

# CMPSCI 611: Advanced Algorithms

## Lecture 7: Bipartite Matchings and Union Find

Andrew McGregor

Last Compiled: January 31, 2024

# Outline

Intersection of Matroids and Bipartite Matchings

Union-Find Data Structure

# Bipartite Matchings

## Problem

- ▶ Input: Bipartite graph  $B = (U, V, E)$  where  $U, V$  are disjoint sets of vertices and  $E$  is a set of edges between  $U$  and  $V$ .
- ▶ Output: The matching (i.e., subset of  $E$  where no two edges share a vertex) of maximum size.

**Example Application:** There's a set of tasks  $V$  to be performed and a set of individuals  $U$ , each capable of doing a subset of the tasks.

- ▶ Each person may be assigned to at most one task.
- ▶ At most one person may be assigned a task.
- ▶ Not every person can do every task. Can encode this in  $E$ .

# Intersection of Matroids

- ▶ The bipartite matching subset system is not a matroid but it is the intersection of two matroids.
- ▶ Define:

$\mathcal{I}$  = subsets of  $E$  where each  $u \in U$  has degree at most 1

$\mathcal{I}'$  = subsets of  $E$  where each  $v \in V$  has degree at most 1

Then, bipartite matching subset system is  $(E, \mathcal{I} \cap \mathcal{I}')$

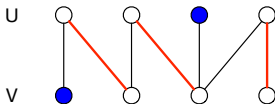
## Theorem

*For matroids  $(E, \mathcal{I})$  and  $(E, \mathcal{I}')$ , the largest set in  $\mathcal{I} \cap \mathcal{I}'$  can be found in time  $O(|E|^3 \cdot C(\mathcal{I}, \mathcal{I}'))$  where  $C(\mathcal{I}, \mathcal{I}')$  is time to check  $i \in \mathcal{I}$  or  $i \in \mathcal{I}'$ .*

We won't prove this general theorem but will focus on the special case of bipartite matching. Note that there is no analogous theorem for the intersection of three matroids.

# Augmenting Paths Definitions

Let  $M$  be a matching in a bipartite graph  $B = (U, V, E)$ . A *free vertex* is a node not incident to any edge in  $M$ . E.g., the blue vertices in

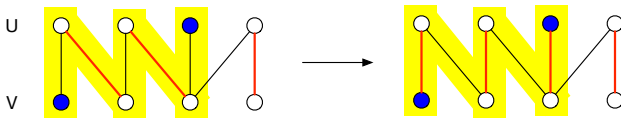


## Definition

An *augmenting path* is an odd sequence of edges that begins and ends at (different) free vertices and alternates between matching edges  $e \in M$  and non-matching edges  $e \in E - M$ .

## Definition

If  $P$  is an augmenting path for matching  $M$ , the *symmetric difference* of  $M$  and  $P$  is  $M \oplus P := (M \cup P) - (M \cap P)$ .



# Augmenting Paths Properties

## Lemma

*For matching  $M$  and augmenting path  $P$ ,  $M \oplus P$  is a matching and*

$$|M \oplus P| = |M| + 1 .$$

## Lemma

*If  $M$  is non-maximum matching, there exists an augmenting path.*

**Algorithm:** Find augmenting paths until we can't find anymore!

# Finding an augmenting path allows us to “grow” matching

## Lemma

For matching  $M$  and augmenting path  $P$ ,  $M \oplus P$  is a matching and

$$|M \oplus P| = |M| + 1 .$$

## Proof.

- ▶ A matching is a graph where no node has degree  $> 1$
- ▶ Size of matching is (number of degree 1 nodes)/2
- ▶ Remove edges in  $P \cap M$  and add edges in  $P \setminus M$ :
  - ▶ Adds one to degree of two nodes that initially were free.
  - ▶ Degree of interior points of  $P$  still have degree 1.



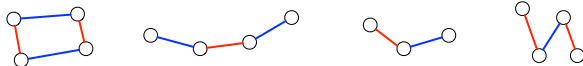
# Augmenting path exists for non-maximum matching

## Lemma

If  $M$  is non-maximum matching, there exists an augmenting path.

## Proof.

- ▶ Let  $M'$  be a matching such that  $|M'| > |M|$
- ▶ Consider  $E' = M \oplus M'$ ... consists of simple paths and cycles whose edges alternate between  $M$  and  $M'$



- ▶ Cycles have the same number of edges from  $M'$  and  $M$
- ▶ There must exist a path  $P$  with more edges from  $M'$  than  $M$ , i.e., one that starts and ends with an edge in  $M'$
- ▶ This is an augmenting path: edges alternate between  $M'$  and  $M$  and it starts and ends with free vertices





# Bipartite Matching Algorithm

## Algorithm

- ▶  $M \leftarrow \emptyset$
- ▶ While there exists an augmenting path  $P$ :  $M \leftarrow M \oplus P$
- ▶ Return  $M$

We can find an augmenting path in  $O(|U||E|)$  time:

- ▶ Direct matched edges  $V \rightarrow U$  and non-matched edges  $U \rightarrow V$
- ▶ For each free vertex  $u \in U$ , grow a BFS: If a free vertex  $v \in V$  is reachable from  $u$ , we have an augmenting path

Total running time is  $O(\min(|U|, |V|)|U||E|)$  because the maximum matching size is at most  $\min(|U|, |V|)$ . Can be improved by finding the augmenting paths in a more clever way.

# Outline

Intersection of Matroids and Bipartite Matchings

Union-Find Data Structure

## Recall Kruskal's Algorithm...

**Problem:** Given an undirected, connected graph  $G = (V, E)$  with positive edge weights, find the minimum-weight subset  $E' \subset E$  such that the graph  $G = (V, E')$  is a minimum spanning tree.

### Algorithm (Kruskal)

1. Sort edges by non-decreasing weight
2.  $F = \emptyset$
3. Until  $F$  is a spanning tree of  $G$ 
  - 3.1 Get the next edge  $e$
  - 3.2 If  $F + e$  is acyclic then  $F = F + e$

We saw how to implement this with  $O(|E| \log |E| + |V|^2)$  running time.  
This class: improving to  $O(|E| \log |E|)$  via the **union-find data structure**.

# Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$  labeled “a”       $\{d, e, f\}$  labeled “e”

Supports three operations:

1. **Make-Set( $v$ )**: Adds a set  $\{v\}$  with label “v”

$\{a, b, c\}$  labeled “a”       $\{d, e, f\}$  labeled “e”       $\{v\}$  labeled “v”

2. **Union-Set( $u, v$ )**: Replaces sets including  $u$  and  $v$  with a new set that is union of both sets and labels this set by some element it contains. For example, the result of  $\text{Union-Set}(f, v)$  is

$\{a, b, c\}$  labeled “a”       $\{d, e, f, v\}$  labeled “e”

3. **Find( $v$ )**: Returns the label of the set including  $v$

# Kruskal's Algorithm with Union-Find

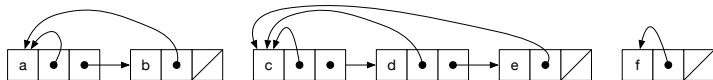
## Algorithm (Kruskal)

1. Sort edges by non-decreasing weight
2. For each vertex  $v \in V$ :  $\text{Make-Set}(v)$
3.  $F = \emptyset$
4. For each edge  $e = (u, v)$  in  $E$ 
  - 4.1 If  $\text{Find}(u) \neq \text{Find}(v)$  then  $\text{Union}(u, v)$  and  $F = F + e$

Well, how should we implement union-find...

## Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:
  - 2.1 name of element
  - 2.2 "label" pointer to label of the set
  - 2.3 "next" pointer to next node in list
3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.



1. **Make-Set( $v$ )**: Takes  $O(1)$  time to add a single node.
2. **Find( $v$ )**: Takes  $O(1)$  time to follow pointer to label.
3. **Union-Set( $u, v$ )**:  $O(\text{size of smaller set})$ .
  - ▶ Update "next" pointer at end of longer list to point to start of shorter list
  - ▶ Update "label" pointers of shorter list to point to label of other list
  - ▶ Update auxiliary pointers and size information

# Union-Find Analysis

## Theorem

Consider a sequence of  $m$  operations including  $n$  Make-Set operations. Total running time is  $O(m + n \log n)$ .

## Proof.

- ▶ Total time from Find and Make-Set:  $O(m)$
- ▶ Total time from Union:  $O(n \log n)$ 
  - ▶ Updating next pointers:  $O(n)$
  - ▶ Updating label pointers:  $O(n \log n)$  because the label pointer for a node can be updated at most  $\log_2 n$  times.



Hence, Kruskal's algorithm can be implemented in time

$$O(|E| \log |E|) + O(|E| + |V| \log |V|) = O(|E| \log |E|)$$

# Faster Implementation of Union Find

## Theorem

*There exists an implementation that, given a sequence of  $n$  Make-Set operations and  $m$  total operations, takes  $O(m\alpha(n))$  time where  $\alpha$  is the inverse Ackermann's function.*

## Definition (Ackermann's Function)

Define a sequence of functions:  $A_0(x) = 1 + x$  and

$$A_k(x) = A_{k-1}(A_{k-1}(\dots A_{k-1}(x) \dots))$$

where  $A_{k-1}$  is applied  $x$  times. E.g.,  $A_1(x) = 2x$ ,  $A_2(x) = 2^x x$ . Ackermann function is  $A(k) = A_k(2)$ .  $\alpha(n)$  is defined as smallest  $k$  such that  $A(k) \geq n$ .

## Example

$\alpha(n) \leq 4$  for all  $n \leq 2^{2^{2^{\dots^{2048}}}}$  where tower is of height 2048.



# Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
  - ▶ Each node encodes an element and pointer to the parent.
  - ▶ The element at the root is the label of the set.
1. **Make-Set( $v$ )**: Takes  $O(1)$  time to add a single node.
  2. **Find( $v$ )**: Takes  $O(d_v)$  time where  $d_v$  is the depth of  $v$
  3. **Union-Set( $u, v$ )**:  $O(d_v + d_u)$  time
    - ▶ Perform Find( $u$ ), and Find( $v$ )
    - ▶ Add pointer from root of smaller tree to root of larger tree

**Extra Trick!** Do *path compression*. When we do a Find operation, update the pointers from all nodes encountered to point to the root. Increases time by a constant factor but saves time for future Find operations.

**More Details:** See Section 21.4 of CLRS (3rd edition) or Section 5.1 of DPV.

# Blank Slide