Lecture 16 — Bipartite Graphs, Error Correcting Codes, and Set Cover

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 20, 2011

Today:

- Bipartite Graphs (bigraphs)
- Bigraphs in error correcting codes
- Bigraphs in set cover

1 Bipartite Graphs

A bipartite graph is a graph with two sets of vertices which are connected to each other, but not within themselves. More formally a bipartite graph $G = (U, V, E)$, consists of a set of vertices U a disjoint set of vertices V and a set of edges $E \subset U \times V$. As with regular graphs we might or might not put weights or other labels on the edges. Bipartite graphs have many applications. They are often used to represent binary relations between two types of objects. A *binary relation* between two sets A and B is a subset of $A \times B$. We can see that this is equivalent to the definition of bipartite graphs as long as A and B are disjoint (i.e. $A \cap B = \emptyset$). In the next lecture we will note that bipartite graphs (and hence binary relations) can also be viewed as matrices. Which view is "best" depends on the context or even on what someone is comfortable with. However, often using the graph view is more intuitive since it gives a pictorial interpretation of what is going on, instead of just a set of pairs.

Here are just a few examples applications of bipartite graphs:

Document/Term Graphs: Here U are documents and V are terms or words, and there is an edge (u, v) if the word v is in the document u. Such graphs are use often to analyze text, for example to cluster the documents.

Movies preferences: In 2009 Netflix gave a $1Million prize to the group that was best able to predict how much someone would enjoy a movie based on their preferences. This can be viewed, and in the submissions often was, as a bipartite graph problem. The viewers are the vertices U and the movies the vertices V and there is an edge from u to v if u viewed v. In this case the edges are weighted by the rating the viewer gave. The winner was algorithm called "BellKor's Pragmatic Chaos". In the real problem they also had temporal information about when a person made a rating, and this turned out to help.

Error correcting codes: In low density parity check (LDPC) codes the vertices U are bits of information that need to be preserved and corrected if corrupted, and the vertices V are parity checks. By using the parity checks errors can be corrected if some of the bits are corrupted. LDPC codes are used in satellite TV transmission, the relatively new 10G Ethernet standard, and part of the WiFi 802.11 standard. We discuss this in this class.

Students and classes: We might create a graph that maps every student to the classes they are taking. Such a graph can be used to determine conflicts, e.g. when classes cannot be scheduled together.

Stable marriage and other matching problems: You've seen this in 251. This is an approach used for matching graduating medical students to resident positions in hospitals. Here U are the students and V the resident slots at hospitals and the edges are the rankings of both the hospitals and residents.

Set Cover: to be covered in this lecture.

2 Error Correcting and LDPC Codes

Here we will only cover a brief introduction to error correcting codes and specifically low density parity check (LDPC) codes. Assume we are given n bits that we want to transmit, but that our transmission media (e.g. wires or radio) is noisy. Such noise will corrupt some of the bits by either making them unreadable, or even worse, flipping them. As industry gets more and more aggressive about how much information they want to transmit over a fixed "sized" channel, the more likely it becomes that signals will be corrupted. In fact theory says that you are better off transmitting at a high enough rate that you will regularly get errors, and then correcting them, than you would be by being conservative and rarely getting errors. Indeed this is part of the reason transmission rates have improved significantly over the years.

The idea of all error correcting codes is to add extra bits of information that can be used to fix certain errors. As long as the number of bits that are corrupted during transmission is small enough then the error can be fixed. Exactly how many errors can be fixed depends on the number of extra bits as well as the specific type of code that is used. We will use c to indicate the number of extra bits. In error correcting codes we imagine the following model:

```
 Input
  |  n-bit message
  v
ENCODER
  |  n + c bit codeword
  v
NOISY CHANNEL
  | possibly corrupted n+c bit codeword
  v
FIX ERRORS
  |  valid n+c bit codeword, hopefully matches
  v       codeword before errors
DECODER
  | n-bit message extracted from codeword
  v          hopefully matches input
 Output
```

In some codes the codeword is simply the original message with some extra bits tagged on (systematic codes) but generally the encoder might arbitrarily transform the message. In the remainder of our discussion we are only going to be interested in the codewords and the step that fixes the errors and not about how to encode and decode.

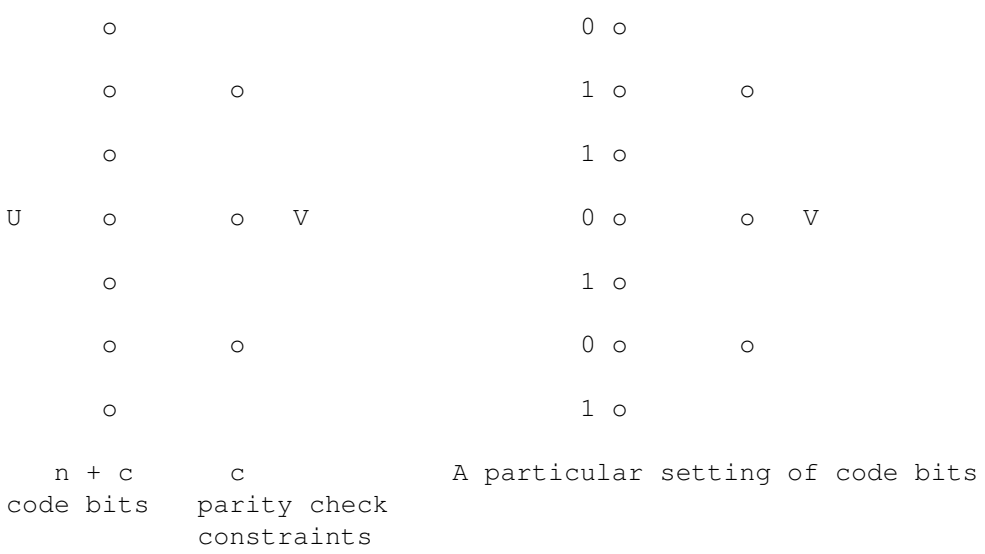

Figure 1: The bipartite graph corresponding to a low density parity check code. The edges between the code bits and parity check constraints are not shown.

LDPC codes. We now consider LDPC codes specifically. An LDPC coding scheme is defined by a bipartite graph. This graph is fixed for all time, i.e. all encoders and decoders use the same graph. In the graph there are $n + c$ vertices on the left (U), c vertices on the right (V), and edges between them (E). Figure 1 gives an example. Typically c will be some factor smaller than n, e.g. $c = n/4$. We now consider the meaning of each of U, V and E.

The vertices U on the left correspond to the bits of a codeword, and hence there are $n + c$ of them. These will be exactly the bits we send over our noisy channel and which might get corrupted. Any particular input message of length n will map to a particular codeword of length $n + c$. The right side of Figure 1 shows a particular setting of the bits. We will not explain how to generate an $(n + c)$-bit codeword from an n-bit message, but will instead just explain how we fix a codeword after it has been corrupted by noise, which is the more interesting part.

Now lets consider the vertices V on the right. Each of these corresponds to a *parity check* constraint. Each parity check on the right puts a constraints on its neighbors. In particular it requires that the neighbors must add to an even number. This is called even parity. This means that only certain code words on the left are valid since each valid word has to satisfy all constraints. You might think that it might be impossible to satisfy all constraints, but note that there are $n + c$ code bits, and only c constraints. This implies that there are n degrees of freedom. Think of this as having $n + c$ variables and c equations. This is why we can encode inputs of n bits.

Finally the edges between the code bits and the parity check constraints are selected so that every code bit has some fixed reasonably small degree k. Furthermore the edges are selected so that any small subset of code bits on the left do not share very many parity check constraints on the right. This notion of not sharing too many parity check constraints can be formalized by using so called *expander graphs*. Expander graphs are beyond the scope of this class, but are basically graphs in which any small enough subset of vertices X must have a neighborhood size that is a constant factor larger than $|X|$. They have significant importance in many areas of theoretical computer science.

We note that the term "low-density parity check" comes from the use of the parity check constraints and the fact that the vertices have low degree.

Now imagine that we start with a valid codeword that satisfy all the parity constraints and we send it across a noisy channel. Some of the bits will be corrupted either being too noisy to read or flipped. Now when we receive the message we can detect errors since we either can't read them or if flipped we will notice that some of the constraints are no longer satisfied. What is interesting is that we will not only be able to determine which bits are bad, but we will be able to fix the errors. In the following discussion we say that a constraint is *violated* if it is not satisfied (i.e. has odd instead of even parity).

We just consider the case when bits are flipped. In this case we don't even immediately know what bits are bad, but we will notice that some constraints are violated. Our goal is to determine which bits to flip back so that all constraints are satisfied again. Here is a simple rule for deciding which bits to flip. Each code bit on the left counts how many of its neighboring parity constraints are violated. We say a code bit is *critical* if more than half of its neighbors are violated. Now simply flip bits that are critical—we can either do this sequentially or in parallel. Repeat this process until no more constraints are violated. It turns out that under certain conditions this techniques is guaranteed to converge on the original code word. In particular the conditions are:

1. the number of errors is less than $c/4$.
2. only flip two bits in parallel if they don't share a constraint
3. the bipartite graph has good expansion (theory beyond the scope of this class)

Exercise 1. *Argue that if we only flip one bit at a time, the number of violated parity constraints goes down on each step.*

There is much more to say about error correcting codes and even LDPC codes. We did not explain how to generate a codeword from the message or the inverse of generating a message from a codeword. This can be done with some matrix operations, and is not that hard, but a bit tedious. We also did not explain what it means to be an expander graph and prove that an expander graph gives us the guarantees about convergence. We encourage you to take more advanced theory courses to figure this one out. The main purpose of the example is to give a widely used real-world example of the use of bipartite graphs.

3 Set Cover

An important problems in both theory and practice is the set cover problem. The problem is given n sets of elements for which the union of all sets is U, determine the smallest number of these sets for which the union is still U. For example we might have the sets:

$$\{a, c, d\}$$
$$\{a, f\}$$
$$\{a, b, d, e\}$$
$$\{b, e, f\}$$
$$\{d, e\}$$

which gives $U = \{a, b, c, d, e, f\}$, and the smallest number of sets that covers U is two. More formally for a set of sets $\mathcal{S}$ for which the *universe* is $\mathcal{U} = \bigcup_{s \in \mathcal{S}} s$, we say that C is a *cover* if $\bigcup_{s \in \mathcal{C}} s = \mathcal{U}$. The

set cover problem is to find a cover of minimum size. There is also a weighted version in which each element of S has a weight and one wants to find the a cover that minimizes the sum of these weights. Unfortunately, both versions of set cover are NP-complete so there is probably no polynomial work algorithm to solve them exactly. However, set cover and variants are still widely used in practice. Often approximation algorithms are used to solve the problems non-optimally. Other times the problems can be solved optimally, especially if there is some structure to the instance that makes it easy.

Various forms of set cover are widely used by businesses to allocated resources. For example consider the problem of deciding where to put distribution centers. Sunbucks, the well known tea shop, decides it needs a distribution center within 50 miles of every one of its stores. After searching for real estate on the web it comes up with n possible locations where to put a distribution center. Each location covers some number of stores. It wants to select some subset of those locations that cover all its stores. Of course it wants to spend as little as possible so it want to minimize the number of such centers. Set cover is also used for crew scheduling on airlines: the sets correspond to multihop cycles a crew member can take, and the elements are the flight legs that need to be covered.

The set location problem can be viewed as a bipartite graph $G = (V, U, E)$. We can put the sets on the left (V) and the elements on the right (U), and an edge from v to u if set v includes element u. For example the above sets would be.

A cover is then any set of vertices on the left whose neighborhood covers all of U, i.e. $V' \subset V, s.t. N(G, V') = U$. Such a view can be helpful in developing algorithms for the problem.

We now discuss one such algorithm. It is a greedy approximation algorithm. It typically does not find the optimal solution, but is guaranteed to find a solution that is within a factor of $\ln n$ of optimal. It was one of the earliest approximation algorithms for which a provable bound on the quality was shown. Furthermore it has since been shown that unless P = NP we cannot do any better than this in polynomial time. The algorithm is very simple:

```
1  fun greedySetCover(V, U, E) =
2  if |E| = 0 then {}
3  else
4     let
5        % select a vertex v in V with the largest neighborhood
6        val v = argmax_{v∈V} |N(v)|
7        % remove v and N(v) from G
8        val (V', U') = (V \ {v}, U \ N(v))
9        % remove unconnected edges
10       val E' = {(u, v) ∈ E | v ∈ V' ∨ u ∈ U'}
11    in
12       {v} ∪ greedySetCover(V', U', E')
13    end
```

If described in terms of sets, roughly, on each step this algorithm selects the set that covers the most remaining elements, removes this set and all the elements it covers, and repeats until there are no more elements left.

What is the cost of this algorithm as described? If we naively search for the vertex with the largest neighborhood then each step will require $O(m)$ work, where $m = |E|$. There will be as many steps as sets we end up picking. By being smarter and using a priority queue, the algorithm can be implemented to run with $O(m \log n)$ work. The algorithm is inherently sequential, but there are parallel variants of the algorithm that sacrifice some small constant factor in the approximation ratio.

So how good is the solution given by this algorithm. It turns out that it is not very hard to show that it guarantees a solution that is within logarithmic factor of optimal.

Theorem 3.1. *Given a set cover instance $G = (V, U, E)$, let k be the number of sets needed by the optimal set cover solution for G, then $|\texttt{greedySetCover}(V, U, E)| \leq k(1 + \ln |U|)$.*

We give a proof of this theorem below although this wasn't covered in class.

Proof. To bound the number of sets used in our solution, we write a recurrence for the number of recursive calls to `greedySetCover` and then solve it. Since each call adds one element to the result, this bounds the size of the solution. Consider a step of the algorithm in which $|U| = n$. The largest set (neighborhood of a vertex in V) has to be of size at least n/k. This is because there is a solution using at most k sets (the optimal one), so on average, the size of the sets in this solution is n/k and hence, the largest has to be at least n/k. Therefore we can write the following recurrence based on n for the number of rounds.

$$\begin{aligned} R(n) &\leq R\left(n - \frac{n}{k}\right) + 1 \\ R(1) &= 1 \end{aligned}$$

To solve this recurrence, we note that each round reduces the size of the size of the universe by a factor of $(1 - \frac{1}{k})$, so after r rounds, the number of elements in the universe is at most

$$n\left(1 - \frac{1}{k}\right)^r$$

Therefore, the number of rounds r^* to reduce the size to below 1 can be determined by finding the smallest r^* such that

$$n\left(1 - \frac{1}{k}\right)^r < 1$$

To solve this, we apply the super useful inequality $1 + x \leq \exp(x)$, so we have $n\left(1 - \frac{1}{k}\right)^r \leq ne^{-r/k}$. Since $r = k(\ln n + 1)$ gives that $n \exp(-\frac{1}{k} \cdot k(\ln n + 1)) < 1$, we know that $r^* \leq r = k(\ln +1)$. This allows us to conclude that $|\texttt{greedySetCover}(V, U, E)| = R(|U|) \leq k(1 + \ln |U|)$ and we have our proof. □