Maximum Matching in Bipartite Graphs using LAGraph

TEXAS A&M UNIVERSITY Engineering

Christina Koutsou, Dr. Timothy Davis, Department of Computer Science & Engineering, Texas A&M University

ARISTOTLE UNIVERSITY OF THESSALONIKI

Background and Motivation

Many graph algorithms are usually described periphrastically, instead of realizing the underlying linear algebra operations performed. **LAGraph** is an open-source selection of graph algorithms implemented using **GraphBLAS**, a parallel and efficient framework for sparse matrices 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 to solve 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 issues 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 using GraphBLAS, as the algorithm favors parallelism and already makes use of linear algebra abstractions.

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.

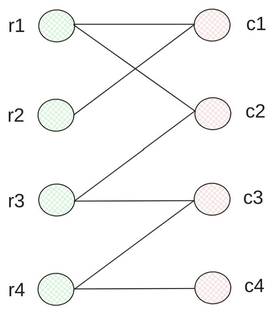

Figure 1: Bipartite graph

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 column's perspective. **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 stay independent.** In the example, all rows are unmatched in the beginning, so we end up with the matches shown in bold in Figure 2.

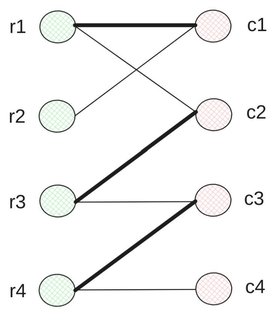

Figure 2: First iteration matches

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 stemming 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 parent column encountered depth-wise. 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 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 child to be matched with instead.

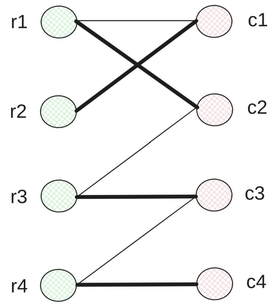

Figure 3: Alternate matchings, Maximum Matching of the Bipartite graph

Implementation

```
1:  procedure MCM-DIST(A, mate_c, mate_r)
2:      repeat do
3:          π_r ← -1  GrB_vector_clear(parentsR);
4:          path_c ← -1
5:          f_c ← an empty sparse vector of size n of type VERTEX
6:          for i ∈ IND(mate_c) do
7:              if mate_c[i] = -1 then
8:                  f_c[i] ← VERTEX(i, i)        } GrB_Vector_apply_IndexOp_UDT(frontierC,
                                                    mateC, NULL, initFrontierOp, I, &y, GrB_DESC_RSC);
9:          while f_c ≠ φ do
10:             ▷ Step 1: Explore neighbors of column frontier (one step of BFS)
11:             f_r ← SPMV(A, f_c, SR=(select2nd, minParent))   } GrB_mxv(frontierR, parentsR, NULL,
12:             ▷ Step 2, 3, 4: Select unvisited, matched, and unmatched row vertices MinParent_2nd_Semiring
13:             f_r ← SELECT(f_r, π_r = -1)                                          A, frontierC, GrB_DESC_RSC);
14:             π_r ← SET(π_r, PARENT(f_r))  GrB_Vector_apply(parentsR, frontierR, NULL, getParentsOp, frontierR,GrB_DESC_S));
15:             uf_r ← SELECT(f_r, mate_r = -1)  GrB_Vector_assign(ufrontierR, mateR, NULL, frontierR, GrB_ALL, nrows, GrB_DESC_RSC));
16:             f_r ← SELECT(f_r, mate_r ≠ -1)  GrB_Vector_assign(frontierR, mateR, NULL, frontierR, GrB_ALL, nrows, GrB_DESC_RS));
17:             if uf_r ≠ φ then
18:                 ▷ Step 5: Store endpoints of newly discovered augmenting paths
19:                 t_c ← INVERT(ROOT(uf_r))  GrB_Vector_apply to get roots, GxB_Vector_unpack on roots of ufR, GrB_Vector_build tc
20:                 path_c ← SET(path_c, t_c)  GrB_Vector_assign(pathC, pathUpdate, NULL, pathUpdate, GrB_ALL, ncols, GrB_DESC_S);
21:                 ▷ Step 6: Prune vertices in trees yielding augmenting paths
22:                 f_r ← PRUNE(f_r, ROOT(uf_r))   } if ufR != empty: get roots of fR and their mates and create a vector v(i) = j,
23:             ▷ 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
24:             SET(PARENT(f_r), mate_r)             entries stemming from the same paths; invert v to move entries to their final
25:             f_c ← 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
26:         ▷ Step 8: Augment matching by all augmenting paths discovered in this phase
27:         AUGMENT(path_c, π_r, mate_r, mate_c)
28:     until an augmenting path is discovered in the current phase  while (npaths);
```

```
1:  procedure AUGMENT( path_c, π_r, mate_r, mate_c)
2:      v_c ← sparse vector from path_c by removing entries with -1
3:      while v_c ≠ φ do  while (npaths);          pathC is already sparse
4:          v_r ← INVERT(v_c)  GxB_Vector_unpack_CSC on pathC; GxB_Vector_pack_CSC vr
5:          v_r ← SET(v_r, π_r)  GrB_Vector_assign(vr, vr, NULL, parentsR, GrB_ALL, nrows, GrB_DESC_S);
6:          v_c ← INVERT(v_r)  GxB_Vector_unpack_CSC on vr; GxB_Vector_pack_CSC pathC
7:          v'_c ← SET(v_c, mate_c)  GrB_Vector_assign(pathCopy, pathC, NULL, mateC, GrB_ALL,
                                                       ncols, GrB_DESC_RS));
8:          mate_c ← SET(mate_c, v_c)  GrB_Vector_assign(mateC, pathC, NULL, pathC, GrB_ALL, ncols,
                                                         GrB_DESC_S));
9:          mate_r ← SET(mate_r, v_r)  GrB_Vector_assign(mateR, vr, NULL, vr, GrB_ALL, nrows,
                                                         GrB_DESC_S));
10:         v_c ← v'_c  GrB_Vector temp = pathC;
                        pathC = pathCopy;
                        pathCopy = temp;
```

Benchmarks

Speedup: (1) x4 times faster, (2) x417 times faster, (3) x199 times faster

References

- *Distributed-Memory Algorithms for Maximum Cardinality Matching in Bipartite Graphs* by Ariful Azad and Aydın Buluç
- GraphBLAS repository and User Guide: https://github.com/DrTimothyAldenDavis/GraphBLAS
- LAGraph repository: https://github.com/GraphBLAS/LAGraph