

Geometry, Flows, and Graph-Partitioning Algorithms

By Sanjeev Arora, Satish Rao, and Umesh Vazirani

1. INTRODUCTION

“Graph partitioning” refers to a family of computational problems in which the vertices of a graph have to be partitioned into two (or more) large pieces while minimizing the number of the edges that cross the cut (see Figure 1). The ability to do so is a useful primitive in “divide and conquer” algorithms for a variety of tasks such as laying out very large circuits on silicon chips and distributing computation among processors. Increasingly, it is also used in applications of clustering ranging from computer vision, to data analysis, to learning. These include finding groups of similar objects (customers, products, cells, words, and documents) in large data sets, and image segmentation, which is the first step in image analysis. Unfortunately, most graph-partitioning problems are NP-hard, which implies that we should not expect efficient algorithms that find optimal solutions. Therefore researchers have resorted to *heuristic* approaches, which have been implemented in several popular freeware codes and commercial packages.

The goal of this paper is to survey an interesting combination of techniques that have recently led to progress on this problem. The original motivation for this work was theoretical, to design algorithms with the best provable *approximation guarantees*.^{*} Surprisingly, these ideas have led also to a new framework for designing very fast and

practically viable algorithms for this problem. On the theoretical end, the ideas have also led to a breakthrough in a long-standing open question on metric space embeddings from the field of function analysis, and new algorithms for semidefinite programming. These are all surveyed in this paper.

We will actually describe two disparate-seeming approaches to graph partitioning which turn out, surprisingly, to be related. The first approach is geometric, and holds the key to actually analyzing the quality of the cut found by the algorithm. The second approach involves routing *flows* in the graph, which we will illustrate using *traffic flows* in a road network. This approach holds the key to designing algorithms that run fast. The relationship between these two approaches derives from the fact that they are (roughly) dual views of each other, thus resulting in algorithms which are both fast and also produce high-quality cuts.

Below, we first sketch the two approaches, and give more details in Sections 2 through 4.

1.1. Sketch of the geometric approach

Let us start by describing the geometric approach. We draw the graph in some geometric space (such as the unit disk in two-dimensional Euclidean space \mathbb{R}^2), such that the average length of an edge is short—i.e., the distance between its endpoints is small—while the points are spread out. More precisely, we require that the distance between the average pair of vertices is a fixed constant, say 1, while the distance between the average adjacent pair of vertices is as small as possible. We will refer to this as an *embedding* of the graph in the geometric space.

The motivation behind the embedding is that proximity in the geometric space roughly reflects connectivity in the graph, and a good partition of the graph should correspond to separating a large area by cutting along a geometric curve. Indeed, given the properties of the embedding, even a “random partition” of the space by a simple line or curve should work well! The typical edge is unlikely to be cut by a random partition since it is short while a typical pair of vertices is likely to be separated since the distance between them is large. This means that the expected number of vertices on each side of the cut is large while the expected number of edges crossing the cut is small.[†]

The actual space that our algorithm will “draw” the graph in is not two-dimensional Euclidean space \mathbb{R}^2 , but

* We say that the *approximation guarantee* of the algorithm is C if given any graph in which the best cut has k edges, the algorithm is guaranteed to find a cut which has no more than $C \cdot k$ edges. Sometimes, we also say the algorithm is a C -*approximation*.

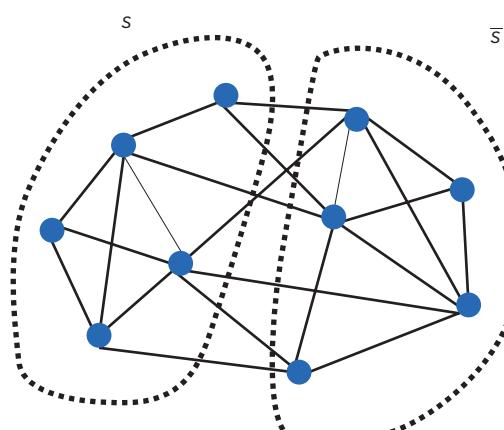


Figure 1: A graph and a partition into two subsets S , \bar{S} . In this case, the two subsets have equal number of vertices; such a partition is called a *bisection*. The number of edges crossing the cut is 7. If the number of vertices on the two sides is within a constant factor of each other (say, factor 2), then we call the partition *balanced*. Balanced partitions are useful in many applications.

the surface of the unit sphere in an n -dimensional Euclidean space, where n is the number of graph vertices. Moreover, the “distance” between points in this space is defined to be the square of the Euclidean distance. This means that we draw the graph so that the sum of squares of the lengths of the edges is as small as possible, while requiring that the square of the distance between the average pair of points is a fixed constant, say 1.

There are some important additional constraints that the geometric embedding must satisfy, which we have suppressed in our simplified outline above. These are described in Section 2, together with details about how a good cut is recovered from the embedding (also see Figure 3 in Section 2). In that Section, we also explain basic facts about the geometry of n -dimensional Euclidean space that are necessary to understand the choice of parameters in the algorithm. The proof of the main geometric theorem, which yields the $O(\sqrt{\log n})$ bound on the approximation factor achieved by the algorithm, makes essential use of a phenomenon called *measure concentration*, a cornerstone of modern convex geometry. We sketch this proof in Section 6.

The main geometric theorem has led to progress on a long-standing open question about how much distortion is necessary to map the metric space l_1 into a Euclidean metric space. This result and its relation to the main theorem are described in Section 7.

1.2. Sketch of the flow-based approach

We now describe the flow-based approach that holds the key to designing fast algorithms. In Section 3, we mention why it is actually dual to the geometric approach. To visualize this approach imagine that one sunny day in the San Francisco Bay area, each person decides to visit a friend. The most congested roads in the resulting traffic nightmare will provide a sparse cut of the city: most likely cutting the bridges that separate the East bay from San Francisco.*

More formally, in the 1988 approach of Leighton and Rao¹⁴ (which gives an $O(\log n)$ -approximation to graph-partitioning problems), the flow routing problem one solves is this: route a unit of flow between every pair of vertices in the graph so that the congestion on the most-congested edge is as small as possible; i.e., route the traffic so that the worst traffic jam is not too bad. A very efficient algorithm for solving this problem can be derived by viewing the problem as a contest between two players, which we can specify by two (dueling) subroutines. Imagine that a traffic planner manages congestion by assigning high tolls to congested edges (streets), while the flow player finds the cheapest route along which to ship each unit of flow

[†] It may appear strange to pick a random partition of this geometric space instead of optimizing this choice. Though some optimization is a good idea in practice, one can come up with worst-case graphs where this fails to provide substantial improvements over random partitions. A similar statement holds for other algorithms in this paper that use random choices.

* Of course, unlike San Francisco’s road system, a general graph cannot be drawn in the Euclidean plane without having lots of edge crossings. So, a more appropriate way of picturing flows in a general graph is to think of it as a communications network in which certain vertex pairs (thought of as edges of the “traffic graph”) are exchanging a steady stream of packets.

(thereby avoiding congested streets). In successive rounds, each player adjusts the tolls and routes respectively to best respond to the opponent’s choices in the previous round. Our goal is to achieve an equilibrium solution where the players’ choices are best responses to each other. Such an equilibrium corresponds to a solution that minimizes the maximum congestion for the flow player. The fact that an equilibrium exists is a consequence of linear programming duality, and this kind of two-player setting was the form in which von Neumann originally formulated this theory which lies at the foundation of operations research, economics, and game theory.

Indeed, there is a simple strategy for the toll players so that their solutions quickly converge to a nearly optimal solution: assign tolls to edges that are exponential in the congestion on that edge. The procedure in Figure 3 is guaranteed to converge to solution where the maximum congestion is at most $(1 + \varepsilon)$ times optimal, provided $\mu \geq 1$ is a sufficiently close to 1. The number of iterations (i.e., flow reroutings) can also be shown to be proportional to the flow crossing the congested cut.

But how do we convert the solution to the flow routing problem into a sparse cut in the graph? The procedure is very simple! Define the distance between a pair of vertices by the minimum toll, over all paths, that must be paid to travel between them. Now consider all the vertices within distance R of an arbitrary node v . This defines a cut in the graph. It was shown by Leighton and Rao,¹⁴ that if the distance R is chosen at random, then with high probability the cut is guaranteed to be within $O(\log n)$ times optimal. The entire algorithm is illustrated in the figure below.

Here is another way to think about this procedure: we may think of the tolls as defining a kind of abstract “geometry” on the graph; a node is close to nodes connected by low-toll edges, and far from nodes connected by large toll edges. A good cut is found by randomly cutting this abstract space. This connection between flows and geometry will become even stronger in Section 3.

In our 2004 paper,⁶ we modified the above approach by focusing upon the choice of the traffic pattern. Instead of routing a unit of flow between every pair of vertices—i.e., a traffic pattern that corresponds to a complete graph—one can obtain much better cuts by carefully choosing the traffic

Figure 2: Parameter μ depends upon ε , which specifies the accuracy with which we wish to approximate the congestion.

<p>ROUTE PATHS AND CUT Input: $G = (V, E)$ Maintain: $d(\cdot)$ on the edges of G. Initially $d(e) = 1$.</p> <ol style="list-style-type: none"> 1. Do until the maximum $d(e)$ is n. <ol style="list-style-type: none"> (a) Choose a random (i, j) pair. (b) Find a shortest path, p, from i to j. (c) Multiply $d(e)$ on each edge of p by μ. 2. Choose a value δ randomly and an arbitrary node u and return the set of nodes within distance δ of u.

pattern. We showed that for every graph there is a traffic pattern that reveals a cut that is guaranteed to be within $O(\sqrt{\log n})$ times optimal. This is proved through a geometric argument and is outlined in Section 3. An efficient algorithm to actually find such a traffic pattern, the routing of the flow, and the resulting cut was discovered by Arora, Hazan, and Kale.² The resulting $\tilde{O}(n^2)$ implementation of an $O(\sqrt{\log n})$ approximation algorithm for sparse cuts is described in Section 4. Thus, one gets a better approximation factor relative to the Leighton–Rao approach without a running time penalty. Surprisingly, even faster algorithms have been discovered.^{4,12,16} The running time of these algorithms is dominated by very few calls to a single commodity max-flow procedure which are significantly faster in practice than the multicommodity flows used in Section 4. These algorithms run in something like $O(n^{1.5})$ time, which is the best running time for graph partitioning among algorithms with proven approximation guarantees. We describe these algorithms and compare them to heuristics such as METIS in Section 5.

2. THE GEOMETRIC APPROACH AND THE ARV ALGORITHM

In this Section, we describe in more detail the geometric approach for graph partitioning from our paper.⁶ Henceforth referred to as the ARV algorithm. Before doing so, let us try to gain some more intuition about the geometric approach to graph partitioning. We will then realize that the well-known spectral approach to graph partitioning fits quite naturally in this setting.

2.1. The geometric approach

In Section 1, we introduced the geometric approach by saying “We draw the graph in some geometric space (such as the two-dimensional Euclidean space \mathbb{R}^2). . . .” Well, let us consider what happens if we draw the graph in an even simpler space, the real line (i.e., \mathbb{R}). This calls for mapping the vertices to points on the real line so that the sum of the (Euclidean) distances between endpoints of edges is as small as possible, while maintaining an average unit distance between random pairs of points. If we could find such a mapping, then cutting the line at a random point would give an excellent partition. Unfortunately, finding such a mapping into the line is NP-hard and hence unlikely to have efficient algorithms.

The popular *spectral method* does the next best thing. Instead of Euclidean distance, it works with the square of the Euclidean distance—mapping the vertices to points on the real line so the sum of the squares of edge lengths is minimized, while maintaining average unit squared distance between a random pair of points. As before, we can partition the graph by cutting the line at a random point. Under the squared distance, the connection between the mapping and the quality of the resulting cut is not as straightforward, and indeed, this was understood in a sequence of papers starting with work in differential geometry in the 1970s, and continuing through more refined bounds through the 1980s and 1990s.^{1,8,18} The actual algorithm for mapping the points to the line is simple enough

that it might be worth describing here: start by assigning each vertex i a random point x_i in the unit interval. Each point x_i in parallel tries to reduce the sum of the squares of its distance to points corresponding to its adjacent vertices in the graph, by moving to their center of mass. Rescale all the x_i 's so that the average squared distance between random pairs is 1, and repeat. (To understand the process intuitively, think of the edges as springs, so the squared length is proportional to the energy. Now the algorithm is just relaxing the system into a low-energy state.) We note that this is called the *spectral method* (as in *eigenvalue spectrum*) because it really computes the second largest eigenvector of the adjacency matrix of the graph.

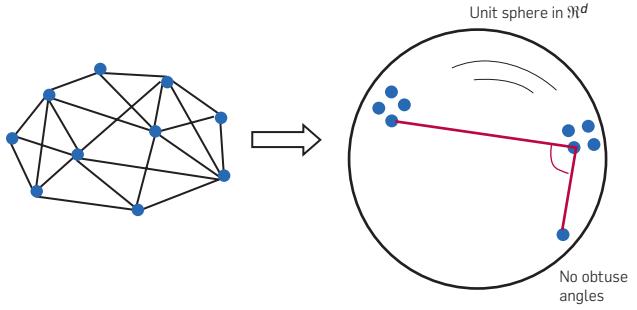
2.2. The ARV algorithm

The algorithm described in Section 1.1 may be viewed as a high-dimensional analog of the spectral approach outlined above. Instead of mapping vertices to the line, we map them to points on the surface of the unit sphere in n -dimensional Euclidean space. As in the spectral method, we work with the square of the distance between points and minimize the sum of the squares of the edge lengths while requiring that the square distance between the average pair of points is a fixed constant, say 1. The sum of squares of lengths of all edges is called the *value* of the embedding.

How closely does such an embedding correspond to a cut? The ideal embedding would map each graph vertex to one of two antipodal points on the sphere, thus corresponding to a cut in a natural way. In fact, the value of such an embedding is proportional to the number of edges crossing the cut. It follows that the minimum-value two-point embedding corresponds to an optimal partitioning of the graph.

Unfortunately, the actual optimal embedding is unlikely to look anything like a two-point embedding. This is because squaring is a convex function, and typically we should expect to minimize the sum of the squared lengths of the edges by just sprinkling the vertices more or less uniformly on the sphere. To reduce this kind of sprinkling, we require the embedding to satisfy an additional constraint (that we alluded to in Section 1.1): the angle subtended by any two points on the third is not obtuse. The equivalent Pythagorean way of saying this is that the squared distances must obey the *triangle inequality*: for any triple of points u ,

Figure 3: A graph and its embedding on the d -dimensional Euclidean sphere. The triangle inequality condition requires that every two points subtend a non-obtuse angle on every other point.



$v, w, |u - v|^2 + |v - w|^2 \geq |u - w|^2$. The two-point embedding remains viable since it satisfies this constraint. By contrast, the spectral method described above fails to meet this condition, since any three distinct points on a line form an obtuse triangle. (See the accompanying box for a discussion of the hypercube, which is a graph that provides an intuitive picture about the nature of this constraint.)

The conditions satisfied by the embedding are formally described in Figure 4. An embedding satisfying these conditions in a sufficiently high-dimensional space can be computed in polynomial time by a technique called SDP. A semidefinite program is simply a linear program whose variables are allowed to be certain quadratic functions. In our case, the variables are squared distances between the points in \mathbb{R}^d . Indeed, researchers have known about this approach to graph partitioning in principle for many years (more precisely since^{10,15}), but it was not known how to analyze the quality of cuts obtained. For a survey of uses of SDP in computing approximate solutions to NP-hard problems, see.⁹

The specific conditions in Figure 4 were chosen to express the semidefinite relaxation of the graph *min-bisection* problem, where the two sides of the cut have to be of equal size. In this case, a feasible solution is to map each partition of an optimal bisection to two antipodal points. Therefore, the optimal solution to the SDP provides a lower bound on the cost of the optimal bisection. How do we recover a good bisection from this embedding? This is a little more involved than the simple “random partition” technique described in the previous subsection.

The key to finding a good cut is a result from⁶ that any embedding satisfying all the above conditions is similar to a two-point embedding in the following sense: given such an embedding $\{v_1, \dots, v_n\}$, we can efficiently find two disjoint “almost antipodal” sets, S and T , each with $\Omega(n)$ points that are at least $\Delta = \Omega(1/\sqrt{\log n})$ apart. That is, every point in S has distance at least Δ from every node in T . Once the sets S and T are identified, there is a very simple procedure

Figure 4: Criteria for geometric embedding.

Let the i th vertex get mapped to the point v_i . The mapping must satisfy the following conditions:

- The points lie on the unit sphere:

$$\forall i \quad |v_i|^2 = 1$$

- The points are well spread:

$$\sum_{i < j} |v_i - v_j|^2 \geq 2 \binom{n}{2}$$

- Satisfy the triangle inequality:

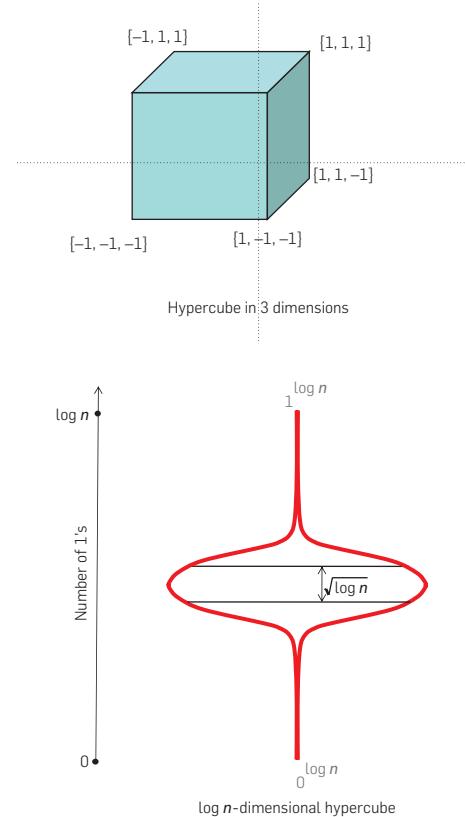
$$\forall i, j, k \quad |v_i - v_j|^2 + |v_j - v_k|^2 \geq |v_i - v_k|^2$$

- Edges are short:

$$\min \sum_{(ij) \in E} |v_i - v_j|^2$$

EXAMPLE: THE HYPERCUBE

Insisting that the embedding of the graph into \mathbb{R}^d satisfies the triangle inequality is quite a severe constraint. Indeed, on the surface of a unit sphere in d dimensions, the maximum number of distinct points that can satisfy this condition is 2^d . An extreme example is the d -dimensional hypercube, which has exactly $n = 2^d$ points. The points are identified with vectors in $[-1, +1]^d$, and these vectors obey the triangle inequality since they do so coordinate wise.

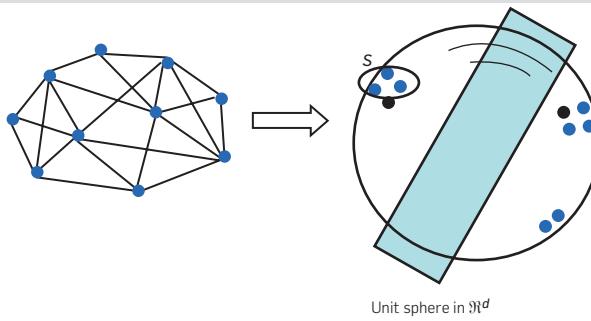


Each point of the d -dimensional hypercube is connected by an edge to exactly d other points—those that differ from it in exactly one coordinate. The optimal bisection cut corresponds to any one of the d dimensions: separate the points according to whether that particular coordinate is -1 or $+1$. The number of edges crossing this cut is exactly $n/2 = 2^{d-1}$, one for each 1-pair.

Another natural way of bisecting the d -dimensional hypercube is by a level cut. Imagine that we arrange the points on levels according to the sum of the coordinates. There are exactly $d+1$ levels where the j th level has exactly $\binom{d}{j}$ points. The bisection cut separating the lowest $d/2$ levels from the remaining $d/2$ levels has $\Theta(\sqrt{d}2^d)$ edges (which is $\Theta(\sqrt{d}) = \Theta(\sqrt{\log n})$ factor worse than optimal bisection cut). This is because the middle level has roughly $2^d/\sqrt{d}$ points, and each point has $d/2$ edges that cross the cut. The interesting thing is to note that there was nothing special about the sum of coordinates, and indeed we could start with any point v at the lowest level, and then assign the rest of the points of the hypercube to levels according to their distance from v . The number of distinct level cuts is 2^{d-1} (choosing v or its “antipodal” point leads to the same level sets).

Our algorithm uses a random projection to define a cut. When run on the hypercube, this corresponds to finding a level cut. Thus, the algorithm fails to discover anything close to one of the d optimal dimension cuts. In this case, the algorithm’s answer is indeed $\sqrt{\log n}$ factor away from optimal.

Figure 5: Procedure to produce a cut from the embedding of the graph on the sphere. The points in the thin slice in the middle are thrown out.



for finding a balanced partitioning of the vertices with few crossing edges.[†] This is done as follows: simply pick a random distance $0 \leq r \leq \Delta$, and select every vertex i such that v_i is within r of some point in S , i.e. for some $x \in S$, $|v_i - x|^2 \leq r$. To see that this works, note that the probability that any edge e with length $l(e)$ is in the cut is at most $l(e)/\Delta$. Thus, the typical number of edges crossing such a cut is at most $1/\Delta$ times the total length of the edges (which, recall, is at most the size of the optimal cut). This yields an $O(\sqrt{\log n})$ -approximation algorithm since $\Delta = \Omega(1/\sqrt{\log n})$.

We now turn our attention to actually finding the almost antipodal sets S and T . Before we do so, it is instructive to understand an important geometric property of a high-dimensional Euclidean space \mathbb{R}^d . Consider projecting the surface of the unit sphere on to a line u through its center. Clearly, the points on the sphere will project to the interval $[-1, 1]$. If we start from a uniformly random point v on the sphere, what is the distribution of the projected point (v, u) in $[-1, 1]$? It turns out that the projected point has a Gaussian distribution, with expectation 0 and standard deviation $1/\sqrt{d}$. This means that the expected squared distance of a projected point from the center is $1/d$, and a constant fraction of the surface of the sphere is projected at least $1/\sqrt{d}$ away from the origin. Another way to say all this is that if we consider slices of the sphere by parallel hyperplanes the surface area of the slices vary like a Gaussian according to distance of the slice from the center of the sphere.

Now returning to our procedure for identifying sets S and T from the embedding of the graph, we start by projecting the n points v_1, \dots, v_n on a randomly chosen line \vec{u} through the center. We discard all points whose projection has absolute value less than $1/\sqrt{d}$. The remaining points form two sets P and N , according to whether their projection is positive or negative. By the Gaussian property above, the sets P and N have expected size $\Omega(n)$. Ideally, we would like to stop here and assert that every point in $x \in P$ is far from every point in $y \in N$.

Indeed, it is true (and easily checked) that with high probability over the choice of the line on which we are

[†] Note that we started off trying to find a partition into sets of *equal* size, but this method will only yield a partition in which the two sets have sizes within a factor 2 of each other.

doing the projection, $|x - y|^2 \geq 1/\log n$. However, we actually want a greater separation of $\Delta = 1/\sqrt{\log n}$, so we enforce this condition by sequentially deleting pairs $\{x, y\}$ that violate it. The remaining points define two sets $S \subseteq P$ and $T \subseteq N$ which are Δ -separated. The difficult part is to show that S and T have cardinality $\Omega(n)$. For the interested reader, we have sketched the main ingredients of the proof of this fact in Section 6. Figure 6 below summarizes the resulting algorithm for finding a good cut.

Figure 6: Finding a good cut.

Given the embedding v_1, \dots, v_n ,

- Pick a random line \vec{u} through the origin.
- Let $P = \{v_i : (v_i, u) \geq 1/\sqrt{d}\}$
and $N = \{v_i : (v_i, u) \leq 1/\sqrt{d}\}$
- Discard pairs of points from $x \in P$ and $y \in N$ such that $|x - y|^2 \leq \Delta = 1/\sqrt{\log n}$.
Call the resulting sets S and T .
- Choose random $0 \leq r \leq \Delta$, and let $X = \{i : |v_i - x|^2 \leq r\}$ for some $x \in S$.
- Output partition $(X, V - X)$.

3. EXPANDER FLOWS

In Section 1.2, we described a dual flow-based approach to partitioning a graph. In this section, we describe how to extend that to the expander flow framework, which leads to an efficient implementation of the $O(\sqrt{\log n})$ approximation algorithm. Before we can describe this new framework, we must introduce a number of new concepts. Let us start by recalling the San Francisco Bay Area traffic nightmare scenario of Section 1.2, where we chose to route a traffic pattern described by a complete graph. We described an efficient algorithm for computing good routes for the traffic to minimize the worst traffic jams. We also showed how to use the resulting information to actually find a good partition for the county's road network.

In this section, we will focus on the choice of traffic pattern. We will see that choosing the traffic pattern in a clever way will reveal a much better partition of the road network. To begin with, notice that if the traffic pattern is very local—for example, if each person visits a friend who lives a few blocks away—then the precise location of the congested streets has little information about the actual bottlenecks or sparse cuts in the underlying road network. So we start by formally understanding what kinds of traffic graphs will reveal the kind of information we seek. To be more concrete, by the traffic graph, we mean the weighted graph in which nodes i and j have an edge of weight c_{ij} if they have a flow rate c_{ij} between them.

We require the traffic graph to be “expanding,” that is, the total traffic flow out of every subset of nodes is proportional to the number of nodes in the subset. In more

mathematical language, we say that a cut $(S, V - S)$ is β -expanding, if the edges crossing this cut have total weight at least β times $\min\{|S|, |V - S|\}$. A graph $H(V, F)$ is said to be a β -expander, for some constant β , if for every subset $S \subseteq V$, the cut $(S, V - S)$ is β -expanding. Expander graphs have no small graph separators. Here, we will be interested in traffic graphs that are β -expanders for some constant β , and where the maximum degree of a node is at most d for some constant d . (The degree of a vertex is the sum of weights of all edges incident to it. We assume that the degree of each vertex is at least 1.) An important property of this class of graphs is that there is an efficient test to distinguish constant degree expander graphs from graphs with small separators. Indeed, this test is based on the spectral method, which works very well for constant degree expanders.

We said earlier that a clever choice of traffic graph will reveal a better way to partition the road network graph. How do we make this clever choice? The idea behind the new algorithm is to search for the best “expanding” traffic graph: which means among all expanding traffic graphs pick one that leads to the smallest traffic jams. This might seem counterintuitive. Shouldn’t locating the worst cut correspond to finding a traffic pattern that leads to the worst traffic jams?

To understand this point more clearly, let us introduce some notation. Let $G(V, E)$ be the graph that we wish to partition, and $H(V, F)$ be an expander graph on the same vertex set that we use as the traffic graph to route a traffic flow in G . Recall that this means that one unit of flow must be shipped between each pair of nodes connected by an edge in F . (In fact, the algorithm must allow fractional edges in F , which our description will ignore.) Suppose α is such that the sparsest cut in $G(V, E)$ separates the set of vertices $S \subseteq V$ from $V - S$, by cutting only $\alpha|S|$ edges. Thus α is a measure of how sparse the optimum cut is. Now, regardless of how the flow corresponding to $H(V, F)$ is routed in $G(V, E)$, at least $|S|$ units of flow (traffic) must be routed between S and $V - S$. Since this must be routed through a bottleneck containing only $\alpha|S|$ edges, it follows that the worst edge-congestion must be at least $1/\alpha$. The real issue in designing an approximation algorithm for sparsest cuts is this: how much larger can the edge-congestion be than this lower bound of $1/\alpha$? If it is no larger than C/α , then it will follow that we can approximate the sparsest cut (at least its numerical value) to within a factor of C in the worst case. To obtain the best results, we must try to minimize C . In other words, we wish to pick an expanding traffic graph which leads to the smallest traffic jams.

The main result in ARV⁶ on expander flows states that for any graph $G(V, E)$ there is an expander graph $H(V, F)$ that can be routed in $G(V, E)$ with congestion at most $O(\sqrt{\log n}/\alpha)$. Moreover, using the powerful computational hammer of the ellipsoid algorithm for linear programming, the expander graph $H(V, F)$ as well as its routing in $G(V, E)$ minimizing edge-congestion can be computed in polynomial time. This gives an algorithm for sparsest cuts, different from the geometric one presented earlier, yet achieves the same $O(\sqrt{\log n})$ approximation factor in the worst case. It also serves as a starting point for a new framework for

designing very efficient algorithms for finding sparse cuts.

In the rest of this section, we will sketch the proof of the main expander flows result of ARV.⁶ To do so, we shall generalize the router/toll-collector game from Section 1.2 to incorporate the selection of an expanding traffic graph. The existence of an expanding traffic graph that achieves the desired $O(\sqrt{\log n})$ bound will follow from understanding the equilibrium of the game. Since we are interested in the equilibrium solution, rather than detailing an efficient procedure for getting to equilibrium, it is easier to formulate the game as lasting for a single round.

In the generalized game, the toll-collector not only specifies a toll for each edge (the sum of all edge tolls is n , the number of vertices in G), but also a set of prizes. Each prize is associated with a cut in the graph, and is collected when a path crosses the cut. The number of possible cuts is 2^n of course, and the toll collector selects some subset of cuts to place prizes on, such that the sum of all cut prizes is 1. We associate with each pair of vertices the total prize for moving from one vertex to another—the sum of cut prizes for all cuts that separate the two vertices. The task of the routing player is to select a pair of vertices, separated by total cut prize of at least β (where β is a given constant), such that the total toll paid in moving between these vertices is as small as possible. The goal of the toll player is to assign edge tolls and cut prizes to maximize the toll paid by the routing player. It can be shown using linear programming duality that this payoff is essentially the congestion of the best expander flow. The main idea is that the flow player’s optimal strategy is specified by a probability distribution over pairs of vertices, which we may view as defining the desired expander graph (for any balanced cut, a random edge from this graph must cross the cut with probability at least β).

Let us consider how the toll player can maximize his take. Assume that $G(V, E)$ has a balanced separator with αn edges crossing the cut. Then by assigning a prize of 1 to this cut and a toll of $1/\alpha$ on each edge of this cut, the toll player can force the routing player to pay at least $1/\alpha$, simply because the routing player is forced to route flow between two vertices on opposite sides of this optimal cut. Can the toll player force the route player to pay a lot more? We show that he cannot force him to pay more than $O(\sqrt{\log n}/\alpha)$: no matter how the toll player determines tolls and picks a set of cuts, the routing player can always pick a pair of vertices which are separated by a β fraction of the cuts, and such that the cheapest path between them costs $O(\sqrt{\log n}/\alpha)$. The proof of this fact relies upon the geometric view from the previous section.

To make this connection, we start by viewing the set of cuts as defining a hypercube—each cut corresponds to a dimension, and each vertex gets a ± 1 label depending upon which side of the cut it lies. It is natural to associate each vertex in the graph with a vertex of the d -dimensional hypercube, where d is the number of cuts. We observed in the previous section that the d -dimensional hypercube can be embedded on the surface of a unit sphere in \mathbb{R}^d and satisfy the triangle inequality. By the main geometric theorem of ARV, it follows that there are two large almost antipodal

sets S, T , each with $O(n)$ vertices, such that every pair of vertices $x \in S$ and $y \in T$ is at least $1/\sqrt{\log n}$ apart. In our example, this means that there are large sets of vertices S, T such that every pair of vertices $x \in S$ and $y \in T$ lies on opposite sides of at least $1/\sqrt{\log n}$ fraction of the cuts. By a simple averaging argument, since every cut separating S from T has at least αn edges, there must be a pair of vertices x and y which are connected by a path of cost at most $1/\alpha$.

We are not quite done yet, since we must actually exhibit a pair of vertices that lie on opposite sides of β fraction of the cuts. So far, we have only exhibited a pair of vertices that lie on opposite sides of $1/\sqrt{\log n}$ fraction of the cuts. The last step in the proof is to piece together a sequence of $O(\sqrt{\log n})$ pairs to obtain a pair of vertices that lie on opposite sides of β fraction of the cuts, and which are connected by a path of cost $O(\sqrt{\log n}/\alpha)$. To carry out this last step, we actually have to appeal to the internal structure of the proof establishing the existence of the antipodal sets S and T . This “chaining” argument has found other uses in other research in the last few years.

3.1. The duality connection

There is a notion of duality for SDP and an expander flow turns out to be a type of dual solution to the semidefinite program outlined in Figure 1. Thus as in all settings involving duality, a dual solution can be seen as “certifying” a lower bound on the optimal value of the primal. Thus, the expander flow is a way to certify a lower bound on the size of the optimal cut.

4. AHK ALGORITHM FOR FINDING EXPANDER FLOWS

Arora, Hazan, and Kale (AHK)² managed to turn the 1-round game described in the previous section into an efficient algorithm, specifically, by designing efficient strategies for the toll player and the routing player.

During the AHK algorithm, the routing player maintains a traffic graph and the toll player maintains edge tolls and cut prizes as above. At each round, the toll player does spectral partitioning on the current traffic graph, finds a sparse cut in it, and places some nonzero prize on it. (This requires adjusting down the prizes on existing cuts because the net sum has to stay 1.) The routing player then finds pairs of nodes as mentioned in Section 3—namely, a pair whose prize money is at least β and whose edge-toll-distance is smallest—and adds the resulting paths to the traffic graph (again, this requires adjusting the values of the existing edges in the traffic graph because the total degree of each node in the traffic graph is constant). The game finishes when the traffic graph is a β -expander, a condition that can be checked by the spectral methods mentioned earlier.

With some work, each round can be implemented to run in $O(n^2)$ time. The key to the performance of the algorithm lies in the fact that the readjustment of tolls can be done in a way so that the game finishes in $O(\log n)$ rounds, which results in a running time of $O(n^2 \log n)$. This uses a feasible version of von Neumann’s minmax theorem that Arora, Hazan, and Kale (in a separate survey paper³) call the *multiplicative weight update rule*:

the toll player updates tolls on the edges and cuts accordingly by always multiplying or dividing by a fixed power of the quantity $(1 + \varepsilon)$ where ε is small. (This updated rule has been around for five decades and been rediscovered in many settings including convex optimization and machine learning; see³ for a survey.) They set up the game with some care so that the toll on an edge at any time is an exponential function of the flow routed through it, which is where the logarithmic bound on the number of rounds comes from.

4.1. Faster algorithms for semidefinite programs

Motivated by the above discoveries, Arora and Kale⁴ have designed a new combinatorial approach for computing approximate solutions to SDPs. This is important because solving SDPs usually is quite slow in practice (though polynomial time in theory). Their idea is to define a multiplicative weight update rule for matrices, which is used in a primal-dual fashion to get a quick approximation. The algorithm has some formal similarities to the “random walk” idea described in Section 5. They obtain the best running times known for a host of SDP-based approximation algorithms. For finding cuts, one may prefer the algorithms of the next section on account of simplicity, though the Arora–Kale approach is comparable in running time and approximation factor. Again, we refer the reader to the survey.³

5. TOWARD PRACTICAL ALGORITHMS

So far, we have described a framework for improving the approximation factor or the quality of cut found by graph-partitioning algorithms. In this section, we describe an algorithm of Khandekar, Rao, and Vazirani¹² that modifies the framework to design a much faster algorithm for graph partitioning—its running time is bounded by a small number of calls to a single commodity max-flow subroutine. As in the AHK algorithm from the previous section, the new algorithm may also be viewed as a contest between two players, though the underlying game, called the cut-matching game, is slightly different.

The object of the cut-matching game, played over a sequence of rounds, is to construct an expanding traffic graph. Initially, the traffic graph is the empty graph on n vertices. In each round, the cut player chooses a bisection of the vertex set, and the matching player specifies a perfect matching that pairs up vertices across the bisection. The edges of the perfect matching are added to the traffic graph, and the game continues until the traffic graph has expansion greater than 1. The goal of the cut player is to minimize the number of rounds before the game ends. The matching player, of course, tries to prolong the game.

How does the cut-matching game relate to partitioning a given graph $G(V, E)$? Given a bisection, the matching player tries to find a perfect matching that can be routed through $G(V, E)$ with small congestion. This is accomplished by performing a single max-flow computation, and the corresponding min-cut is a candidate partition of $G(V, E)$ (see Figure 5).

The cut player uses a novel spectral type algorithm,

described in the box below, to find a bisection in almost linear time. It can be shown that by following this strategy the cut player can limit the number of rounds of the game to $O(\log^2 n)$. Now the best candidate partitioning among the $O(\log^2 n)$ rounds is the output of the algorithm, and can be shown to be within a $O(\log^2 n)$ factor of the optimal partition.

PROCEDURE: MATCHING PLAYER

Input: Bisection (S and $V - S$), of graph G .

- Run a concurrent flow computation between S and $V - S$.
Formulate a flow problem where each node in S is a source and each node in $V - S$ is the sink of one unit of flow.
The goal is to satisfy each source and sink while minimizing the maximum integral flow across any edge of G . This is a standard single commodity flow problem.
- Decompose the flow into $n/2$ source-sink paths to obtain the matching:
Greedily follow a path of positive flow arcs from a source node u to some sink. Remove this path from the flow, and repeat.
- Output the matching.

PROCEDURE: CUT PLAYER

Input: Set $\{M_1, \dots, M_T\}$ of perfect matchings.

1. Let v^0 be a random n -dimensional unit vector.
2. For $t = 1$ to T
For each edge $\{i, j\} \in M_t$
 $-v^{t+1}(i) = (v^t(i) + v^t(j))/2$
3. Output cut $(S, V - S)$, where
 $S = \{i : v_T(i) \leq m\}$,
with m being the median value of the $v_T(i)$'s.

The total running time of the matching player is bounded by $O(\log^2 n)$ invocations of single commodity max-flow computations. This is the dominant term in the running time of the algorithm, since the cut player runs in nearly linear time.

The overall algorithm starts by invoking the cut player with the empty set of matchings and invokes the cut player and matching player for $O(\log^2 n)$ rounds. Each invocation of the matching player results in a candidate partition of the graph given by the min-cut in the invocation of the max-flow procedure. The algorithm outputs the best candidate partition over all the rounds. It was shown in¹² that the approximation factor achieved by this partition is $O(\log^2 n)$. As mentioned in Section 4, a different primal-dual algorithm (with some formal similarity to the above algorithm) was used⁴ to improve the algorithm to yield an approximation factor of $O(\log n)$. This result was recently matched in¹⁶ by an algorithm that stays within the framework of the cut-matching game. They also showed that the best

approximation factor that can be obtained within the cut-matching framework is $\Omega(\sqrt{\log n})$. Designing a $O(\sqrt{\log n})$ approximation algorithm in the cut-matching framework remains an intriguing open question.

It is instructive to compare the flow-based algorithm described here to a clustering-based heuristic embodied in the widely used program METIS.¹¹ That heuristic proceeds by collapsing random edges until the resulting graph is quite small. It finds a good partition in this collapsed graph, and successively induces it up to the original graph, using local search. The flow-based algorithm may be viewed as a continuous version of this heuristic since each matching may be thought of as “virtually collapsing” each matched pair. In particular, early iterations of the flow-based algorithm will select matchings that mostly consist of random edges. Moreover, the cuts provided by the cut player will rarely partition the resulting connected components. Thus, it will proceed very much like METIS. Indeed, this algorithm might have an advantage over METIS since the “virtual collapsing” of edges in the actual sparse cut need not preclude finding this cut.

We point the reader interested in empirical results to Chris Walshaw’s graph-partitioning benchmark (<http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/>), which provides results of various heuristics on an interesting set of benchmarks. The impressive results for METIS can be compared to SDP (by Lang and Rao), which uses a simplified version of SDP with a max-flow cleanup. Though this resembles the max-flow-based algorithm described in this section, to our knowledge, the specific algorithms discussed in this paper have yet to be rigorously compared against METIS.

6. CORRECTNESS PROOF FOR THE GEOMETRIC APPROACH

In this section, we sketch a proof of the main geometric theorem of,⁶ for any well-separated set of points on the surface of the unit sphere in \mathbb{R}^d that satisfy the triangle inequality, there are two linear-sized almost antipodal sets, S, T , that are $\Delta = 1/\sqrt{\log n}$ separated. The sketch here also incorporates a simplification due to Lee.¹³

Recall that our procedure for identifying these two sets S, T was to project the points on a random line \vec{u} through the origin, discard all points whose projection has absolute value less than $1/\sqrt{d}$, and label the remaining points as being in P and N , according to whether their projection is positive or negative. We then discarded pairs of points from the two sets that were less than Δ separated, to finally obtain sets S and T .

For the theorem to fail, for most choices of directions \vec{u} , most points must be paired and discarded. Recall that for a pair to be discarded, the points must be Δ -close to each other while their projection on \vec{u} must be at least $\epsilon = 2\frac{\sigma}{\sqrt{d}}$ apart. These discarded pairs form a matching for that direction. Let us make a simplifying assumption that all n points are paired and discarded in each direction. (In the actual proof, we use an averaging argument to show that there are $\Omega(n)$ points such that in most directions a constant fraction of them is matched.)

The overall plan is, given a random direction \vec{u} , to piece

together a sequence of matched edges, each with a large projection onto \vec{u} , into a path., i.e., we find a sequence of points x_1, \dots, x_l with the property that on average $(x_i - x_{i+1}, u) \geq \epsilon/2$ and therefore $(x_1 - x_l, u) \geq l \epsilon/2$. On the other hand, since $|x_i - x_{i+1}|^2 \leq \Delta$, triangle inequality implies that $|x_1 - x_l| \leq l\Delta$, and therefore $|x_1 - x_l| \leq \sqrt{l\Delta}$. Note that ϵ was chosen to be (roughly) the standard deviation of the projection length of a unit vector. Therefore, the projection length of $x_1 - x_l$ is $t = \sqrt{l/\Delta}$ many standard deviations from its mean. The probability that the projection is t standard deviations from its mean is at most $e^{-t^2/2}$. So if the number of hops l in the path is $\Omega(\sqrt{\log n})$, then the probability of this event is polynomially small. We get a contradiction by applying the union bound to the n^2 pairs of points.

The main challenge then is to piece together such a path. To find a path for a direction u , we clearly cannot limit ourselves to the matching in that direction. To understand how matchings for different directions relate to each other, we need two new ideas. Let us first introduce some notation.

Recall that each point v has a matched pair for half the directions (i.e., either for \vec{u} or $-\vec{u}$). We say that v is ϵ -covered in these directions. And we say that v is (ϵ, δ) -covered if, v is ϵ -covered for atleast δ measure of directions.

The first idea is simple: if v is ϵ covered in direction \vec{u} , then it is almost ϵ -covered in every neighboring direction \vec{u}' . Quantitatively, let $|\vec{u} - \vec{u}'| \leq \gamma$ then v is $(\epsilon - 2\gamma)$ -covered in direction \vec{u}' . This fact follows from elementary geometry.

The second idea is to use measure concentration. Consider a set of points A on the surface of the unit ball in R^d . If the measure of A is at least δ for some constant δ , then the γ -neighborhood of A (i.e., all points within distance at most γ from some point in A) has measure almost 1. Quantitatively, the measure is approximately $1 - e^{-t^2/2}$ if $\gamma = t/\sqrt{d}$ (i.e., t standard deviations).

Returning to our proof, we have a point v that is ϵ -covered for a set of directions A of measure δ . Moreover, the covering points are within Δ of v , and so we are working with a ball of radius $\sqrt{\Delta}$ rather than 1. So, a $\gamma = t\sqrt{\Delta/d}$ neighborhood of A has measure $1 - e^{-t^2/2}$. So, v is also $(\epsilon/2, \delta')$ -covered for δ' very close to 1.

It might seem that we are almost done by the following induction argument: v is covered in almost all directions. Pick a direction \vec{u} at random and let v' cover v in this direction. Now v' is also almost covered in almost all directions. Surely, v' has a good chance of being covered in direction \vec{u} since it was chosen at random. Of course this argument, as it stands, is nonsense because the choice of v' is conditioned by the choice of \vec{u} . Here is a potential fix: for random \vec{u} with high-probability v is covered both in direction \vec{u} and $-\vec{u}$. If x and y are the covering points in these two directions, then $x - y$ has a projection on \vec{u} that is twice as long. This time the problem is that most nodes v might yield the same pair of points x and y thus making it impossible to continue with an induction. For example, in a k -dimensional hypercube with $n = 2^k$, every point is $\Theta(\sqrt{\log n})$ covered in almost all directions. However, for a particular direction, most points are covered by only a few in the tails of the resulting Gaussian distribution.

To actually piece together the path, we perform a more delicate induction. For random direction \vec{u} , node v is covered

with high probability by some node x . Also in direction $-\vec{u}$, with probability 1/2, v is matched to some node y (i.e., $\{v, y\}$ formed a discarded pair). So with probability almost 1/2, y is now covered in direction u by x . Note that this time y takes this role only once for this direction, since $\{v, y\}$ is a matched pair.

To actually carry out, the induction requires some work to ensure that boosting using measure concentration can be carried out. Moreover, the size of the covered set does indeed drop. Indeed, other ideas need to be included to allow the induction to continue for $\sqrt{\log n}$ steps.

7. IMPLICATIONS FOR METRIC SPACE EMBEDDINGS

It is well known that there are different ways of measuring distances; for instance, l_1 , l_2 , and l_p , as well as more exotic methods such as Kullback–Leibler divergence. It is an important question in functional analysis to understand the inter-relationship among these different measures. One frequent method of quantifying this is to look at the *minimum distortion* required to realize one distance function with the other distance function.

Let (X_1, d_1) and (X_2, d_2) be two metric spaces, by which we mean that d_i is a *distance function* on X_i that is nonnegative, and satisfies triangle inequality. An *embedding* of X_1 into X_2 is a mapping $f: X_1 \rightarrow X_2$. Its distortion is the minimum C such that

$$d_1(x_1, x_2) \leq d_2(f(x_1), f(x_2)) \leq C \cdot d_1(x_1, x_2) \quad \forall x_1, x_2 \in X_1 \quad (1)$$

The minimum distortion of X_1 into X_2 is the lowest distortion among all embeddings of X_1 into X_2 . Though this notion had been studied extensively in functional analysis, its importance in algorithm design was realized only after the seminal 1994 paper of Linial, London, and Rabinovich.¹⁵ Many algorithmic applications of this idea have been subsequently discovered.

It was realized before our work that the accuracy of the approach to cut problems involving SDPs (see Section 2) is closely related to analyzing the minimum distortion parameter linking different metric spaces (see the survey¹⁷). Indeed, the ARV analysis can be viewed as an embedding with low “average” distortion, and subsequent work of Chawla, Gupta, Raecke⁷ and Arora, Lee, and Naor⁵ has been built upon this observation. The final result is a near-resolution of an old problem in analysis: what is the minimum distortion required to embed an n -point l_1 space (i.e., where the points are vectors and distance is defined using the l_1 metric) into l_2 ? It was known for a long time that this distortion is at least $\sqrt{\log n}$ and at most $O(\log n)$. The Arora–Lee–Naor paper shows that the lower bound is close to truth: they give a new embedding whose distortion is $O(\sqrt{\log n \log \log n})$.

We note here a connection to a general demand version of the sparsest cut problem: given a set of pairs of vertices $(s_i, t_i)(s_k, t_k)$, find the cut that minimizes the ratio of number of cut edges and the number pairs that are separated. An illustrative application is how to place few listening devices in a network to listen to many pairs of communicating agents; minimizing the ratio of listening devices to compromised pairs of agents.

The approximation ratio for the natural SDP relaxation

turns out to be equal to the distortion required to embed L_2^2 metrics into L_1 . The embeddings of $5,7$ actually embed L_2^2 into L_2 (which in turn embeds with no distortion into L_1), thus implying an $O(\sqrt{\log n} \log \log n)$ approximation algorithm for this general form of graph partitioning.

Acknowledgments

This work was supported by NSF grants MSPA-MCS 0528414, CCF 0514993, ITR 0205594, CCF-0515304, CCF-0635357, CCF-0635401 and ITR CCR-0121555.

References

- Alon, N. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.
- Arora, S., Hazan, E., and Kale, S. $O\sqrt{\log n}$ approximation to sparsest cut in $\tilde{O}(n^2)$ time. In *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, pages 238–247, Washington, DC, USA, 2004. IEEE Computer Society.
- Arora, S., Hazan, E., and Kale, S. Multiplicative weights method: a meta-algorithm and its applications—a survey, 2005.
- Arora, S. and Kale, S. A combinatorial, primal-dual approach to semidefinite programs. In *STOC '07: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, pages 227–236, New York, NY, USA, 2007. ACM.
- Arora, S., Lee, J. R., and Naor, A. Euclidean distortion and the sparsest cut. *J. Amer. Math. Soc.*, 21(1):1–21, 2008 (Electronic).
- Arora, S., Rao, S., and Vazirani, U. Expander flows, geometric embeddings and graph partitioning. In *STOC '04: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, pages 222–231, New York, NY, USA, 2004. ACM.
- Chawla, S., Gupta, A., and Raecke, H. Embeddings of negative-type metrics and an improved approximation to generalized sparsest cut. In *SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 102–111, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
- Cheeger, J. A lower bound for the smallest eigenvalue of the Laplacian. In *Problem in Analysis*, pages 195–199, 1970.
- Goemans, M. X. Semidefinite programming and combinatorial optimization. In *Proceedings of the International Congress of Mathematicians, Vol. III (Berlin, 1998)*, pages 657–666, 1998 (electronic).
- Goemans, M. X. and Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, 1995.
- Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Comput.*, 20(1):359–392, 1998.
- Khandekar, R., Rao, S., and Vazirani, U. Graph partitioning using single commodity flows. In *STOC '06: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, pages 385–390, New York, NY, USA, 2006. ACM Press.
- Lee, J. R. On distance scales, embeddings, and efficient relaxations of the cut cone. In *SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 92–101, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
- Leighton, T. and Rao, S. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM (JACM)*, 46(6):787–832, 1999.
- Linial, N., London, E., and Rabinovich, Y. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.
- Orecchia, L., Schulman, L., Vazirani, U., and Vishnoi, N. On partitioning graphs via single commodity flows. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17–20, 2008*, pages 461–470, 2008.
- Shmoys, D. B. Cut problems and their application to divide and conquer. In D. S. Hochbaum, ed., *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, 1995.
- Sinclair, A. and Jerrum, M. Approximate counting, uniform generation and rapidly mixing Markov chains (extended abstract). In *Graph-Theoretic Concepts in Computer Science (Staffelstein, 1987)*, volume 314 of *Lecture Notes in Comput. Sci.*, pages 134–148, Berlin, 1988. Springer.

Sanjeev Arora (arora@cs.princeton.edu)
Computer Science Department,
Princeton University,
Princeton, NJ 08544, USA

Satish Rao (satishr@cs.berkeley.edu)
Computer Science Department, UC,
Berkeley, CA 94720, USA

© 2008 ACM 0001-0782/08/1000 \$5.00

- semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, 1995.
11. Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Comput.*, 20(1):359–392, 1998.
12. Khandekar, R., Rao, S., and Vazirani, U. Graph partitioning using single commodity flows. In *STOC '06: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, pages 385–390, New York, NY, USA, 2006. ACM Press.
13. Lee, J. R. On distance scales, embeddings, and efficient relaxations of the cut cone. In *SODA '05: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 92–101, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
14. Leighton, T. and Rao, S. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM (JACM)*, 46(6):787–832, 1999.
15. Linial, N., London, E., and Rabinovich, Y. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.
16. Orecchia, L., Schulman, L., Vazirani, U., and Vishnoi, N. On partitioning graphs via single commodity flows. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17–20, 2008*, pages 461–470, 2008.
17. Shmoys, D. B. Cut problems and their application to divide and conquer. In D. S. Hochbaum, ed., *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, 1995.
18. Sinclair, A. and Jerrum, M. Approximate counting, uniform generation and rapidly mixing Markov chains (extended abstract). In *Graph-Theoretic Concepts in Computer Science (Staffelstein, 1987)*, volume 314 of *Lecture Notes in Comput. Sci.*, pages 134–148, Berlin, 1988. Springer.
- A previous version of this paper, entitled “Expander Flows, Geometric Embeddings, and Graph Partitioning,” was published in *Proceedings of the 36th Annual Symposium on the Theory of Computing* (Chicago, June 13–16, 2004).

The *Asian Language Information Processing Transaction (TALIP)* publishes high quality original archival papers and technical notes in the areas of computation and processing of information in Asian languages and related disciplines. Some of the subjects to be covered by this quarterly publication are: Computational Linguistics; Linguistic Resources; Hardware and Software Algorithms and Tools for Asian Language Processing; Machine Translation; and Multimedia Asian Information Processing. Emphasis will be placed on the originality and the practical significance of the reported research.

To learn more about *TALIP*, please visit
www.acm.org/pubs/talip/

SUBSCRIBE TODAY!

PRODUCT INFORMATION

ISSN: 1530-0226
Order Code: 138
Price: \$38 Professional Member
\$33 Student Member
\$160 Non-Member
\$16 Air Service (for residents outside North America only)

TO PLACE AN ORDER

Contact ACM Member Services

Phone: 1.800.342.6626 (U.S. and Canada)

+1.212.626.0500 (Global)

Fax: +1.212.944.1318

(Hours: 8:30am–4:30pm, Eastern Time)

Email: acmhelp@acm.org

Mail: ACM Member Services

General Post Office

PO Box 30777

New York, NY 10087-0777 USA

www.acm.org/pubs/talip/