Authors: #JackEdmonds Link

Introduction

Consider a graph $G(V, E)$

- A **matching** in G is a subset of its edges such that no two are adjacent.
- A **maximum matching** is a matching with maximum cardinality.

This paper describes an efficient algorithm to find a maximum matching.

Definitions and Notations

- $A \oplus B = (A - B) \cup (B - A)$ (XOR operation on sets)
- Alternating Path: For matchings M and M', an alternating path is a path that has alternate edges from M and M'.
- Augmenting Path: For a matching M on a graph G, an M-augmenting path is a path (e1, e2, . . . , ek) of odd length from v1 to v2 such that v1 and v2 are not covered by M, $e_1, e_k \notin$ M, and the edges alternate membership in M.

M is maximum $\iff$ There is no augmenting path

- M is maximum $\rightarrow$ There is no augmenting path
  - Suppose P is an augmenting path
  - Invert P (swap matched and unmatched edges) to form M'.
  - $|M'| > |M|$, hence contradiction
- There is no augmenting path $\rightarrow$ M is maximum
  - M is not maximum $\rightarrow$ There is an augmenting (Converse)
    - Suppose M' is a maximum matching
    - Consider the graph $G'(V, M \oplus M')$
      - Its components are alternating paths as each vertex can lie only on edges from M or M' (which are each at most one), hence maximum degree = 2
    - There exists an alternating path P in G' such that there are more edges from M'
      - Suppose there is no such P
      - Then for every component C, $C \cap M \geq C \cap M'$
      - Hence $|M| \geq |M'|$, contradiction
    - P is an Augmenting path for M

Rough Outline of the Algorithm

```
M = {};
while ( There is an Augmenting Path P ) {
        // M is not maximum
        M = M ^ P;    // xor
```

```
    }
    return M;
```

Correctness

- while loop
  - initialisation: M is not maximum
  - maintenance: M is not maximum
  - termination: M is maximum
- follows from the theorem above

Challenge: How to efficiently find augmenting paths?

Definitions and Notations

At any step in the algorithm, let M be the current matching and let X be the set of exposed vertices (vertices not covered by M).

- M-Alternating trees: An M-alternating tree is a tree in G with a root vertex r ∈ X such that along every path $P = e_1 e_2 \ldots e_j$ from r to a leaf v the edges alternate between being in M and not being in M (ei ∈ M ⇔ i is even).
- For an M-alternating tree T with root r ∈ X, Odd is the set of vertices in T at an odd distance from r and Even is the set of vertices in T at an even distance from r (including r).
- A maximal M-alternating tree is an M-alternating tree such that no Even vertex in the tree has an edge to a vertex not in the tree (i.e. no additional vertices can be added to the M-alternating tree).

If an M-alternating tree contains a vertex $v \in X$ (set of exposed vertices) distinct from the root, then there exists an M-augmenting path.

- There is a unique path P from r to v
- The edges alternate as per definition
- The edge incident on v is not in M
- Hence, P is an augmenting path

Blossom Algorithm

```
for( r in V ){
        if( match[r] != NULL) continue;
        bfs_queue.push(r);
        parent[n];
        level[n];

        parent[r] = r;
        level[r] = 0;
        while(bfs_queue.empty() == false){
                v = bfs_queue.pop();
                if(v != r) level[v] = level[parent[v]]+1;
                for(all neighbours w of v){
                        if( parent[w] == NULL && match[w]!=null ){
                                parent[w] = v;
                                level[w] = level[parent[w]] + 1;
                                parent[match[w]] = w;
                                level[match[w]] = level[parent[match[w]]] + 1;
```

```
                    }
                    else if (parent[w]!=NULL){
                            if( (level[v] + level[w] +1) %2 == 0 ){     // even cycle
                                    continue;
                            }
                            else {                                            // odd
cycle
                                    contract cycle;
                            }
                    }
                    else if ( w in F ){
                            expand contracted cycles and reconstruct augmenting path;
                            invert augmenting path;
                            break;
                    }
            }
        }
}
```

Time Complexity

- The for loop iterates n times
- The bfs takes O(n+m) time but due to cycle detection and contraction, it takes O(n * m) time
- Hence overall time complexity of $O(m * n^2)$