Bipartite Perfect Matching in O(n log n) Randomized Time

Nikhil Bhargava and Elliot Marx

Background

Matching in bipartite graphs is a problem that has many distinct applications. Many problems can be reduced to prototypical allocation problems, where a set of players each maintain interest in a subset of available goods. These problems can be represented as bipartite graphs where edges denote a player's interest in a good, and based on the weights and distribution of these edges, finding an optimal allocation reduces to finding a maximum matching on the corresponding bipartite graph. This paper in particular focuses on computing matchings in d-regular bipartite graphs.

For the purposes of the paper and this follow-up report, we introduce the following notation. A bipartite graph G is represented as the tuple (P, Q, E) where P and Q represent the sets of nodes in the graph and E represents all of its undirected edges. As is the case with bipartite graphs, we assume that the intersection of P and Q is empty and every edge in E has exactly one endpoint in P and exactly one endpoint in Q. In particular though, we are interested in d-regular bipartite graphs. A d-regular bipartite graph is a bipartite graph where all nodes have degree d.

Note that for d-regular bipartite graphs, we first know that the size of the sets P and Q must be the same. Every edge has exactly one endpoint in P, so if we say there are n nodes in P, then there are dn edges connected to the nodes in P. Similarly all edges must have exactly one endpoint in Q, and since each node has degree d, this directly gives us that $|P| = |Q| = n$. For convenience, we say $m = nd$.

Another noteworthy property of d-regular bipartite graphs is that a perfect matching always exists. This property is simply seen through an application of max-flow principles. To demonstrate this,

we introduce two new nodes s and t, representing our source and sink respectively, and construct a directed edge from s to each node in P and a directed edge from each node in Q to t. We also orient all edges as going from P to Q, and for every edge in the new graph, we assign it a capacity of one. We know that a fractional flow of weight n exists where you push 1 unit of flow from s to each of the nodes of P, $\frac{1}{d}$ units of flow through each edge from P to Q, and 1 unit of flow from each of the nodes of Q to t. Since the capacities themselves are integers, we know an equally large flow with all integer values exists, so we can find a maximum flow of n where each edge has either one or zero units of flow. The edges with flow going over them represent our matching and since the total flow is n, we get that n edges from P to Q are used and thus we have a perfect matching.

The previous example not only shows that perfect matchings in d-regular bipartite graphs exist, but it also gives a construction for how to find such a matching by using max-flow algorithms, such as Fork-Fulkerson. The problem of finding perfect matchings in d-regular bipartite graphs has been studied for a long time with many varied approaches.

Another way to find perfect matchings is by constructing Euler tours. Imagine a d-regular bipartite graph where $d = 2^k$ for some integer k. We can find a perfect matching on our graph by successively computing Euler tours. An Euler tour is a path on a graph that traverses every edge exactly once and starts and ends at the same node. An Euler tour is known to exist whenever every node in a graph has even degree, and it is computable in $O(m)$ time.

Consider what happens when we compute an Euler tour on our 2^k-regular bipartite graph. Our resulting path orients all edges with half of the edges going from P to Q and the other half going from Q to P. Since every edge has one endpoint in P and one endpoint in Q, every step in the path either

transitions you from P to Q or from Q to P. Since we start and end at the same node, this means that exactly half of the edges are oriented in each direction. If we restrict our view to those edges going from P to Q, these edges also form a bipartite graph but now the graph is 2^{k-1}-regular instead. We know that the graph is regular because half of the edges previously adjacent to a particular node p were used to reach that node, and half were used to leave it. If it were not balanced equally, then our path would not have been an Euler tour since the number of times we go to a node has to exactly equal the number of times we leave it. Finally after k iterations, we eventually get a 1-regular bipartite graph. In this graph every node in P is connected to exactly one node in Q and vice versa. This means that the 1-bipartite graph we generate is in fact a perfect matching. Since the amount of work we do at each level is $O(m)$, we can write the recurrence relation as $T(m) = T(\frac{m}{2}) + O(m)$, which means that the algorithm overall runs in $O(m)$. In subsequent years, this algorithm was extended to find perfect matchings when d was not a power of 2 while still maintaining the $O(m)$ running time.

Prior to this paper, the best algorithm for finding a perfect matching in this type of graph was assumed to be $soft - O(min\{m, \frac{n^2}{d}\})$. However, this paper goes above and beyond the previous best by establishing an $O(n\ log\ n)$ expected running time for finding a perfect matching. Note that this is sublinear in the input size, which is $O(nd)$. Thus, with effective use of randomization, the authors were able to construct an algorithm capable of exactly finding a perfect matching without having to inspect the entire graph structure.

Main Algorithm

The main algorithm in this paper relies highly on the concept of an alternating random walk in the presence of a partial matching. After taking a sufficient number of alternating random walks, and

updating the recorded matches appropriately, eventually a perfect matching is obtained. The alternating random walk begins at an unmatched node in the graph, and at each step in the alternating random walk, an edge is randomly sampled from the current node's unmatched outgoing edges. Once an edge is selected, we follow that edge to the next node. If the subsequent node is unmatched, the alternating random walk is complete. However, in the case that the node we move to is matched, then we immediately add the matching edge to our walk, move along that edge and then repeat the process until we reach an unmatched node.

In the walk that we take, there exists a possibility that we momentarily walk along a cycle before reaching our end destination. In order to turn our walk into a path, at the end of the walk we excise any cycles from our walk so that our path becomes one from an unmatched node through some number of matched nodes before terminating at an unmatched node. The outputted path has one fewer edge belonging to the partial matching than edges that weren't a part of the partial matching. Every time we sample an edge we always sample among unmatched edges. After we follow the unmatched edge we either go back along a matched edge or terminate the algorithm. Thus the final path is of the form U(MU)* where U represents an unmatched edge and M represents a matched edge.

Now that we have our alternating random walk and it's free of cycles, we can update our matching. The way we do so is simple -- every edge that was unmatched in the alternating random walk becomes matched and every matched edge in the alternating random walk becomes unmatched. This is a valid operation since every node except the initial and final nodes in the random walk touched exactly one matched edge at the start of the algorithm, and one edge touching each matched node is removed from the matched set and one edge touching each matched node is added to the matched set. Further the initial and final nodes at the end of the algorithm also touch exactly one matched edge because

originally they were completely unmatched. Additionally because there was one more unmatched than matched edge initially, following this operation the total number of matched edges in the entire graph increases by one.

Having established an alternating random walk, we can now move on to the main algorithm presented in the paper. We begin with an empty matching. For each of n iterations, we compute an alternating random walk on our d-regular bipartite graph, G. We take the random walk, remove cycles and use the resulting path to update our matching as described previously. After n iterations, the process is complete and we have a perfect matching on G. It is apparent from its construction that upon completion, this algorithm will output a perfect matching, but the important claim made is that this algorithm will do so in $O(n \log n)$ time.

Proof of Running Time

The main claim of the paper is that their algorithm to find a perfect matching does so in randomized expected $O(n \log n)$ time. This fact can be derived from a key lemma of their paper, which is that an alternating random walk in a d-regular graph where $2k$ nodes are unmatched (that is to say $n-k$ pairs have already been matched together) takes on expectation $1+\frac{n}{k}$ steps. If this lemma is true, then it is clearly $O(n \log n)$. We can sum over the expectation when there are $2k$ unmatched nodes, $2k-2$, $2k-4$, …, all the way down to 2 unmatched nodes and we get that the total number of expected steps is $\sum_{k=1}^{n} 1 + n/k = n + nH_n \in O(n \log n)$.

The Matching Graph

To establish the overall expected running time, it then suffices to show that an alternating random walk in a d-regular graph with $2k$ unmatched nodes takes expected $1 + \frac{n}{k}$ steps. In order to simplify the analysis, we will construct a directed graph $H = (V', E')$ from our original graph $G = (P, Q, E)$ and our partial matching M on G. To construct H, we first take all edges in E and orient them from P to Q. Then, we take all of our matched edges from E and contract them into supernodes on our new graph. Finally, we add new source and sink nodes s and t and add d edges from s to each node in P and d edges from each node in Q to t. Note that in adding s and t, we do not add any edges connected to supernodes.

The analysis graph, H, that we constructed will be useful to prove properties about our original graph, G. Note that the original algorithm only concerns itself with G. It does not construct H, as it only needs to take random walks on the input G. H is only used for formal analysis. The most important property of H, which will allow us to derive the expected number of steps taken by the alternating random walk algorithm, is that an alternating random walk in the input graph G uniquely corresponds to a path from s to t on the analysis graph H and vice versa.

Consider an alternating random walk on our input graph G. Without loss of generality we can assume that our alternating random walk starts at a node from P. If we started from a node in Q, we can construct an equivalent graph where the nodes in P are labeled as belonging to Q and vice versa. We start our random walk then by choosing a random unmatched node in P. This is represented analogously in H by randomly selecting an edge from s to a node in P since all of the nodes that s is connected to are unmatched in the original graph G. At every step in our walk, we randomly sample along unmatched edges. If we reach an unmatched node in Q, then we're done. On the analysis graph

H, this happens when we continue our path directly to a node in Q. Once we reach Q, we take the only available outgoing edge which brings us to the end node t.

Now consider the case where we didn't immediately reach an unmatched node. In the case of the alternating random walk on G, when we take an edge and reach a matched node, we immediately take the edge corresponding to the matching to return to P. In the case of the analysis graph, taking an edge to a matched node represents taking an edge to a supernode. Since the supernode is a contraction of nodes that are matched, taking the path back is the same as remaining at the supernode. When we then sample among unmatched outgoing edges, this is the same as taking an outgoing edge from the supernode since all outgoing edges originally belonged to a node from P and the supernode has all outgoing edges except the one contained within the matching. Thus, any alternating random walk in the algorithm graph G directly corresponds to a path from s to t in the analysis graph H.

Similarly every path on H from s to t corresponds to exactly one alternating random walk. From s you can take an edge only to one of the nodes in P that wasn't matched. This corresponds to randomly selecting an unmatched node in our original graph since without loss of generality we can assume the randomly unmatched node comes from P. In our analysis graph, from a node in P we can then transition either to a node in Q that wasn't matched, which corresponds to reaching an unmatched final node, or we can transition to a supernode, which corresponds to reaching a matched node. The only outgoing edges from a supernode are the outgoing ones belonging to the node from P. Further the outgoing edges do not include the edge that was contracted into the supernode, so taking any edge from a supernode corresponds to jumping back across the implicit matched edge and then randomly sampling over all unmatched edges. Eventually a supernode transitions to a node in Q and the only way for the

path to end is to go to t, which represents the end of our algorithm. Thus, there is a one-to-one correspondence between paths on H and alternating random walks.

Proof Details

The relationship between the two graphs is useful because while it's difficult to reason about the expected number of steps taken in an alternating random walk, it's much easier to reason about the expected number of steps in a path from s to t on a directed graph. The general approach for determining the total number of expected steps in an alternating random walk goes as follows. We first consider the probability of arriving at each node, in our analysis graph H, at a particular time. Then we determine the number of times that we expect to visit each node in an entire alternating random walk. Finally, by summing over all nodes, we arrive at the total number of steps taken during one alternating random walk.

Since, in the analysis graph, our path is determined by randomly sampling among outgoing edges, we can construct a matrix P that represents the probability of transitioning from one node to another at a particular time step. We define P_{ij} to be $\frac{1}{deg(i)}$ whenever there is an edge from i to j in H, and alternative $P_{ij} = 0$ when there is no edge from i to j. Note that for the node t, $P_{tj} = 0$ for any choice of j since t has no outgoing edges. We can then define the probability that our algorithm reaches node i from node j after one timestep as $e_i P^T e_j$ where e_k is a vector of all zeros with a one at index k.

To determine the probability that our algorithm reaches node i from node j after t time steps, we simulate another transition by multiplying in additional factors of P; the probability can then be expressed as $e_i(P^T)^t e_j$. Further, because every alternating random walk corresponds to a path starting at s in H, the expected probability of being at a particular node i after t steps of our algorithm is exactly $e_i(P^T)^t e_s$.

By linearity of expectation, the number of times we visit node i is given by $\sum_{t=0}^{\infty} e_i(P^T)^t e_s = e_i(\sum_{t=0}^{\infty}(P^T)^t)e_s$.

However, this infinite sum only converges if $(P^T)^t$ converges to 0 as t tends to infinity.

We know that row t of P is zero by construction, so it is equivalent to show that t is reachable from any node in a finite number of steps. For this, we examine the specific structure of the analysis graph. We know that from s we must transition to a node in P. From P, we must transition to a supernode or a node in Q. From a supernode, we must transition to another supernode or a node in Q. Finally, from Q we must transition directly to t. We can demarcate a node as belonging to one of 5 zones (s, P, supernode, Q, t), and all edges move us across zones closer to t except for those transitioning from supernode to supernode. There must be a supernode with an edge to a node in Q. Since the analysis was constructed so that the indegree equals the outdegree of every node if there were no outgoing edge to Q from a supernode, then all the incoming edges to supernodes would be outgoing edges from other supernodes and we would never reach a supernode. Thus, if we reach a supernode, we know we can eventually reach Q. Since the number of supernodes is bounded by n, we know that within n steps, Q is reachable from a supernode. Thus, t is reachable in a finite number of steps (namely $n+3$ steps) from every node in the graph.

Since $\lim_{k\to\infty}(P^T)^k = 0$, we know that

$(I - P^T)\sum_{t=0}^{\infty}(P^T)^t = \sum_{t=0}^{\infty}(P^T)^t - (P^T)^{t+1} = I - \lim_{k\to\infty}(P^T)^k = I$. In other words, we have

$\sum_{t=0}^{\infty}(P^T)^t = (I - P^T)^{-1}$, so for each node we can compute the expected number of visits to that node by $e_j(I - P^T)^{-1}e_s$. However, we also know that for any node j, $e_j(I - P^T)^{-1}e_t = 0$. Thus, we can instead focus on finding $e_j(I - P^T)^{-1}(e_s - e_t)$. Notably, this value is exactly equal to $\frac{deg(j)}{dk}$.

Arriving at this expression is nontrivial, and it holds for all possible values of j. Here we will just show that the equality holds when $j \neq s,t$ in order to demonstrate the final conclusion. We will let $deg(\cdot)$ be a vector of the outdegrees of each node. To show that $e_j(I - P^T)^{-1}(e_s - e_t) = \frac{deg(j)}{dk}$, it is sufficient to show that $e_s - e_t = (I - P^T)\frac{deg(\cdot)}{dk}$. For a fixed j, we know that $e_j P^T deg(\cdot)$ can be rewritten as $\sum_i P_{ij} deg(i)$. Since $P_{ij} = \frac{1}{deg(i)}$ when (i, j) is an edge in the graph, $\sum_i P_{ij} deg(i)$ is equal to the sum of node j's incoming edges. However by the construction of the analysis graph, for any $j \neq s,t$ the indegree of j is exactly equal to the outdegree of j. Thus, we get that $e_j P^T deg(\cdot) = deg(j)$. Returning to what we originally wanted to show, for each j we have that $e_j(e_s - e_t) = 0 = e_j\frac{deg(\cdot)}{dk} - e_j P^T\frac{deg(\cdot)}{dk}$. Thus, $e_s - e_t = (I - P^T)\frac{deg(\cdot)}{dk}$ and $e_j(I - P^T)^{-1}(e_s - e_t) = \frac{deg(j)}{dk}$.

Once we know that the expected number of times we visit a node is $\frac{deg(j)}{dk}$, then we know that the expected number of steps taken in a single run of alternating random walk is $\sum_j \frac{deg(j)}{dk}$, where $j \neq s,t$. From our analysis graph, it is straightforward to compute the sum of all the degrees. If there are $2k$ unmatched nodes, then each of those nodes have degree exactly d. With $2k$ unmatched nodes, we have $n - k$ supernodes each of which have degree $d - 1$. The sum of the outdegrees of all nodes is $2kd + (n-k)(d-1) = kd + dn + k - n = n(d-1) + k(d+1)$. Since $n \geq k$, we know $n(d-1) + k(d+1) \leq d(n+k)$. Thus the expected number of steps in a single run of alternating random walk is bounded above by $\frac{d(n+k)}{dk} = \frac{n+k}{k} = 1 + \frac{n}{k}$.

Since we know that the expected number of steps in a run of alternating random walk is bounded above by $1 + \frac{n}{k}$ when there are $2k$ unmatched nodes, we can conclude that the alternating

random walk algorithm in sum finds a perfect matching on d-regular bipartite graphs in expected $O(n\ log\ n)$ time.

Conclusion

This algorithm improved significantly over any known previous algorithms. The previous best ran in time $soft - O(min\{m, \frac{n^2}{d}\})$, while this algorithm runs in time $O(n\ log\ n)$ expected. Especially impressive is that this algorithm is sublinear in the size of the input, which is of size *nd*. Note that this algorithm breaks down when the graph is not regular. When the graph is regular the likelihood of transitioning to an unmatched node at any point in our alternating random walk is at least equal to the proportion of unmatched nodes in the graph, but when it isn't, we lose this guarantee.

We implemented the algorithm from the paper, and tested the effectiveness of the algorithm outside its specified usage. Specifically, we created 1000 goods and players, and created an edge between the i'th player and the i'th good. Then, we added a sparse graph over these perfect matching edges, with probability shown below in the table. The results show that with 1000 nodes, the probability of recovering a perfect matching is above 94 percent for including edges with probability less than 0.01. We also see that increased sparseness improves the efficiency of the algorithm. Running this with n=2000, we saw the probabilities drop to 83 percent with 0.002 percent edges included, and 63 percent with 0.004 edges included. The code for producing these results is attached.