1 Summary

In this lecture we discuss a sublinear time algorithm for finding the size of a maximal matching in a sparse graph. We start by describing a greedy algorithm to find a maximal matching, and then develop a sublinear time algorithm based on it. Our sublinear algorithm will approximate the size of a maximal matching to within an additive error of $\epsilon n$ and will run in $2^{O(d)}/\epsilon^2$ where $d$ is the maximum degree of the graph (hence why our graph must be sparse).

2 Maximal Matching

A matching in a graph is a subset of a graph’s edges such that no two edges in the matching share a vertex. More formally, given a graph $G = (V,E)$, $M \subseteq E$ is a matching if $\forall (u,v),(w,x) \in M\{u,v\} \cap \{w,x\} = \emptyset$. We call a matching $M$ maximal if we cannot add any more edges to it; i.e. $\not\exists e \in E\setminus M$ such that $\{e\} \cup M$ is a matching. This should not be confused with a maximum matching, which is a matching of maximum possible size.

3 Greedy Algorithm

The greedy algorithm to find a maximal matching is obvious: we simply keep adding edges to the matching until we can’t anymore. This can be done in a single loop as follows:

$M \leftarrow \emptyset$

for $e = (u,v) \in E$ do
  if neither $u$ nor $v$ is matched then
    $M \leftarrow \{e\} \cup M$
  end if
end for

This algorithm clearly produces a maximal matching as any edges not in the matching were checked and rejected as they could not be added to the matching. However, this algorithm runs in $O(E)$ time while we desire a sublinear algorithm. We will see in the next section how we will make this sublinear.
4 Sublinear Algorithm

The basic idea is to take a small random subset of the vertices and for each one check if it is in the greedy solution. We then estimate the size of the complete maximal matching to be roughly the fraction of vertices we found to be in the greedy solution scaled times the total number of vertices and then divided by two as each edge in the matching hits two vertices. To be precise:

\[ c \leftarrow 0 \]
\[ S \leftarrow t = O(1/\epsilon^2) \text{ random vertices} \]
\[ \text{for } v \in S \text{ do} \]
\[ \quad \text{if } O((u,v)) \text{ returns } true \text{ for any } u \text{ then} \]
\[ \quad \quad c \leftarrow c + 1 \]
\[ \quad \text{end if} \]
\[ \text{end for} \]
\[ f \leftarrow c/t \]
\[ \text{return } f/2 \cdot n + \epsilon/2 \cdot n \]

Where \( O(v) \) is an oracle that returns whether or not \( v \) is in the maximal matching returned by the greedy algorithm and the \( \epsilon/2 \cdot n \) term is a corrective factor that we will explain in the analysis later.

4.1 The Oracle

In order for this algorithm to work the oracle must be consistent such that if it was queried for every single edges in the graph we would have an exact maximal matching. However, whether or not a given edge is added to the matching by the greedy algorithm may depend on the algorithm’s decision for every edge it inspects prior to that one. Thus, if we go about executing the oracle naively each query to it may take \( O(E) \) time.

The key observation in making an efficient oracle is that it doesn’t matter what order the greedy algorithm visits the edges in as long as its consistent. We then can choose this order randomly. The advantage of a random ordering is that as we will see, the dependency chains the oracle must follow to answer a query are short.

We compute our randomized order by assigning each edge \( e \) a random value \( r_e \in [0,1] \) and saying that an edge \( e_1 \) comes before another edge \( e_2 \) in the ordering if and only if \( r_{e_1} < r_{e_2} \). Our oracle then can determine if an edge is in the matching by recursively checking if any of its neighbors of lower \( r_e \) are in the matching. This recursion is terminated once we reach an edge with no neighbors of lower \( r_e \) as then that edge must be in the matching. To preserve the sublinear time nature of our algorithm, we don’t actually assign any values \( r_e \) in advance, but instead calculate them on the fly, only assigning \( r_e \) to an edge when it is inspected for the first time.

Oracle pseudocode:

\[ \text{for } e' \in E \text{ such that } e' \text{ is adjacent to } e \text{ do} \]
\[ \quad \text{if } r_{e'} < r_e \text{ and } O(e') \text{ returns } true \text{ then} \]
\[ \quad \text{return } false \]
end if
end for
return true

4.2 Correctness

The oracle accurately simulates the greedy algorithm by construction. Therefore by sampling $O(1/\epsilon^2)$ vertices, we have with high probability by a Hoeffding bound:

$$|MM(G)| - \epsilon / 2 \cdot n \leq f / 2 \cdot n \leq |MM(G)| + \epsilon / 2 \cdot n. \quad (1)$$

Where $f$ is the fraction of sampled vertices in the matching and $|MM(G)|$ is the size of some maximal matching. Since our algorithm outputs $f / 2 \cdot n + \epsilon / 2 \cdot n$, we thus have

$$|MM(G)| \leq f / 2 \cdot n + \epsilon / 2 \cdot n \leq |MM(G)| + \epsilon \cdot n. \quad (2)$$

Which gives us our desired additive $\epsilon n$ approximation error.

4.3 Complexity

We will prove that the expected amount of work done per query is $2^O(d)$ where $d$ is the maximum degree in the graph. We first observe that in a given query

$$\Pr[\text{a given path of length } k \text{ is explored}] \leq 1/k!, \quad (3)$$

because each edge has a random value so every possible ordering of the edges in the path is equally likely. Therefore the probability that we get the one ordering that puts the edges in that path in descending order is $1/k!$. Next we observe that

$$\text{number of edges within distance } k \text{ of } e \leq d^k, \quad (4)$$

which follows naturally from the fact that the maximum degree is $d$. Putting these two facts together, we have

$$E[\text{number of edges explored within distance } k] \leq d^k / k!, \quad (5)$$

and thus

$$E[\text{number of explored edges}] \leq \sum_{k=0}^{\infty} d^k / k! \leq e^d / d. \quad (6)$$

We are not quite done, as each explored edge must check each of its neighbors, regardless of whether or not that neighbor gets explored thus,

$$E[\text{query complexity}] \leq O(d) \cdot e^d / d = 2^O(d). \quad (7)$$

Therefore as we perform $O(1/\epsilon^2)$ such queries, the total complexity of the algorithm is $2^{O(d)}/\epsilon^2$.  

3
4.4 Further Work

In practice, we don’t actually need the oracle to always be accurate so long as it is almost always accurate. We could observe when the oracle is taking too long and then answer arbitrarily so long as it doesn’t happen to often. It has also been shown that by recursing on the neighbor with the minimum edge value at each step the number of queries is reduced to $O(d^3/\epsilon^2)$ [YYI09]. There are even better approaches for minor-free graphs [HKNO09].

This basic framework can be extended to other sparse problems such as maximum matching, vertex cover and set cover.

References
