University of Michigan–Ann Arbor

Department of Electrical Engineering and Computer Science

EECS 498 004 **Advanced Graph Algorithms**, Fall 2021

Lecture 10: Deterministic Vertex Connectivity via Ramanujan Graphs

September 30, 2021

Instructor: Thatchaphol Saranurak

Scribe: Chaitanya Nalam

- Today, we will show an application of Ramanujan graphs to fast graph algorithms.

1 Vertex Connectivity

Let $G = (V, E)$ be an unweighted and undirected graph. We introduce some basic definitions related to vertex connectivity below.

Definition 1.1 (Vertex Connectivity)**.** Given a connected graph G, the smallest number of vertices needed to be deleted from G to disconnect G.

Definition 1.2 (Vertex cut)**.** For $G = (V, E)$, partition (L, S, R) on set of vertices V is said to be a vertex cut if $L, R \neq \phi$ and $E_{G \setminus S}(L, R) = \phi$, i.e. vertices in S separates L from R. $|S|$ is called the size of vertex cut.

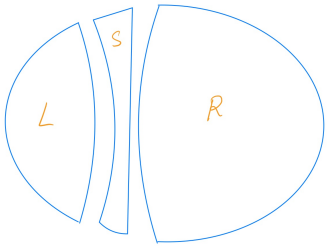

Figure 1: Vertex Cut

The size $|S^*|$ of the smallest possible vertex cut (L^*, S^*, R^*) is equal to the vertex connectivity of graph G denoted by $\kappa(G)$ or just κ.

Definition 1.3 ((s,t)−vertex cut)**.** A set $S \subset V$ is said to be a (s,t)− vertex cut for vertices $s, t \in V$ if removing the nodes in S will disconnect vertex s from t. It is also called (s,t)−separator.

Definition 1.4 ((s,t)− vertex mincut)**.** The smallest possible (s,t)− vertex cut is called (s,t)− vertex mincut or minimum (s,t)−separator and denote its size by $\kappa(s,t)$.

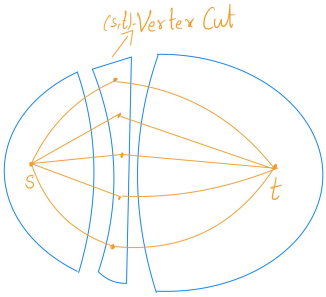

Figure 2: (s,t)-Vertex Cut

1.1 Naive Method

By max flow min cut theorem $\kappa(s,t)$ is the maximum number of vertex disjoint paths. Hence, we can compute $\kappa(s,t)$ in one max-flow call.

Hence, $\kappa(G)$ is obtained by taking minimum over $\kappa(s,t)$ all possible (s,t) pairs in the graph and finding the minimum separator of the graph.

$$\kappa(G) = \min_{s,t} \kappa(s,t)$$

However, this naive method takes $O(n^2)$ max flow calls. Today we will see a deterministic algorithm that takes $O(n^{1.5} \log n)$ max flow calls based on the paper by [Gabow'00].

1.2 State of the Art

There is a deterministic algorithm by [Gabow'00] that takes $O(m \cdot (n + \min\{\kappa^{5/2}, \kappa n^{3/4}\}) = O(mn^{1.75})$ time when κ is as big as $O(n)$. [1] It is the only fast algorithm that exploits Ramanujan Graphs as far as we know. This algorithm is much slower than the current algorithm that we present.

There is a randomized algorithm by [LNP**S**Y'21] which takes polylog(n) "max flow calls". Current best algorithm for max flow runs in $O(m^{4/3+o(1)})$ time. [2] The max flow calls are made on smaller graphs with fewer edges. The total size of the graph that they run max flow is still near-linear $\tilde{O}(m)$.

1.3 Overall Plan

Given an unweighted undirected graph G, the goal would be to return the minimum vertex cut (L, S, R) of size $\kappa(G)$.

Note: **We assume** $\delta \leq 9n/10$ to avoid annoying corner case where δ denote the minimum vertex degree.

[1] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.1674&rep=rep1&type=pdf
[2] https://arxiv.org/abs/2104.00104

2 Algorithm for vertex connectivity

There are two cases depending on how balanced the vertex cut is. We first assume that the minimum vertex cut is balanced and find it. Next we reduce the unbalanced case to balanced case.

Definition 2.1 ($\beta-$balanced vertex cut)**.** A vertex-cut (L, S, R) is said to be β**-balanced** if $|L| \geq \beta$ and $|R| = \Omega(n)$.

The main idea underlying the algorithm is as follows. Let (L^*, S^*, R^*) be the minimum vertex cut of size k. If we some how know a vertex $x \in L^*$ and $y \in R^*$, then we can do a max flow call to find minimum $(x, y)-$ vertex cut (L, S, R) whose size is same as $|S^*|$ (although exact cut may be different). This is because S^* is a $(x, y)-$vertex cut whence $\kappa(x, y) \leq k$; otoh since (L^*, S^*, R^*) is minimal, $\kappa(x, y) \geq k \implies \kappa(x, y) = k$.

However, we do not know which pair of vertices belongs to L^*, R^* respectively, whence the naive way is to try all possible pairs however this may take as large as $O(n^2)$ max flow calls.

But can we reduce the number of max flow calls using the "promise" that our vertex cut is $\beta-$balanced?

Yes, this is our next idea for an improved algorithm. By using the structural property of the Ramanujan Graphs we can decrease the number of pairs on which we need to call max flow, for finding the $\beta-$balanced cut. We will state the structural property of the Ramanujan Graph here without proof.

Corollary 2.2. *Let $H = (V, E)$ be a d-regular Ramanujan graph. For any $L, R \subseteq V$, if $|L| \cdot |R| \cdot d > 4n^2$, then $E(L, R) \neq \emptyset$.*

We want to identify a vertex pair such that they belong to either sides of the minimum vertex cut (L^*, S^*, R^*). From the above corollary we have that whenever $|L| \cdot |R| \cdot d > 4n^2$ then there always exist an edge from L to R.

Since we want to find a vertex pair between L^* and R^* when $|L^*| \geq \beta$ and $|R^*| = \Omega(n)$ we can as well set $d = \Theta(n/\beta)$ such that $|L^*| \cdot |R^*| \cdot d > 4n^2$ is satisfied and then by the property of Ramanujan Graph (H) we have a guaranteed edge between L^*, R^*.

However, we do not know which edge is between L^* and R^* (or really, we don't know where L^* and R^* are) so we use all edges of Ramanujan Graph (E_H) as candidate pairs for calling max flow and take the minimum value of the vertex cut between all of them. By the property of Ramanujan Graph and the balanced nature of minimum vertex cut we are guaranteed to find such a pair which gives us minimum cut.

Lemma 2.3. *If there exists a β-balanced vertex-mincut (L^*, S^*, R^*), then we can find a vertex-cut of size at most k in $O(n \times \frac{n}{\beta})$ max flow calls.*

Proof. We use the edges of H, E_H to get candidate pairs for finding the balanced cut. Since there can be at most $n \cdot d$ edges in a $d-$regular graph, $|E_H| = O(n^2/\beta)$ which implies at most $O(n^2/\beta)$ many max flow calls. □

Note that H is totally different graph from G but shares the same vertices.

3 Reduction to Balanced Case

We saw above that we can improve over the naive $O(n^2)$ max flow calls to $O(n^2/\beta)$ calls when we know that minimum vertex cut is balanced. However we do not know if the (L^*, S^*, R^*) is balanced. So in this section we present a way to reduce the unbalanced case to balanced case.

Our plan would be to gradually modify the graph to make it balanced and then solve it in the above mentioned way. We measure the balanced-ness of the minimum cut using a condition called the **gap condition**. We keep on improving gap condition until that it is sufficient to imply "every" minimum cut is $\beta-$balanced.

Note that balanced case requires that at least one of the all possible minimum cuts need to be balanced. But gap condition ensures that all minimum cuts are balanced.

3.1 Gap Condition

We know that $\kappa(G) = \kappa \leq \delta$ because a vertex with degree δ can be separated from the rest of the graph by removing its neighbours which are of size δ. Hence minimum cut can only be either smaller or equal.

Since $\kappa \leq \delta$, the difference between the min degree and min cut, $\delta - \kappa$ is called **Gap**. Why is this quantity useful? It is formalized in the lemma below.

Lemma 3.1. *Every mincut (L^*, S^*, R^*) is β-balanced where $\beta = \delta - \kappa$.*

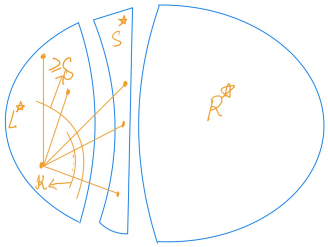

Figure 3: Gap Condition

Proof. Let $x \in L^*$ (we know L^* is non empty) and $N(x)$ be the set of x's neighbors. We have $N(x) \subseteq L^* \cup S^*$ as it cannot have edges to R^*.

$$
\begin{aligned}
|N(x)| &\geq \delta && \text{(minimum degree)} \\
|N(x)| &\leq |L^*| + |S^*| = |L^*| + \kappa \\
|L^*| &\geq \delta - \kappa
\end{aligned}
$$

There are always $\delta - \kappa$(**the gap** between minimum degree and minimum cut) vertices on both sides of the minimum cut. □

So to make the minimum cut balanced we need to increase the gap $\delta - \kappa$. We need more structural understanding of the vertex connectivity to see why we can do this and how to do this.

Definition 3.2 (x-rooted vertex connectivity)**.** The minimum number of vertices needed to disconnect x from any other vertex in the graph.

$\kappa(x) = \min_{y \neq x} \kappa(x, y)$ which can be computed in $n - 1$ max flows.

Lemma 3.3. *If* $\kappa(x) > \kappa(G)$*, then* $\kappa(G \setminus x) = \kappa(G) - 1$.

Proof. If $\kappa(x) > \kappa(G)$ then for all possible min cuts (L^*, S^*, R^*), $x \in S^*$ if not $\kappa(x) = \kappa(G)$.

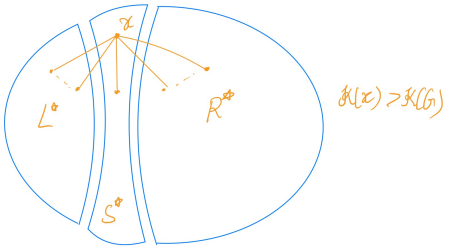

Figure 4: Remove vertex to decrease κ

$\kappa(G \setminus x)$ cannot be $< \kappa(G) - 1$ because then we can add x back and get a min cut for graph G of size $< \kappa(G)$ which is contradiction.

$\kappa(G \setminus x)$ need not be $> \kappa(G) - 1$ because we already have a cut of size $= \kappa(G) - 1$ which is $S^* \setminus \{x\}$ where S^* is any min cut separator and we also know that $x \in S^*$. □

We use the above structural lemma as follows. Compute $\kappa(x)$ along with the corresponding cut. If $\kappa(x) = \kappa$ then we have a found a vertex cut of optimal size. Otherwise $\kappa(x) > \kappa$ then remove x from the graph and compute $\kappa(G \setminus x)$ and according to above lemma we know that $\kappa(G) = \kappa(G \setminus x) + 1$.

From now on we use κ to always refer to the minimum vertex cut size of the current graph which is $G \setminus x$ after x is removed.

However, if we repeat the above step it may take $O(n)$ steps to find min-cut as κ can be $O(n)$. Since each step takes $O(n)$ max flows we are again at square one. Recall our target is to increase the gap between δ and κ.

Using the above step we have reduced κ by 1. So gap should increase if δ has stayed the same. However, by removing x, δ can also decrease by 1 if $N(x)$ contains some vertex of minimum degree, in which case the gap cannot be increased.

Let F be the set of neighbors of x such that $\forall x' \in F$, the degree of x' become $\delta - 1$ after removing x. So all nodes in F need to be "fixed" so that their degree become δ again. We can fix them without affecting the min cut size for which we need the following structural lemma.

Lemma 3.4. *If* $\kappa(x, y) > \kappa$*, then* $\kappa(G \cup (x, y)) = \kappa(G)$.

Proof. $\kappa(x, y) > \kappa$ then x, y always belong to same side either $L^* \cup S^*$ or $S^* \cup R^*$ for every possible min cut (L^*, S^*, R^*). If not $\kappa(x, y)$ would have been κ. So adding an edge between x, y (if it does not already exist) does not affect any minimum vertex cut. □

For any vertex w compute $\kappa(w)$ and if it is $= \kappa$ then we are done, else we can add edges to any other vertex from this without effecting the min cut as it belongs to S^* for every min cut (L^*, S^*, R^*).

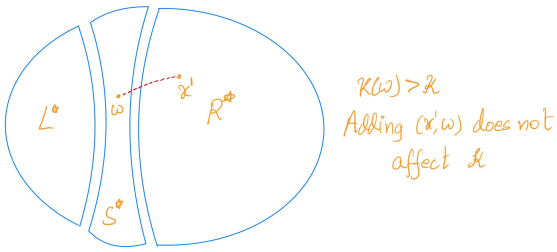

Figure 5: Add edge (w, x') to G

- Let $F_w = \{x' \in F \mid (x', w) \notin E\}$ be all nodes that need to be fixed, and **not** adjacent to w.
- For each $x' \in F_w$, we can add the edge (x', w) into the graph without changing κ.
- So all nodes in F_w are now fixed!

However, F_w might be very small when compared to F and may take many max flows to fix all vertices in F. We show below that it is not the case.

Lemma 3.5. *Let G' be the current graph after removing x. There exists $w \in V(G')$ such that*

$$|F_w| \geq |F| \cdot (1 - \frac{\delta - 1}{|V(G')|})$$

.

Proof. We have $\text{vol}_{G'}(F) \leq |F|(\delta - 1)$ as each vertex in F is of degree $\delta - 1$. So there is a vertex $w \in V(G')$ such that $|E_{G'}(w, F)| \leq \frac{|F|(\delta-1)}{|V(G')|}$.

That is, w is adjacent to at most $\frac{|F|(\delta-1)}{|V(G')|}$ vertices in F. So we can fix at least $|F| \cdot (1 - \frac{\delta-1}{|V(G')|})$ vertices in F by adding edges to w. □

We will see that the algorithm will guarantee $|V(G')| \geq n - \sqrt{n}$, and we assume $\delta \leq \frac{9}{10}n$ from the beginning. Hence, $|F_w| \geq |F| \cdot (1 - \frac{\delta-1}{|V(G')|}) \geq |F|/100$. So we fix at least $(1/100)^{th}$ fraction of vertices in F every time by incurring an overhead of n max flows for computing $\kappa(w)$.

We find w by scanning through all possible vertices and find the vertex w that fixes at least $(1/100)^{th}$ fraction of vertices in F, which is guaranteed to exist based on the above lemma. So it takes a total of $O(\log n)$ rooted vertex connectivity calls to fix all vertices. Once all vertices are fixed we have successfully increased the minimum degree to δ without increasing the minimum vertex cut $\kappa(G \setminus x) = \kappa(G) - 1$ and hence increasing the gap by 1. This took a total of $O(n \log n)$ max flow calls.

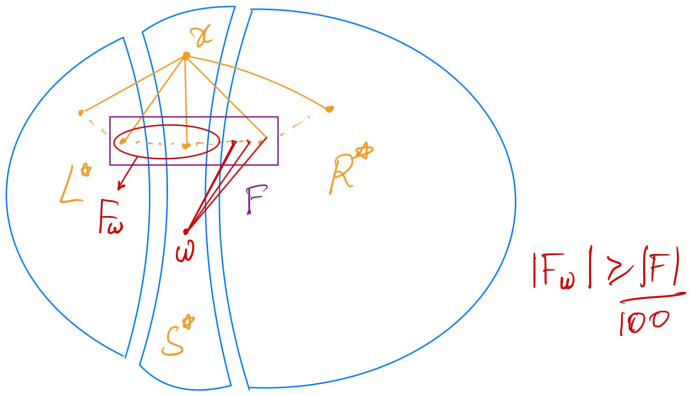

Figure 6: Vertex Cut

Remark 3.6. Note that there is an easy way to just find any arbitrary non-neighbour w of a vertex for a $x' \in F$ and compute $\kappa(x', w)$.

- If $\kappa(x', w) = \kappa$, since we track of minimum vertex cut we found so far, we are done.
- If $\kappa(x', w) > \kappa$, then we can fix $x' \in F$ by adding edge to w.

iHowever we do not choose the above said process and use $O(\log n)$ rooted vertex connectivity calls because, if later we can speed up the algorithm for problem of finding rooted vertex connectivity faster than $o(n)$ max flow calls which immediately implies an improved algorithm for vertex connectivity problem.

Finding rooted vertex connectivity in $o(n)$ max flow calls is an independent and very interesting open problem.

3.2 Summary of the Algorithm in the Unbalanced Case

Algorithm 1: $IncreaseGap(G)$

Result: Graph in which the gap, $\delta - \kappa$, is increased by 1
Choose any $x \in V$. Compute $\kappa(x)$ and the corresponding cut
Let $G' = G \setminus x$
Let $F = \{x' \in N(x) \mid \deg_{G'}(x') = \delta - 1\}$
while $F \neq \phi$ **do**
  Iterate over V to find w such that $|F_w| \geq |F|/100$
  Compute $\kappa(w)$ and the corresponding cut and set $\hat{\kappa}_{G'} \leftarrow \min\{\hat{\kappa}_{G'}, \kappa(w)\}$
  For each $x' \in F_w$, add edge (x', w) to G'
end

Note that if $\kappa(x) > \kappa$, then the gap is increased by 1 and our candidates for minimum cut are $\kappa(x)$ and $\hat{\kappa}_{G'} + 1$; otherwise if $\kappa(x) = \kappa$ or $\kappa(w) = \kappa$, then we had found the minimum vertex cut of G and $\kappa(G)$. If we repeated this process for β times and in each round we have $\kappa(x) > \kappa$ we could increase the gap by β, the vertex min cut in the resulting graph is β-balanced. Meanwhile we keep

track of the current smallest cut and update it using $\min\{\kappa(x), \hat{\kappa}_{G'} + 1\}$ and the corresponding cut. In this way, if at any round we found κ, then it will be recorded.

We finally use the balanced case to solve this graph for minimum vertex cut in the resulting graph and then ripple back through the deletions of vertices to find the minimum cut among all possible candidates.

3.3 Correctness

All the while we are comparing our rooted vertex connectivities with the minimum vertex connectivity κ however we do not know it before hand.

It is not an issue as we only over estimate the minimum cut. If we ever find the minimum cut exactly somewhere in our algorithm. Since we store all the cut values and take the minimum. Hence, we are guaranteed to find the minimum cut.

If we never found the minimum cut while reducing to balanced case, then according to our algorithm we will find the minimum cut after reducing it to balanced case. So, either way the minimum of all our cuts will give us the minimum cut.

3.4 Runtime

- We call $\beta \times O(n \log n)$ max flows to reduce the problem to the β-balanced case.
- In the β-balanced case, we can solve the problem in $O(n \times \frac{n}{\beta})$ max flows.
- So a total of $O(\frac{n^2}{\beta} + \beta n \log n)$ max flow calls. By choosing $\beta = \sqrt{n}$, you can solve the problem in $O(n^{1.5} \log n)$ max flow calls.

4 Exercises and Open Questions

Exercise 4.1. Show an algorithm for vertex connectivity that takes $O(n \cdot \kappa)$ max flow calls.

Proof. At the i^{th} iteration, pick an arbitrary point $x_i \in V$ and compute $\kappa(x)$ using $O(n)$ max flow calls and record $k_i = \min_{j \le i} \kappa(x_j)$. If $k_i \le i - 1$, then return k_i. The algorithm is going to terminate since as long as $i > \kappa$, then we are guaranteed to pick a point that is not in S^* where (L^*, S^*, R^*) is a minimum vertex cut, whence the algorithm halts in at most $\kappa + 1$ iterations. The algorithm is correct since if $i \le \kappa$ and $k_i > \kappa \ge i$ then the algorithm won't halt. □

Question 4.2 (Open)**.** *Can we deterministically solve vertex connectivity using $O(n)$ max flow calls or even less?*

Note the that for randomized case, it is just polylog(n) max flows now. Something should be possible!

Question 4.3 (Open)**.** *The state of the art for **weighted** vertex connectivity.*

- *Randomized:* $\tilde{O}(mn)$
- *Deterministic:* $\tilde{O}(m^2)$

- So if you can find
  - a randomized algorithm using $o(n)$ max flow
  - a deterministic algorithm using $o(n^2)$ max flow
- This would improve the state of the art.
- Any hardness from fine-grained complexity would be interesting too!