size $n$ of type VERTEX
        for $i \in$ IND($mate_c$) do                 } GrB_Vector_apply_IndexOp_UDT(frontierC,
            if $mate_c[i] =$ -1 then                 } mateC, NULL, initFrontierOp, I, &y, GrB_DESC_RSC);
                $f_c[i] \leftarrow$ VERTEX($i, i$)      }
        while $f_c \neq \phi$ do
            ▷ Step 1: Explore neighbors of column frontier (one step of BFS)
            $f_r \leftarrow$ SPMV($\mathbf{A}, f_c$, SR=(select2nd, minParent))  } GrB_mxv(frontierR, parentsR, NULL,
            ▷ Step 2, 3, 4: Select unvisited, matched, and } unmatched row vertices MinParent_2nd_Semiring
            $f_r \leftarrow$ SELECT($f_r, \pi_r =$ -1)      }                          A, frontierC, GrB_DESC_RSC);
            $\pi_r \leftarrow$ SET($\pi_r$, PARENT($f_r$)) GrB_Vector_apply(parentsR, frontierR, NULL, getParentsOp, frontierR,GrB_DESC_S));
            $uf_r \leftarrow$ SELECT($f_r, mate_r =$ -1) GrB_Vector_assign(ufrontierR, mateR, NULL, frontierR, GrB_ALL, nrows, GrB_DESC_RSC));
            $f_r \leftarrow$ SELECT($f_r, mate_r \neq$ -1) GrB_Vector_assign(frontierR, mateR, NULL, frontierR, GrB_ALL, nrows, GrB_DESC_RS));
            if $uf_r \neq \phi$ then
                ▷ Step 5: Store endpoints of newly discovered augmenting paths
                $t_c \leftarrow$ INVERT(ROOT($uf_r$)) GrB_Vector_apply to get roots, GxB_Vector_unpack on roots of ufR, GrB_Vector_build tc
                $path_c \leftarrow$ SET($path_c, t_c$) GrB_Vector_assign(pathC, pathUpdate, NULL, pathUpdate, GrB_ALL, ncols, GrB_DESC_S);
                ▷ Step 6: Prune vertices in trees yielding augmenting paths
                $f_r \leftarrow$ PRUNE($f_r$, ROOT($uf_r$))  } if ufR != empty: get roots of fR and their mates and create a vector v(i) = j,
            ▷ Step 7: Construct next frontier         } where j is the mate of the first row of fR that has i as root; mask with tc to remove
            SET(PARENT($f_r$), $mate_r$)               } entries stemming from the same paths; invert v to move entries to their final
            $f_c \leftarrow$ INVERT($f_r$)                  } positions on fC; build tuple of fC from v with indices as parents and values as roots
                                                      } if ufR == empty:  update parents of fR; invert fR using mates as indices; assign to fC
        ▷ Step 8: Augment matching by all augmenting paths discovered in this phase
        AUGMENT($path_c$, $\pi_r$, $mate_r$, $mate_c$)
    until an augmenting path is discovered in the current phase while (npaths);
```

```
procedure AUGMENT( $path_c$, $\pi_r$, $mate_r$, $mate_c$)
    ~~$v_c \leftarrow$ sparse vector from $path_c$~~ by removing entries with -1 → pathC is already sparse
    while $v_c \neq \phi$ do while (npaths);
        $v_r \leftarrow$ INVERT($v_c$) GxB_Vector_unpack_CSC on pathC; GxB_Vector_pack_CSC vr
        $v_r \leftarrow$ SET($v_r, \pi_r$) GrB_Vector_assign(vr, vr, NULL, parentsR, GrB_ALL, nrows, GrB_DESC_S);
        $v_c \leftarrow$ INVERT($v_r$) GxB_Vector_unpack_CSC on vr; GxB_Vector_pack_CSC pathC
        $v'_c \leftarrow$ SET($v_c, mate_c$) GrB_Vector_assign(pathCopy, pathC, NULL, mateC, GrB_ALL,
                                              ncols, GrB_DESC_RS));
        $mate_c \leftarrow$ SET($mate_c, v_c$) GrB_Vector_assign(mateC, pathC, NULL, pathC, GrB_ALL, ncols,
                                              GrB_DESC_S));
        $mate_r \leftarrow$ SET($mate_r, v_r$) GrB_Vector_assign(mateR, vr, NULL, vr, GrB_ALL, nrows,
                                              GrB_DESC_S));
        $v_c \leftarrow v'_c$ GrB_Vector temp = pathC;
                 pathC = pathCopy;
                 pathCopy = temp;
```

One of the main goals of this implementation is to provide the user with a lot of flexibility when it comes to the input and output arguments. In more detail, even though the algorithm does not require an initial matching, such an option is given along with a flag denoting the set from whose perspective the match stands, columns or rows. The vectors mate$_c$ and mate$_r$ are initialized accordingly by assigning the initial matching to one and inverting the latter to form the other, leaving the input vector unchanged. As previously explained, the INVERT function relies on unpacking the vector's contents and building a new vector by swapping the indices and values. Even though there should be no duplicate values in the built mate vector to be inverted and, therefore, `GxB_Vector_pack_CSC` should be preferred, this vector cannot remain empty afterwards, so `GrB_Vector_build` is used instead to create the other mate vector. On the same note, as both mate$_r$ and mate$_c$ are computed during each iteration, they are both returned to the user, avoiding any limitation or extra work needed from the user's side.

Similarly to the pseudocode, at the beginning of each iteration of the outer loop, a column frontier from the unmatched columns is constructed. This is accomplished by applying an index operator on a full vector to create the tuples and discarding those corresponding to matched columns. If this set is not empty, a single step of Breadth-First-search is performed using the same semiring mentioned in the paper. GraphBLAS allows for a masked application of the semiring so only unvisited children remain in the R frontier. To distinguish already matched children from unmatched ones for the SELECT operations that follow, the mate$_r$ vector is applied as a structural mask and the old values of the outputs are wiped out or updated, as a result of the 'replace' descriptor. For efficiency, a copy of a trimmed mate$_r$ vector is stored, containing only the matches of the nodes in the current R frontier.

If any unmatched rows are found in these paths, these matches need to be stored and the corresponding paths terminated. Firstly, the roots of the paths are extracted from the unmatched R frontier and later inverted to update the path$_c$ vector.

The PRUNE operation would, at this point, include two INVERT and two NAND operations. In more detail, after extracting the roots of the f$_r$ vector through a call to `GrB_Vector_apply`, this vector would be inverted, masked with the vector containing the latest update of path$_c$ to exclude the latest visited rows stemming from the same column, and then inverted again to mask the R frontier and obtain its pruned version. Subsequently, the parents of the R frontier would have to be set to their column mates, and these parents would be used to invert the R frontier and form the new C frontier. However, the current implementation takes advantage of the fact that PRUNE contains an INVERT operation on the R frontier and the C frontier's creation depends on inverting the R frontier, and, hence, these operations are merged. For pruning, we care about the values of the roots and not the row indices on which they reside, and since we would later have to set these row indices to their column mates, we can combine these steps when inverting the root$_{f_r}$ vector: Both the root$_{f_r}$ vector and the vector storing the column mates of the current R frontier are unpacked in order. These two vectors have entries in the same positions as they stem from the R frontier. Thus, we can build the following vector:

$$rootfRIndexes(j) = i, \quad \text{where (i,j) = (parentC, rootC) of the new C frontier}$$

It is evident now that, after masking with the latest update of path$_c$, which in turn stems from inverting the roots of uf$_r$, we get a pruned R frontier that follows the aforementioned formula. In other words, this vector contains all the information needed to create the new C frontier. Firstly, the vector is inverted so the positions match the mates of the parents of the pruned R frontier. Then, an index unary operation is applied where the tuples of the C frontier are formed as follows:

```c
void *buildfCTuples(vertex *z, uint64_t *x, uint64_t i, uint64_t j, const void *y)
{
    z->parentC = i;
    z->rootC = *x;
}
```

If uf$_r$ is empty, the procedure is simplified by first applying an operator on the R frontier to set the parents to their mates and then unpacking the vector to get its values. The mates of the parents are acquired by the values in the vector of the current row mates- which has the same structure as the R frontier- and they are also used as the indices for the C frontier when the latter is packed.

After all rows are visited, the AUGMENT algorithm is executed. The INVERT functions happening here rely exclusively on `GxB_Vector_unpack_CSC` and `GxB_Vector_pack_CSC` as no duplicates should be present. To set the values of the $u_r$ vector equal to the parents of these visited rows, $u_r$ is also used as a structural mask to the `GrB_Vector_assign` operation in order to ensure that no extra entries will be imported by the $\pi_r$ vector and that only the ones already present will be updated. As for the $u_c^{'}$ vector, it is set by utilizing the path vector as a structural mask along with the 'replace' descriptor to delete any entries present in the previous iteration that are not currently included in the mask. After updating the mates of the columns, the two path vectors are swapped to prepare for the next iteration. The augment loop exits when all paths have been examined and there are no entries left.

The program terminates when there are no augmented paths left, and the helper outputs used internally, $mate_c$ and $mate_r$, are finally assigned to the program's outputs respectively.

## 3.3 Push-Pull optimization

Looking at the algorithm's performance, the heaviest operation appears to be the semiring. It is helpful to think of the semiring as a matrix multiplication and examine the different ways that this can be thought of and, thus, executed.

GraphBLAS treats matrix multiplication as performed from the row perspective. As a result, for each row, the dot product is performed with the vector. The sparsity of the vector does not matter in this case, as all non-zero elements of A must be read. However, matrix multiplication can be executed column-wise, meaning only the columns of A indicated by the vector's non-zero values are touched, leading to less memory accesses and better performance when the vector is sparse [2].

From linear algebra, we know that the following rule is valid:

$$C = A \cdot v = (v^T \cdot A^T)^T$$

Consequently, we can perform matrix-vector multiplication from the column perspective by transposing the vector and the matrix. The output is a vector so we let GraphBLAS decide which is the most efficient way to store it, as a row or as a column. The parents are then applied as a mask separately to ensure that the previous output vector and the mask are aligned correctly.

The name of this optimization stems from this trick being used on the implementation of the PageRank algorithm [7]. The pull direction corresponds to the row-based approach, where each vertex "pulls" the contributions of its incoming neighbors to compute its score. On the opposite side, the push direction corresponds to the column-based approach, where each vertex "pushes" its contribution to its outgoing neighbors.

The current criterion for deciding which method to follow at each iteration is the format of the vector. If the vector is sparse or hypersparse, the push approach is preferred, while if the vector is bitmap or full, the push approach is followed instead.

## 3.4 Vanilla version

The vanilla version of the algorithm excludes any extensions to the GraphBLAS API, which mainly refer to creating operators, using pack/unpack methods and the FIRST operator.

| **GraphBLAS extension** | **Replacement** |
|---|---|
| GxB_*_new | GrB_*_new |
| GxB_Vector_unpack_CSC | GrB_Vector_extractTuples_UINT64 |
| GxB_Vector_pack_CSC | GrB_Vector_build_UINT64 |
| GrB_FIRST_UINT64 | GrB_MIN_UINT64 |

Table 3.2: *Replacement of extensions for the vanilla version*

The vanilla approach to creating the C frontier for the next iteration, by appropriately combining two vectors, differs significantly from the SuiteSparse implementation.

In the SuiteSparse version, the values and indices of both vectors are extracted using the unpack method. The second vector's values are then used as indices for the output vector, while the values of the first vector serve as the corresponding values. To construct the output vector, the pack method is used if there are no duplicate entries in the second vector; otherwise, the build method is chosen.

In contrast, the vanilla version first checks whether the second vector contains duplicate entries. If duplicates are found, the vector is inverted to eliminate them. The remaining indices of the second vector are then used to extract the corresponding elements from the first vector. Finally, the `GrB_Vector_build` function is used to construct the output vector as previously described.

# References

[1] A. Azad and A. Buluç. "Distributed-Memory Algorithms for Maximum Cardinality Matching in Bipartite Graphs". *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

[2] C. Yang, A. Buluç and J. D. Owens. "Implementing Push-Pull Efficiently in GraphBLAS". *ICPP '18: Proceedings of the 47th International Conference on Parallel Processing*, 2018.

[3] Timothy Davis. GraphBLAS. https://github.com/DrTimothyAldenDavis/GraphBLAS.

[4] Timothy Davis. GraphBLAS specification. https://github.com/DrTimothyAldenDavis/GraphBLAS/blob/stable/Doc/GraphBLAS_UserGuide.pdf.

[5] Timothy Davis. LAGraph. https://github.com/GraphBLAS/LAGraph.

[6] Timothy Davis. SuiteSparse. https://github.com/DrTimothyAldenDavis/SuiteSparse.

[7] S. Beamer, K. Asanović and D. Patterson. "Reducing Pagerank Communication via Propagation Blocking". *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.