Mostly Multilevel Graph Partitioning

Today we will cover more on graph separators, discussing:

- *A bad case for the Kernighan-Lin heuristic,*
- *Multilevel graph partitioning for finding edge separators,*
- *A quick overview of spectral partitioning,*
- And an application: *how to compress graphs with large separators.* (This is given as a new defense of why one might bother finding graph partitions and separators.) The work covered here will appear in *SODA 2003* [1].

A bad case for Kernighan-Lin

Last time we talked about the Kernighan-Lin heuristic for finding small edge separators. The basic idea was to take an arbitrary bisection of the graph, and try to improve the cut with local search steps.

More precisely, let $G = (V, E)$ be the graph and S denote a bisection (so $|S| = |V - S| = |V|/2$). This bisection has some set of edges $\delta(S)$ crossing it from S to $V - S$. We consider each pair of $s \in S$ and $t \in V - S$, such that $(s, t) \in \delta(S)$, and determine the value $\text{gain}(s, t) = |\delta((S \cup \{t\}) - \{s\})| - |\delta(S)|$—i.e. how much we will reduce the size of the cut by swapping the pair (or increase if negative). We then consider all pairs starting at the one with highest gain and swap, with the condition that once a vertex has been swapped, it cannot be swapped again. We then back up to the point at which we had the smallest separator. During the algorithm we can take steps that increase the size of the separator.

This heuristic tends to do poorly when the initial cut is bad; here's an example of a particularly bad case with a $2 \times n$ mesh. Suppose we have the graph (V, E) with $V = \{(i, j) \ : \ 1 \leq i \leq 2, 1 \leq j \leq n\}$, and E consists of those pairs which only differ in one coordinate (e.g. $((1, 1), (1, 2))$, $((1, 1), (2, 1))$, $((2, 1), (2, 2))$, etc.) Assume for the sake of simplicity that n is even; then the optimal bisection of this graph would simply be $S^* = \{(1, j), (2, j) \ : \ j \leq n/2\}$, which has $\delta(S^*) = 2$. Informally, we say that this bisection is one "along the j-coordinate", since S^* and $V - S^*$ are different in terms of their vertices' j values.

However, consider when the Kernighan-Lin algorithm starts with a bisection that makes a partition "along the i-coordinate", such as $S = \{(1, j) \ : \ 1 \leq j \leq n\}$ (which has $\delta(S) = n$). Observe that it is impossible to reach the optimal S^* from this initial cut: swapping any pair of vertices between S and $V - S$ will never *decrease* the initial cut size of $|\delta(S)|$. In order to have a chance at reaching S^*, one would need to allow vertices to change from S to $V - S$ (and vice-versa) without the $V - S$ having to give S one of its vertices in exchange. That is, one would have to allow some imbalance between the size of S and $\delta(S)$. However, correcting this imbalance (to return to a bisection) after a few iterations seems like a difficult task.

Multilevel graph partitioning

As a response to K-L's ghastly behavior on this simple example, we slide quickly to the subject of multilevel graph partitioning, which was first proposed (somewhat independently) by several researchers in the early 1990s. Nowadays the top four graph partitioners (entitled Metis, Jostle, TSL, and Chaco) all use some variant of multilevel graph partitioning.

The Main Idea

Multilevel graph partitioning on a graph G may be thought of in terms of finding a subgraph H that is *representative of* G in a certain sense. Given G, we "sample" or "choose" the vertices of G to get a subgraph H for which finding a partition is easy (or easier). Then we use this partition of H to find a similar, structured partition for G. Some basic pseudocode capturing this concept follows, where we have italicized verbs that are intentionally vague. We will elaborate on some of these verbs in a minute.

Pseudocode

Let $G = (V, E)$.

`MultilevelPart(`G`):`

`If` G `is` *tiny*`, find a partition directly (e.g. try all possible partitions` $(S, V - S)$`) and return the best` $(S, V - S)$ `found.`

Contract G `into a smaller` $G' = (V', E')$`, where` $|G'| \leq \alpha |G|$ `for some constant` $0 < \alpha < 1$. `(This is also called the` *coarsening* `of` G`.)`

`Set` $(S', V' - S')$ `:= MultilevelPart(`G'`).`

Expand G' `back to` G`, and` *project* $(S', V' - S')$ `onto` G`, yielding a partition` $(S, V - S)$ `for` G`.`

Refine $(S, V - S)$ `if necessary and return it.`

As promised, we will now clarify some of the fuzzy words above.

Tiny: Roughly speaking, if the size of the current G' is logarithmic in the size of the original G that was given to this recursive algorithm, then trying all possible separators on this graph can be done in polynomial time. But for most purposes it merely suffices to perform this base case step after the size of G is less than some constant.

Contract/Coarsen: Two straightforward possibilities come to mind for contraction. The first is to randomly sample the vertices of G (something we alluded to in our exposition of the main idea), and delete the sample of vertices and their respective edges from G to get G'. However, this method will only work if the graph is sufficiently dense; otherwise, G will become quickly separated into an H consisting of many pieces, but the simple separator found for H won't be useful in partitioning the original G.

The second proposal for contraction involves maximal matchings. Recall that a maximal matching is a set $M \subseteq E$ where every $e \in E$ shares an endpoint with some $e' \in M$, and no two $m \in M$

share an endpoint. Our contraction begins by finding a maximal matching $M \subseteq E$ of the vertices V in G. Then we associate the two vertices with edge $(u, v) \in M$ with a single "multivertex" $\mu = \{u, v\}$ in V', with $(\mu, w) \in E' \iff (u, w) \in E \vee (v, w) \in E$. (And for $v \in V$ that do not appear in M, we include $v \in V'$ and its edges.) The contracted graph is then $G' = (V', E')$. We might want to include weights on the edges $(\mu_i, \mu_j) \in E'$ in the contracted graph representing how many edges there are in G from a $u \in \mu_i$ to a $v \in \mu_j$.

Finding a maximal matching (as opposed to a *maximum* matching) is quite easy; one can be produced by a greedy algorithm that simply picks independent edges from the graph until no more can be picked (this takes $O(|E|)$ time). Another possibility (if the graph has edge weights) is choose independent edges of cheapest weight, which can be implemented in $O(|E| \log |E|)$ by sorting the edges first.

Expand/Project: Expansion simply means reversing the contraction process. We assume the original graph is already stored in memory, so this is not really a problem. Given the partition (A', B') of G', it is projected onto the original graph by:

$$A = \{v \in V : \exists \mu \in A'.\ \mu \in V'\},\ B = \{v \in V := \exists \mu \in B'.\ \mu \in V'\}.$$

Since no vertices are in two multivertices, (A, B) is indeed a partition of G.

Refine: Returning the projected partition (A, B) "as is" is not generally a good idea, as much information about the original graph is lost in a contraction, so the approximate partition it gives can be fairly inaccurate. Hence a final stage is done where we try to correct any glaring problems with the partition (A, B) by performing a few iterations of the Kernighan-Lin heuristic on it (or some other local refinement technique).

If this refinement step is not done, then after the expansions are done (and the partition is projected), the partition is, roughly speaking, not a "smooth" partition but rather somewhat fractal! That is, as the number of vertices n increases, the partition becomes very "jagged". These problems are only local, and thus can be remedied by the Kernighan-Lin heuristic and other local search methods.

Quick overview of spectral partitioning

Spectral partitioning was once the most popular method for partitioning graphs. However, spectral partitioning is a computationally intensive process; faster methods were eventually favored over it.[1] The main idea is to view a graph in terms of a matrix, in such a way that a partition of the graph can be viewed as finding eigenvectors of the matrix. Recall that an λ-eigenvector x of matrix A is just a vector where $Ax = \lambda x$.

The particular matrix we will use for representing a graph is called the *graph Laplacian matrix*, which we will denote by $\lambda(G)$. We assume the vertices of G are indexed, i.e. $V = \{v_1, \ldots, v_n\}$. Then we define the Laplacian as

$$\lambda(G)[i, j] = \begin{cases} deg(v_i) & \text{if } i = j,\ (deg(v_i) = \text{degree of } v_i) \\ -1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

[1] As a nod to CMU theory, we remark that (Gary) Miller et al. [2] showed that spectral partitioning actually fails for certain kinds of graphs, while his student Shang-Hua Teng (with D. Spielman) showed that the method worked for planar graphs and finite-element meshes [3].

Note when G is undirected, $\lambda(G)$ is symmetric, and therefore its eigenvalues are real. Also notice that if we add the entries in any row (or the entries in any column) the sum is zero.

$\lambda(G)$ has a trivial eigenvalue of 0, with the (equally trivial) 0-eigenvector. The eigenvector corresponding to the second smallest eigenvalue is called the *Fiedler vector*. Spectral partitioning essentially determines this eigenvector e, and finds the median m of the entries in e. Let w be m's corresponding vertex in G (i.e. if m is the ith component of e, then $w = v_i$). Then the following partition of G is returned:

$$(A, B) := (\{v_j : e[j] \leq m\}, \{v_j : e[j] > m\}).$$

One might wonder why on earth this would ever give a good partition, what intuition we should have for believing it works. First, it doesn't work all the time, so forming a very strong intuition for it would be misguided. But informally, there is a (very) rough analogy between spectral partitioning with the Fiedler vector and a taut string that is being vibrated. The second largest eigenvector corresponds to the second mode of vibration of a taut string on the x axis, where precisely half of the string is above the x-axis (one half of the partition), and the other half is below the x-axis. The intersection point of the string with the x-axis would correspond to the median of the Fiedler vector. For more information, see `http://www.cs.berkeley.edu/~demmel/cs267/lecture20/lecture20.html`.

For a large graph, computing the second smallest eigenvector of its (respectively large) Laplacian matrix is too time-expensive in most cases. This is negotiated in practice by combining the multilevel strategy with spectral partitioning: a few contractions of the graph are performed (yielding a G') so that finding the eigenvector on G' is not expensive with respect to the size of the original graph G. The eigenvector solution for G' may be projected on G by constructing a vector e which has $e[i] = k$ iff v_i in G was contained in the multivertex w in G'. This vector e is presumably close to a true eigenvector of $\lambda(G)$, so it may be used as input to the method of repeated squaring (which finds an eigenvector iteratively).

Application of separators: Graph Compression

Often the graphs we run into in practice can be enormous, and merely storing them in memory can be expensive and/or impossible. It is hence very desirable to store them in a most efficient way, even compress them if that can be done. In general it cannot, of course: arbitrary graphs with m edges and n nodes require $\Omega(m \log n)$ bits of storage.

To see this, observe that there are at least $K = \binom{\binom{n}{2}}{m}$ such graphs (there are $\binom{n}{2}$ possible edges, and we choose m of them), so the number of bits required to encode them is at least $\log K \geq \log[n(n-1)/m]^m \in \Omega(m \log n)$.

This lower bound is somewhat deceiving, as it holds for *all* graphs, including large dense classes where we have no interest. So we ask the question: how well can one compress graphs that have small separators? The main result of [1] describes an efficient way to encode graphs with $O(n^{1-\epsilon})$ separators using $O(n)$ bits. Furthermore, one can also perform queries on the graph efficiently (e.g. find the d neighbors of a vertex in $O(d)$ time). Here we will describe the encoding we will use, which will work for any graph, but will require $O(n)$ bits for graphs with $O(n^{1-\epsilon})$ edge-separators. The paper also discusses vertex-separators, but we won't cover those results here.

Gamma codes

Gamma codes are prefix codes that allow one to store arbitrary non-negative integers $n \in \mathcal{N}^+$ using $O(\log n)$ bits. They are prefix codes in the sense that it is possible to read a stream of these codes from left to right (with no separation between different number-encodings), and parse the stream into a list of numbers.

Let $(x)_b$ denote the numeral x written in base b starting at the most-significant nonzero digit. Then the encoding E from $\mathcal{N}^+$ to strings in $\{0,1\}^*$ is the map

$$(n)_{10} \mapsto 0^{\lfloor \log_2 n \rfloor}(n)_2.$$

That is, we "pad" the beginning of the code with about $\log_2 n$ zeroes, then treat the next $\log_2 n$ bits as the number n in binary. For example:

$1 \mapsto 1$, $2 \mapsto 010$, $3 \mapsto 011$, $4 \mapsto 00100$, $5 \mapsto 00101$, $6 \mapsto 00110$, et cetera.

The length of the encoding of an n is clearly $|E(n)| = 2\lfloor \log n \rfloor + 1$. We won't prove this is a prefix code; it is instructive to try decoding a few examples to convince yourself, such as 01000110000111111100100 (which decodes to 2 6 15 1 1 4).

Encoding graphs

How may we apply this code to representing graphs? As before, we assume the vertices are indexed $v_1, \ldots, v_n$, and we associate v_i with i. We use the normal adjacency list representation of graphs, but with a twist. For the vertex v_i, we first encode its degree using a gamma code. Then all of its neighbors v_j are encoded using difference coding: i.e. the difference $j - i$. These differences are stored via a gamma code, with an extra sign bit at the front for each neighbor (in case $j - i < 0$).

Again the method is best shown through an example. Suppose we are encoding the vertices adjacent to v_3 which are v_1, v_4, and v_7. Then the "differences" are -2, 1, and 7, respectively. This is encoded by:

$$011\ 1010\ 01\ 000100,$$

which is $3 \quad -2 \quad +1 \quad +4$.

The entire encoding of G consists of simply concatenating n codes of the above form, one for each vertex v_i, in order (from v_1 to v_n). We will call such an encoding an *difference encoded adjacency table.*

Theorem. For a class of graphs $\mathcal{C}$ satisfying an $O(n^{1-\epsilon})$ edge-separator theorem with $\epsilon > 0$, any n-vertex $G \in \mathcal{C}$ can be represented using an difference encoded adjacency table in $O(n)$ bits.

Proof Outline. We first build an edge-separator tree from the graph. Since the graph satisfies the separator theorem, the separator size for each tree node corresponding to a graph of size n is $O(n^{1-\epsilon})$. As described in a previous class, we can assume the separator tree is perfectly balanced (since if it isn't we can balance it, while only increasing the separator sizes by a constant factor). We assume the separator tree separates all the way to single vertices.

We now number the vertices in an inorder traversal of the tree (e.g. from left to right along the leaves). Note that since we are talking about edge-separators, all vertices are at the leaves. We use these numbers as the vertex identifies that are used in the difference coding.

We now show that this numbering will ensure that the total size of the difference encoded adjacency table is bounded by $O(n)$ bits. Every edge appears in one of the separators. Consider an edge (u, v) in a separator corresponding to a graph of size $O(n)$. The value $|u - v|$ must be less than n since the vertices of the graph are numbered consecutively. The difference code will therefore only use $k(\log n)$ bits. Even if we might want to represent the edge in both directions, it is still $O(\log n)$ bits. This gives us a recurrence for the total number of bits required to encode the differences.

$$S(n) = 2S(n/2) + kn^{1-\epsilon} \log n$$

This solves to $O(n)$ bits when $\epsilon > 0$. We also need to encode the lengths, but this is also bounded by $O(n)$ bits (exercise).

References

[1] Dan Blandford, Guy Blelloch, and Ian Kash. *Compact Representations of Separable Graphs* . To appear in Proceedings of ACM-SIAM Symposium on Discrete Algorithms, 2003.

[2] S. Guattery and G. Miller. *On the performance of the spectral graph partitioning methods.* Proceedings of ACM-SIAM Symposium on Discrete Algorithms, 1995.

[3] Daniel Spielman and Shang-Hua Teng. *Spectral partitioning works: Planar graphs and finite element meshes.* Proceedings of ACM Symposium on Theory of Computing, 1996.