St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 9: Metrics
Fall 2021

1 Going the Distance

If you recall our previous discussions on the k-center problem and the traveling salesperson problem, you'll remember that these particular problems involved distances: the k-center problem focused on the distance from a vertex to its cluster center, while the traveling salesperson problem focused on the distances between cities (which we interpreted as a cost to travel between those cities). We were able to make some elementary assumptions about the distances in these problems; for example, we could assume that $d_{uv} = d_{vu}$ for all vertices u and v, since the distance from u to v must necessarily be the same as the distance from v to u in two dimensions, such as on a plane or map.

The distances in these problems, as well as the assumptions we made about each distance, provided our first exposure to *metrics*. Given a set of vertices V, a metric (V, d) gives us a way to measure the distance d_{uv} between two vertices $u, v \in V$. A metric satisfies three properties:

1. $d_{uv} = 0$ if and only if $u = v$;
2. $d_{uv} = d_{vu}$ for all $u, v \in V$; and
3. $d_{uv} \leq d_{uw} + d_{wv}$ for all $u, v, w \in V$.

You might recognize the third property as the familiar *triangle inequality*.

Metrics are a useful tool that we can apply to a certain class of graph problems involving *cuts*. Much like the weighted maximum cut problem we saw previously, a graph cut is simply a partitioning of the vertices of a graph into disjoint subsets. Why are metrics particularly useful for graph cut problems, though? As it turns out, we can define a metric where $d_{uv} = 1$ if vertices u and v belong to different subsets, and $d_{uv} = 0$ otherwise.[1] With this metric, we can take a problem that asks us to maximize or minimize the sum of edge weights in a cut, and simplify it to use the cut metric.

In this lecture, we will consider some approximation algorithms for graph cut problems, and we will see how to incorporate cut metrics into approximation algorithms for graph cut problems.

1.1 Minimum Multiway Cuts

Our first graph cut problem is known as the *multiway cut problem*. This problem is a generalization of the *s-t cut problem*, which is quite similar to the shortest s-t path problem we saw in our previous lecture. In the s-t cut problem, we must find a cut in a graph such that two "distinguished vertices" s and t do not belong to the same connected component of the cut graph, while in the multiway cut problem, we are given a subset of k distinguished vertices and we must ensure that no two distinguished vertices belong to the same connected component of the cut graph.

Here, we will consider a variant of the problem where each edge of the graph has an associated cost, and we must additionally ensure that the cut our algorithm obtains is of minimum cost.

[1] Note that, strictly speaking, this metric doesn't satisfy the first property, which says that $d_{uv} = 0$ if and only if $u = v$. A metric that only satisfies the second and third properties is more correctly called a *semimetric*, but here we will just continue to use the general term "metric".

MINIMUM-MULTIWAY-CUT
Given: an undirected graph $G = (V, E)$, an associated cost $c_e \geq 0$ for each edge $e \in E$, and a subset of k distinguished vertices $\{s_1, \ldots, s_k\} \in V$
Determine: a minimum-cost set of edges F such that no pair of distinguished vertices s_i and s_j, $i \neq j$, are in the same connected component of $G' = (V, E \setminus F)$

Note that, if $k = 2$, then the multiway cut problem becomes the s-t cut problem. By the *max-flow min-cut theorem*, the cost of a minimum s-t cut in a graph is equivalent to the maximum flow amount through that graph, and we have efficient polynomial-time algorithms for computing the maximum flow amount. For $k \geq 3$, however, the minimum multiway cut problem becomes NP-hard, and so this fact makes the problem a good candidate for an approximation algorithm.

In order to discuss one approach to handling the minimum multicut problem, we must introduce the notion of an *isolating cut*. An isolating cut is essentially the set of edges in a graph that we must remove in order to isolate one distinguished vertex s_i from the rest of the distinguished vertices. Therefore, any multiway cut will consist of a total of k isolating cuts: one for each distinguished vertex.

However, we require the additional condition that our multiway cut has minimum cost, so how do we ensure that each isolating cut has minimum cost? We simply need to add a new *sink vertex* to our graph, denoted t, as well as new infinite-cost edges from each distinguished vertex s_i to t. Then, we can reformulate our minimum multiway cut problem in terms of finding k minimum s_i-t cuts in the graph, for all k distinguished vertices s_i.

The last step we will take in our algorithm is to remove the cut of greatest cost. If we isolate $k - 1$ distinguished vertices, then the kth distinguished vertex will belong to a connected component containing only itself and no other distinguished vertices. Therefore, we can save one step by not having to find an isolating cut for that vertex. If we assume that isolating cut has the greatest cost, then we can reduce the total cost of our multiway cut by not including that isolating cut.

Algorithm 1: Multiway cut—greedy

for $1 \leq i \leq k$ **do**
    compute a minimum-cost isolating cut F_i for vertex s_i
$F \leftarrow \bigcup_{j=1}^{k-1} F_j$ ▷ assume wlog that $c(F_1) \leq c(F_2) \leq \cdots \leq c(F_k)$

If we were to simply compute minimum-cost isolating cuts for each distinguished vertex, then we would obtain a 2-approximation algorithm for the minimum multiway cut problem. However, that last step of the algorithm gives us an approximation that is just slightly better.

Theorem 1. *Algorithm 1 gives a $(2 - \frac{2}{k})$-approximation algorithm for the minimum multiway cut problem.*

Proof. Let F^* be an optimal multiway cut in G. We can view F^* as the union of k isolating cuts, since removing the edges of F^* from G will create k connected components, each of which contains one distinguished vertex. Moreover, since F^* is a minimum multiway cut, no more than k connected components will be created. If we take F_i^* to be the cut separating vertex s_i from the rest of the distinguished vertices, then $F^* = \bigcup_{j=1}^{k} F_j^*$.

Since each edge in F^* is incident to two connected components, each edge will belong to two isolating cuts F_i^*. Therefore,

$$\sum_{i=1}^{k} c(F_i^*) = 2 \cdot c(F^*).$$

Since F_i is a minimum-cost isolating cut for vertex s_i, we have that $c(F_i) \leq c(F_i^*)$. By discarding the

isolating cut of greatest cost from F, we get

$$\begin{aligned} c(F) &\leq \left(1 - \frac{1}{k}\right) \cdot \sum_{i=1}^{k} c(F_i) \\ &\leq \left(1 - \frac{1}{k}\right) \cdot \sum_{i=1}^{k} c(F_i^*) \\ &= 2\left(1 - \frac{1}{k}\right) \cdot c(F^*) \\ &= \left(2 - \frac{2}{k}\right) \cdot \text{OPT}. \end{aligned}$$

□

Going further, we can use randomized rounding techniques to obtain an approximation algorithm with a performance guarantee of $\frac{3}{2}$ for the minimum multiway cut problem.

1.2 Minimum Multicuts

Our next graph cut problem is known as the *multicut problem*. Even though the name of this problem is quite similar to that of our previous problem, the multicut problem is in fact more general than the multiway cut problem. (Why computer scientists thought it was a good idea to give such similar names to these different problems, we may never know.)

In the multicut problem, we are given a set of pairs of vertices (s_i, t_i), and we must find a cut in a graph such that every vertex s_i in each pair belongs to a different connected component from the corresponding vertex t_i in the pair. Note that this doesn't prohibit vertices s_i and s_j, $i \neq j$, from belonging to the same connected component, nor does it prohibit vertices s_i and t_j, $i \neq j$, from belonging to the same connected component. All we care about is separating the vertices s_i and t_i in the same pair.

Again, we consider here the "weighted" variant of the problem, where our algorithm must find a graph cut of minimum cost.

MINIMUM-MULTICUT
Given: an undirected graph $G = (V, E)$, an associated cost $c_e \geq 0$ for each edge $e \in E$, and a set of vertex pairs $\{(s_1, t_1), \ldots, (s_k, t_k)\}$, where $s_i, t_i \in V$ for all i
Determine: a minimum-cost set of edges F such that, for all i, there is no path connecting s_i and t_i in $G' = (V, E \setminus F)$

For this problem, we could simply modify Algorithm 1 to obtain a $(2 - \frac{2}{k})$-approximation algorithm just as before. However, here we will focus on taking a linear programming approach. For a given graph G, take a vertex pair (s_i, t_i), and let P_i denote the set of all paths P in G from s_i to t_i. We can then formulate our linear program as follows:

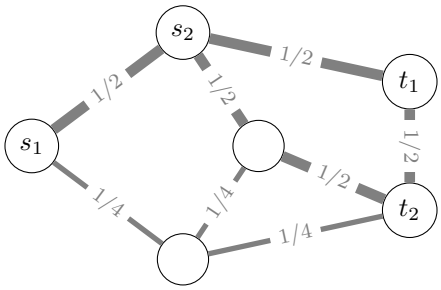
$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e x_e \\ \text{subject to} \quad & \sum_{e \in P} x_e \geq 1, \quad \text{for all } P \in P_i \\ & x_e \geq 0, \quad \text{for all } e \in E \end{aligned} \tag{MMC-LP}$$

Even though this linear program could possibly have an exponential number of constraints in the size of the graph, we can find a solution to the linear program in polynomial time using the ellipsoid method. All we require is an appropriate separation oracle for the problem.

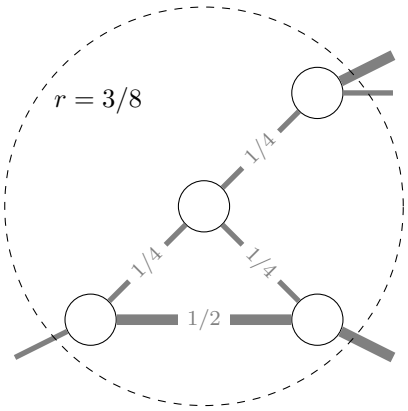
What sort of separation oracle could we take in this case? It suffices to take an oracle that is capable of solving the shortest s-t path problem, as that is essentially what we want to compute for each pair of vertices (s_i, t_i). If we take a graph G where the length of an edge e is denoted by x_e, then the constraint $\sum_{e \in P} x_e \geq 1$ will be satisfied whenever the shortest s-t path has a length of at least 1, and any s-t path with a length

less than 1 violates some primal inequality. Thus, if we know the shortest s-t path has a length of at least 1, then all paths from s to t must have a length of at least 1, and so we can take x as a feasible solution.

To gain a better understanding of how the linear program MMC-LP works, we can interpret the given graph as a network of pipes. If an edge e exists between two vertices i and j, then we represent that edge by a pipe of some length and diameter. The length of the pipe is denoted by x_e, while the diameter of the pipe is denoted by c_e. Taken together, this gives the volume of the pipe as $c_e x_e$: the value that appears in the objective. Thus, the linear program is effectively finding a minimum-volume system of pipes where the distance between vertices s_i and t_i is at least 1 for all i.



Suppose we are given a feasible solution to the linear program MMC-LP, and let $d_x(u, v)$ denote the shortest path length between vertices u and v with edge lengths x_e. Additionally, we will take $B_x(s_i, r) = \{v \in V \mid d_x(s_i, v) \leq r\}$ to be the *ball* of radius r around vertex s_i with edge lengths x_e. Essentially, the ball is the set of vertices within a distance r from vertex s_i.



With this interpretation of the multicut problem, and with the notion of balls, we present the following approximation algorithm for the problem.

Algorithm 2: Multicut—linear programming

$F \leftarrow \emptyset$
solve the program MMC-LP and get an optimal solution x
while there exists some connected pair of vertices (s_i, t_i) in G **do**
  $S \leftarrow B_x(s_i, r)$ for some choice of $r < 1/2$
  $F \leftarrow F \cup \delta(S)$ ▷ $\delta(S)$ is the set of edges with exactly one endpoint in S
  remove S and $\delta(S)$ from G

It is straightforward to show both that the algorithm terminates and that the algorithm gives a valid multicut of the graph. Thus, the remaining question is: how good of an approximation can we get with this algorithm?

Suppose that $V^* = \sum_{e \in E} c_e x_e$ is an optimal solution for the linear program MMC-LP. Then, we know that $V^* \leq \text{OPT}$. We will also denote by $V_x(s_i, r)$ the total volume of the pipes in the ball of radius r surrounding

vertex s_i, plus an extra additive term V^*/k; that is,

$$V_x(s_i, r) = \sum_{\substack{e=(u,v) \\ u,v \in B_x(s_i,r)}} c_e x_e + \sum_{\substack{e=(u,v) \\ e \in \delta(B_x(s_i,r))}} c_e(r - d_x(s_i, u)) + \frac{V^*}{k}$$

The first term sums the volumes of all edges (u, v) such that both vertex u and vertex v are in the ball of radius r around s_i, while the second term sums the volumes of any pipes that straddle the radius of the ball but don't fall completely within the ball. The third term, V^*/k, ensures that $V_x(s_i, 0)$ is nonzero and that the sum over all s_i is V^*.

Evidently, $V_x(s_i, r)$ is an increasing function of r, since the ball of radius r necessarily contains more volume as the value of r increases.

We will also denote by $C_x(s_i, r)$ the total cost of the cut defined by the vertices in the ball of radius r surrounding vertex s_i; that is,

$$C_x(s_i, r) = \sum_{e \in \delta(B_x(s_i,r))} c_e.$$

Interestingly, we can observe a relationship between $V_x(s_i, r)$ and $C_x(s_i, r)$. We will not prove this relationship here, but it makes use of the observation that $\frac{\mathrm{d}(V_x(s_i,r))}{\mathrm{d}r} = C_x(s_i, r)$, which follows as a consequence of their definitions.

The following relationship tells us that it is always possible to find some value r such that the cost of the cut induced by the ball of radius r surrounding vertex s_i is within a logarithmic factor of the volume of the ball itself, thus giving us an upper bound between the two values.

Lemma 2. *There exists a value $r < \frac{1}{2}$ such that*

$$\frac{C_x(s_i, r)}{V_x(s_i, r)} \leq 2\ln(2k).$$

Furthermore, we can find such a value r in polynomial time.

Note that this value r is exactly what we use in our algorithm. We can then use the above relationship to establish the performance guarantee of our algorithm.

Theorem 3. *Algorithm 2 gives a $4\ln(2k)$-approximation algorithm for the minimum multicut problem.*

Proof. Let B_i denote the set of vertices in the ball $B_x(s_i, r)$ selected by Algorithm 2 when the pair of vertices (s_i, t_i) are separated in the graph G. Furthermore, let F_i denote the set of edges in $\delta(B_i)$ when the vertices of B_i and its incident edges are removed from G. Then, $F = \bigcup_{i=1}^{k} F_i$. Lastly, let V_i denote the total volume of edges removed when the vertices of B_i and its incident edges are removed from G.

Note that $V_i \geq V_x(s_i, r) - (V^*/k)$, since V_i contains the full volume of all edges in F_i, while $V_x(s_i, r)$ contains only part of the volume of edges that straddle the radius of the ball, plus the extra term V^*/k.

By our choice of r in Lemma 2, we have that

$$c(F_i) \leq 2\ln(2k)V_x(s_i, r) \leq 2\ln(2k)(V_i + (V^*/k)).$$

Furthermore, we have that the volume of each edge belongs to at most one V_i, since removing the edge from G implies it cannot be included in the volume of a ball B_j removed in some later iteration of the algorithm. This implies that $\sum_{i=1}^{k} V_i \leq V^*$.

Altogether, we get that

$$\begin{aligned}\sum_{e\in F} c_e &= \sum_{i=1}^{k}\sum_{e\in F_i} c_e \\ &\leq 2\ln(2k)\cdot\sum_{i=1}^{k}\left(V_i + \frac{V^*}{k}\right) \\ &\leq 4\ln(2k)\cdot V^* \\ &\leq 4\ln(2k)\cdot \mathrm{OPT}.\end{aligned}$$

□