Lecture Notes CS:5350 MaxMatching and Introduction to Randomized Algorithms
Lecture 10: Sep 26, 2019
Scribe: Daniel Illg

1 Connection between flows and matching

1.1 MaxMatching

The MaxMatching Problem

Input: A graph $G = (V, E)$

Output: A set $M \subseteq E$ such that no 2 edges in M are incident of the same vertex and the size of set M has been maximized.

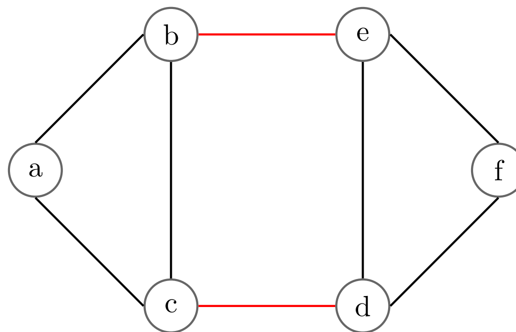

Figure 1: Example of a matching M containing the edges in red.

Figure 1 shows a matching M with the edges b-e and c-d. While this is a valid matching, it is not a maximum-sized matching. Another matching M exists such that there are three edges in M, demonstrated in Figure 2. Not only is this new M a maximum-sized matching, it is also *perfect matching*, or a matching such that for every vertex v in the graph, there is an edge incident on v in the matching.

It is not the case, however, that all maximum-sized matchings are perfect matchings. An example of a maximum-sized matching that does not contain edges incident to all vertices in V is shown in Figure 3.

Notes:

- The examples above depict a simple version of the MaxMatching problem. Other versions include problems such as finding a matching on a weighted graph such that the sum of the weights in the matching is maximum.
- MaxMatching on general graphs can be solved in polynomial time using Edmond's Algorithm.
- MaxMatching on bipartite graphs can be solved by reducing the problem to MaxFlow.

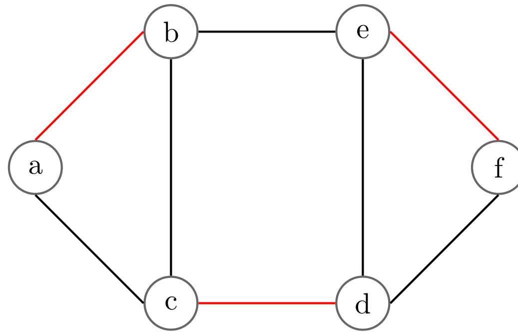

Figure 2: Example of a maximum-sized set M (indicated by the edges in red) such that M is a perfect matching.

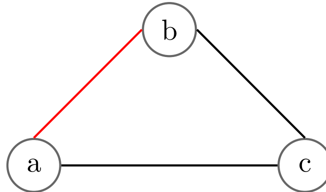

Figure 3: Example of a maximum-sized independent set M (indicated by the edges in red) such that M is not a perfect matching.

1.2 Matching on Bipartite Graphs

Definition: A Bipartite Graph $G = (V, E)$ is such that the vertex set can be partitioned into two sets L and R such that every edge has one endpoint in L and one endpoint in R.

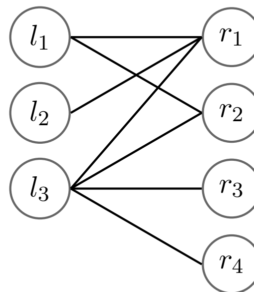

Figure 4: Example of a bipartite graph with the vertices forming set L in the left column and the vertices forming set R in the right column.

Note:

- The set of bipartite graphs is exactly the set of graphs that do not contain a cycle of odd length.

How to find a bipartite graph:
Use BFS (Breadth First Search). A graph G is bipartite if and only if in the search tree constructed by BFS there are no edges between vertices in the same level of the tree.

Figure 5 shows an example BFS search tree of a graph that is not bipartite. The edge in red indicates an edge in the graph between vertices in the same level of the tree. With this edge, it is clear to see that an odd cycle exists ($l_{2,1}$ - $l_{1,1}$ - root - $l_{1,2}$ - $l_{2,4}$ - $l_{2,1}$). The graph is therefore not bipartite.

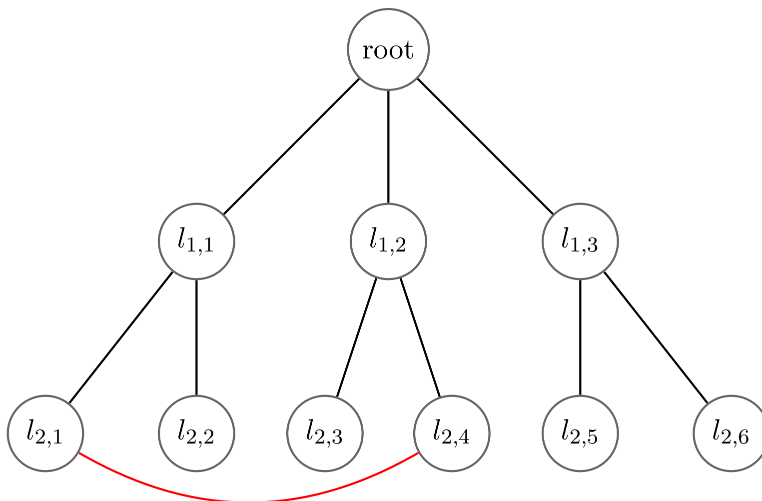

Figure 5: Example of a BFS search tree indicating that a graph is not bipartite since there is an edge between $l_{2,1}$ and $l_{2,4}$ in the graph. As these vertices are in the same level (level 2) of the BFS search tree, this graph is not bipartite.

Bipartite graphs can be recognized in linear time using this technique. Additionally, the (L, R) partition of a bipartite graph can be computed in linear time.

1.2.1 MaxMatching using MaxFlow on Bipartite Graphs

MaxMatching on Bipartite Graphs
Input: A bipartite graph $G = (L, R, E)$
Output: A maximum matching of G

Reduction to maxflow
We will show the transformation of the bipartite graph $G = (L, R, E)$ in Figure 4 into the flow network $H = (V, E', c, s, t)$ where

$$V = L \cup R \cup \{s, t\}$$

and

$$E' = \{(s, v) | v \in L\} \cup \{(v, t) | v \in R\} \cup \{(u, v) | \{u, v\} \in E, u \in L, v \in R\}$$

This new flow network can be seen in Figure 6. It is important to note that the capacities of all the edges in this flow network are exactly 1. After computing a maximum flow on this flow network, we will need to transform this flow into a matching. We can do this by taking all of the saturated edges $\{u, v\} \in E$ in the MaxFlow such that $u \in L$ and $v \in R$.

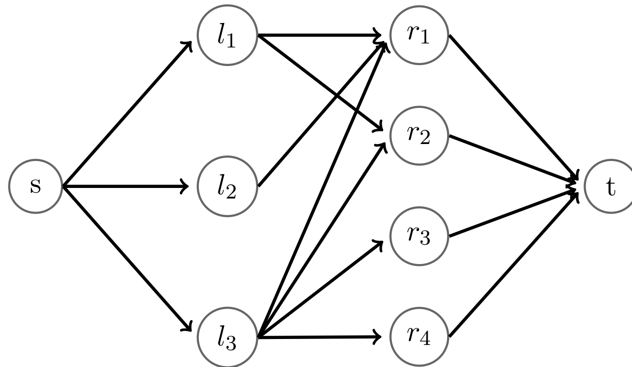

Figure 6: The bipartite graph from Figure 4 transformed into a Flow Network. Each directed edge in this graph has a flow capacity of exactly 1.

Why does this work? Since all capacities are integral, we can find a maximum flow f such that for each edge: $(u, v) \in L \times R$, $f(u, v) \in \{0, 1\}$
We can show that there is a matching M such that $M = \{\{u, v\} \in G | f(u, v) = 1\}$. Since the capacities of all the edges in the flow network are 1 and flow conversation must be upheld, every vertex u in the flow network such that $u \in L$ can push flow to at most one other vertex in the graph. This is because each vertex u in the flow network such that $u \in L$ can receive at most 1 flow from the source vertex s. Similarly, every vertex v in the flow network such that $v \in R$ can receive flow from at most one edge since v can push at most 1 flow to the target vertex t. Thus, a feasible flow on this flow network will result in a matching between vertices in L and R. The fact that f is a maxflow implies that M is a matching of maximum size.

For further applications of maxflow to matching, see the proof of Hall's Theorem in the posted readings.

2 Randomized Algorithms

To start the section on randomized algorithms, we will look at the *distributed maximal independent set* problem. The maximal independent set problem will first be briefly explained.

2.1 Maximal Independent Set

An *Independent Set* is a set of vertices such that no two vertices are connected by an edge. An independent set is *maximal* if we cannot add any other vertex to the set without violating the

above condition. Take for instance the independent set in Figure 7 defined by vertices a, b, and c. It is plain to see that none of the vertices in this set are connected and that no other vertex on the graph can be added to the set. So $\{a, b, c\}$ is a maximal independent set in this graph.

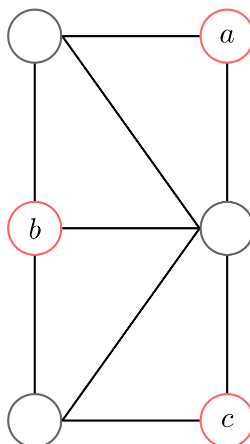

Figure 7: Example of a maximal independent set containing vertices a, b, and c.

An example greedy algorithm to solve this problem
1. Pick an arbitrary vertex v and add it to the set I
2. Mark all neighbors of v as not in the set
3. Repeat above steps until all vertices have been either placed in I or marked as not in I

Maximal Independent set is easy to solve in the sequential model of computation. Let us now look at solving this problem in a distributed model of computation.

2.2 Distributed Maximal Independent Set Problem

Consider the maximal independent set problem in a *distributed model of computation*, described below.
1. Each vertex in the graph represents a machine and each edge represents a communication link between machines.
2. Each machine has a unique ID from the set {1, 2, . . . , n}, where n is the number of vertices in the graph.
3. Each machine only has local information initially, i.e., machines only have know their own communication links and IDs.
4. Communication between machines is synchronous and occurs with each tick of a defined global clock. At each clock tick, each machine sets a message to each neighbor. A clock tick is called a *round*.
5. Each message is relatively small, bounded by $O(\log n)$ bits per message, and is capable of holding a constant number of id's.

We want to design an algorithm in this model so that at the end of the algorithm, some nodes set their status to ON, the remaining nodes set their status to OFF, and the nodes that have set their status to ON form a Maximal Independent Set.

Distributed Deterministic Maximal Independent Set Algorithm

1. Each node sends its ID to its neighbors
2. If a node has the lowest id in its neighborhood, it joins the MIS (i.e. it turns ON)
3. Nodes that join the MIS inform their neighbors
4. Nodes whose neighbors joined MIS turn OFF; these nodes will no longer participate in the algorithm
5. Repeat steps starting at 1 until all nodes are ON or OFF

Below is some sample pseudocode of the above algorithm. The code is written for a single node in the network, and all nodes in the network will be running this code concurrently. The send command referenced in lines 2 and 9 simply takes the value in the parenthesis and sends it to node n in n.send(). It is assumed that the node is able to receive messages synchronously and as it receives these messages, it stores the information for later use. Lines 3 and 10 reference received information, which will be the results of a neighboring node's send command. As both sending and receiving happen within a round, it is assumed that all nodes wait the entire duration of the round before continuing the rest of their code so that they will remain synchronized.

Distributed Deterministic Maximal Independent Set Pseudocode

1. while status != OFF or ON // while a node's status isn't confirmed
2. for each n in neighborhood, n.send(ID) // send the node's ID to all nodes in its neighborhood
3. store received IDs in IDList // no need to keep track of which ID belongs to which neighbor
4. join = true // if join is true at step 6, this node will join MIS
5. for each nid in IDList
    6. if nid < ID then join = false // if a neighbor has a lower ID, this node will not join MIS
7. end for
8. if join == true then status = ON // this node is joining MIS
9. for each n in neighborhood, n.send(join) // send out the nodes join status to its neighbors
10. if node receives true from a neighbor then status = OFF // if a neighbor sends out true, it has joined MIS so this node should turn off
11. end while

An example of this algorithm being performed on a graph is show in Figures 8, 9, and 10. A blue vertex represents the initial state, with the vertex being committed to neither the ON or OFF state. When a vertex turns red, it is declared ON. When a vertex turns black, it is declared OFF. Initially all nodes(vertices) will send out their IDs to their neighbors. In the first pass, only node 1 will turn ON. This is because node 1 is the only node with the smallest ID amongst its neighborhood in the graph. Once node 1 is turned ON, it will tell nodes 2, 3, and 5 to turn off. In the second pass of the algorithm, only nodes 4 and 6 remain. As 4 contains the smallest ID in its neighborhood, it turns ON and tells node 6 to turn off. This completes the algorithm as all nodes in the graph are ON or OFF.

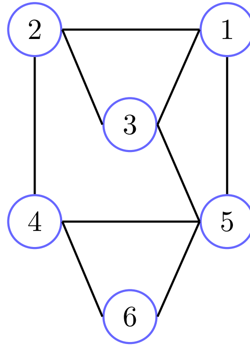

Figure 8: The starting states of the graph, where all nodes are neither ON nor OFF.

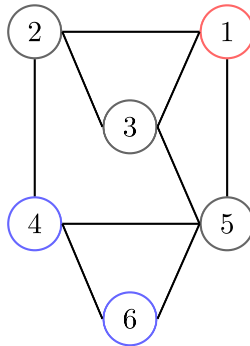

Figure 9: Turn on all nodes that have the smallest ID number in their neighborhoods. In the case of this graph, only node 1 will be turned ON. That node then informs its neighborhood to turn OFF.

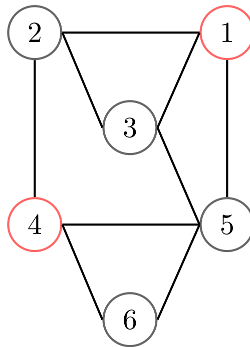

Figure 10: In the second pass of the algorithm, node 4 is now the node with the smallest ID number amongst its neighborhood of nodes that are neither ON nor OFF. Node 4 turns ON and informs its neighborhood to turn OFF.

This algorithm will take $O(n)$ rounds to complete. In each iteration, the clock ticks at Step 1 (when nodes send their id's to their neighbors) and at Step 3 (when nodes inform their neighbors if they have joined MIS)