1 - Program1.py

2 - Program2.py

3. In a graph G, the distance between two vertices u and v, denoted by d(u, v), is defined to be the length of a shortest path joining u and v in G. (It is possible to define the distance by various more general measures.) The diameter of G, denoted by D(G), is the maximum distance over all pairs of vertices in G. The diameter is one of the key invariants in a graph which is not only of theoretical interest but also has a wide range of applications. When graphs are used as models for communication networks, the diameter corresponds to the delays in passing messages through the network, and therefore plays an important role in performance analysis and cost optimization.

Although the diameter is a combinatorial invariant, it is closely related to eigenvalues. This connection is based on the following simple observation: Let M denote an n × n matrix with rows and columns indexed by the vertices of G. Suppose G satisfies the property that M(u, v) = 0 if u and v are not adjacent. Furthermore, suppose we can show that for some integer t, and some polynomial pt(x) of degree t, we have pt(M)(u, v) 6= 0 for all u and v. Then we can conclude that the diameter D(G) satisfies: $D(G) \leq t$. Suppose we take M to be the sum of the adjacency matrix and the identity matrix and the polynomial pt(x) to be $(1+x)t$. The following inequality for regular graphs which are not complete graphs can then be:

(3.1) $D(G) \leq [\log(n-1)/\log(1/(1-\lambda))]$. Here, λ basically only depends on λ_1. For example, we can take $\lambda = \lambda_1$ if $1 - \lambda_1 \geq \lambda_{n-1} - 1$. In general, we can slightly improve (3.1) by using the same “spectrum shifting” trick. Namely, we define $\lambda = 2\lambda_1/(\lambda_{n-1} + \lambda_1) \geq 2\lambda_1/(2 + \lambda_1)$, and we then have

(3.2) $D(G) \leq [\log(n-1)/((\log \lambda_{n-1} + \lambda_1)/(\lambda_{n-1} - \lambda_1)]$

The bound in (3.1) can be further improved by choosing pt to be the Chebyshev polynomial of degree t. We can then replace the logarithmic function by cosh−1 :

$$D(G) \leq [\cosh^{-1}(n-1) / \cosh^{-1}((\lambda_{n-1}+\lambda_1)/(\lambda_{n-1}-\lambda_1))]$$

The above inequalities can be generalized in several directions. Instead of considering distances between two vertices, we can relate the eigenvalue λ_1 to distances between two subsets of vertices. Furthermore, for any $k \geq 1$, we can relate the eigenvalue λ_k to distances among $k + 1$ distinct subsets of vertices.

We will derive several versions of the diameter-eigenvalue inequalities. From these inequalities, we can deduce a number of isoperimetric inequalities which are closely related to expander graphs. It is worth mentioning that the above discrete methods for bounding eigenvalues can be used to derive new eigenvalue upper bounds for compact smooth Riemannian manifolds. This will be discussed in the last section of this chapter. In contrast to many other more complicated graph invariants, the diameter is easy to compute. The diameter is the least integer t such that the matrix M = I+A has the property that all entries of Mt are nonzero. This can be determined by using O(log n) iterations of matrix multiplication. Using the current best known bound M(n) for matrix multiplication where $M(n) = O(n^{2.373})$ this diameter algorithm requires at most O(M(n) log n) steps. The problem of determining distances of all pairs of vertices for an undirected graph can also be done in O(M(n) log n) time. Sorry, I don’t really wish to make extracts from article about multiplying in $O(n^{2.373})$. I can only say that their method led to analyzing tensor power, finding that $\omega \leq 3\tau$ and assuming that $\tau = 2.372873/3$ during long and complex calculations and proofs. By the way, some scientists believe that real value of ω is 2 (people have nothing to do [it’s a joke =)]).

4. Computing Edge–Connectivity Let G = (V,E) represent a graph (or digraph) without loops or multiple edges, with vertex set V and edge (or arc) set edge E. In a graph G, the degree deg(v) of a vertex v is defined

as the number of edges incident to vertex v in G. The minimum degree $\delta(G)$ is defined as: $\delta(G) = \min\{deg(v) \mid v \text{ in graph } G\}$. In case of a digraph, the in–degree in–deg(v) and the out–degree out–deg(v) are defined respectively as the number of arc incoming to and arcs outgoing from vertex v in G, and the corresponding minimum degree is: $\delta(G) = \min\{\text{in-deg(v), out-degree} \mid v \text{ in digraph } G\}$. Throughout the Chapter, we will denote the order and the size of a graph (or a digraph) by n and m, respectively. Let v and w be a pair of distinct vertices in graph G. We define $\lambda(v, w)$ as the least number of edges whose deletion from G would destroy every path between v and w. In case of a digraph, $\lambda(v, w)$ would represent the least number of arcs whose deletion would destroy every directed path from v to w. Note that in a graph G, we have $\lambda(v, w) = \lambda(w, v)$, whereas the equality may not hold in case of a digraph. The edge–connectivity $\lambda(G)$ of a graph G is the least cardinality $|S|$ of an edge set $S \subseteq E$ such that G – S is either disconnected or trivial. Similarly, the edge– connectivity $\lambda(G)$ of a digraph G is the least cardinality |S| of an arc set S such that G – S is no longer strongly connected or is trivial. Such a set S is called a minimum edge–cut (or arc–cut in case of a digraph). Note that when G is not a trivial graph, we can define $\lambda(G)$ in terms of $\lambda(v, w)$ as follows: If G is a graph then $\lambda(G) = \min\{ \lambda(v, w) \mid$ unordered pair v , w in G } (1) In case of a digraph, we have $\lambda(G) = \min\{ \lambda(v, w) \mid$ ordered pair v , w in G } (2) The correctness of the above equalities should be clear; after all, removing a least number of edges to disconnect a graph G, for example, would in fact destroy all paths between at least a pair of vertices, and vice versa. Given the above definitions, one can compute λ of a graph (or a digraph) by knowing how to compute $\lambda(v, w)$ for arbitrary v and w. It turns out that $\lambda(v, w)$ can be computed by solving a max–flow problem in a particular network.

Algorithm 6. Input: A connected non–trivial graph G = (V,E). Output: Value of $\lambda(G)$. 1. Select a dominating set D of G. 2. Select an arbitrary vertex $v \in D$, and let $X = D – \{v\}$. 3. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in X$. 4. Assign $c \leftarrow \min\{ \lambda(v, w) \mid w \in X \}$. 5. Assign $\lambda(G) \leftarrow \min\{ c , \delta(G) \}$. Stop. Clearly this algorithm determines $\lambda(G)$

correctly. Furthermore, the smaller the set D, the fewer calls to Algorithm 1. While finding a smallest dominating set is NP– hard, finding some dominating set is easy. The next algorithm provides a way of for generating a "small" dominating set. Recall that for graph G = (V,E), we define the neighbourhood set N(X) of a vertex set $X \subseteq V$ as: $N(X) = \{ v \in V - X \mid v \text{ is adjacent in G to some vertex in X} \}$.

Algorithm 7: Input: A connected non–trivial graph G = (V,E). Output: A dominating set D 1. Select a vertex $v \in V(D)$, and let $D = \{v\}$. 2. If $V - (D \cup N(D)) = \varnothing$, then Stop. 3. Select a vertex $w \in V - (D \cup N(D))$, and assign $D \leftarrow D \cup \{w\}$. Go to Step 2. By using a dominating set D as produced by this algorithm, and amortizing the cost of computing $\lambda(v, w)$ for the vertices in D, Matula was able to bring down the overall complexity of computing $\lambda(G)$ to O(nm). His algorithm is the fastest known algorithm for determining $\lambda(G)$.

5. Полносвязный граф с n вершин (причём известно, что $n \vdots 2$) разбивается на 2 равные части таким образом, что после разбиения мы имеем 2 подмножества вершин по $\frac{n}{2}$. Пусть граф G = (V,E). Тогда после разбиения имеем $G_1=(V_1,E_1)$ и $G_2=(V_2,E_2)$ такие, что $(V_1 \cup V_2) = \varnothing$ и $(E_1 \cup E_2) = \varnothing$. То есть, ребра, которые соединяют эти два подграфа, будут удалены. Так как граф полносвязный, то можно не беспокоиться о том, что G_1 и G_2 потеряют связность. Поэтому задача из задача о теории графов переходит в задачу на комбинаторику, целью которой будет найти количество способов разделить множество вершин на 2 равные по количеству части. Это можно сделать через сочетания (не через размещения, так как порядок вершин в данном случае не важен). Берется число способов отобрать из n элементов $\frac{n}{2}$ элементов. Число сочетаний из n по $\frac{n}{2}$ равно: $C_n^{\frac{n}{2}} = \frac{n!}{(\frac{n}{2})!\left(n-\frac{n}{2}\right)!} = \frac{n!}{2(\frac{n}{2})!}$. Правда, необходимо отметить одну особенность: для $\forall$ полученного сочетания, состоящего из элементов множества V (назовём F_1, $F_1 \subset V$) всегда найдется такое сочетание, состоящее из элементов множества F_2 ($F_2 \subset V$), такие,

что ($V_1 \cup V_2$) = V, то есть, найдется дополнение множества V_1 до множества V. Данная ситуация не учитывается полученной формулой, поэтому количество способов будет вдвое меньше, то есть $\frac{C_n^{\frac{n}{2}}}{2} = \frac{n!}{2(\frac{n}{2})!\left(n-\frac{n}{2}\right)!} = \frac{n!}{4(\frac{n}{2})!}$

6. This is the funniest task. I read many articles (tried to do it) so I want to share with you something I found.

NP-hardness (non-deterministic polynomial-time hardness), in computational complexity theory, is the defining property of a class of problems that are, informally, "at least as hard as the hardest problems in NP". A simple example of an NP-hard problem is the subset sum problem.

A more precise specification is: a problem *H* is NP-hard when every problem *L* in NP can be reduced in polynomial time to *H*; that is, assuming a solution for *H* takes 1 unit time, we can use *H*'s solution to solve *L* in polynomial time. As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered difficult.

Algorithm complexity means how "close" one solution is to another is to determine its standard error. The less the better. In 2000 one algorithm with $O(\sqrt{n}\log(n))$ appeared. Fundamentally new approach was in merging vertices.

(Extended Abstract)*

Uriel Feige Robert Krauthgamer Kobbi Nissim
Deptartment of Computer Science and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel
{feige,robi,kobbi}@wisdom.weizmann.ac.il

February 22, 2000

Abstract

A bisection of a graph with n vertices is a partition of its vertices into two sets, each of size $n/2$. The bisection size is the number of edges connecting the two sets. Finding the bisection of minimum size is NP-hard. We present an algorithm that finds a bisection that is within $O(\sqrt{n}\log n)$ of optimal. No sublinear approximation ratio for bisection was previously known.

1.3 Our results

We present three approximation algorithms for bisection.

Theorem 1 *The minimum bisection problem can be approximated within the following ratios:*

1. $O(\sqrt{\frac{m}{b}}\log n)$
2. $O(n^{2/3}(\log n)^{4/3}/b^{1/3})$
3. $O(\sqrt{n}\log n)$

The third of these ratios gives the best approximation guarantee, unless either $b > m/n$ (in which case the first ratio is better) or $b > \sqrt{n}\log n$ (in which case the second ratio is better).

We use as building blocks approximation algorithms for the problem of cutting away exactly j vertices in a graph, i.e. finding a set of vertices $S \subset V$ of cardinality $|S| = j$ with the minimum number of edges connecting S to its complement $V \setminus S$. The following approximation guarantee that we obtain for this problem is particularly useful when j is small (e.g. when $j \ll \sqrt{n}$), and may be of independent interest.

We use a different procedure for merging vertices (this is done in algorithm SV in Section 3). In principle, this procedure can be applied to reduce the number of edges to nb. This fact can serve as intuition why such a procedure can potentially lead to an $O(\sqrt{n}\log n)$ approximation algorithm. However, we arrive at an $O(\sqrt{n}\log n)$ approximation in an indirect way, without ever reducing the number of edges in the graph to $O(nb)$. First we cut the graph into parts using min-ratio cuts. We then use the procedure for merging vertices in order to cut j vertices from a part by using a number of edges that is at most a factor of j more than optimal (this is a slight generalization of a result of [12]). Then the parts are combined into two roughly equal parts, using a fairly elaborate dynamic programming. Finally, there is a balancing stage in which some vertices switch sides to get an exact bisection.

When bounding the approximation ratio of our algorithm, we count every edge that was cut by one of the min-ratio cuts, even though some of these edges do not end up in the bisection found by the algorithm. Likewise, we overcount the number of edges added to the bisection during the balancing stage. The approximation ratio acheived is the ratio of the total number of edges ever counted by our analysis, divided by b.

3 Cutting a small number of vertices

Let G be a graph on n vertices and m edges. For $j \leq n/2$, let b_j be the size of the minimum cut that cuts away exactly j vertices from G. Of course, $b_{n/2}$ is the size of the minimum bisection. We present two algorithms that find a $(j, n-j)$ cut whose size is small (relative to the optimum value b_j). These algorithms are important building blocks in our approximation algorithms for the bisection.

Algorithm Greedy: Greedily pick the j vertices of smallest degree.

Proposition 4 *The number of edges cut by algorithm Greedy is at most* $\min\{2jm/n\,,\, b_j + 2\binom{j}{2}\}$.

Proof: The average degree in G is $2m/n$, and hence there exists a set of j vertices (e.g. a random one) whose degrees sum up to at most $2jm/n$. The sum of degrees of the j vertices in the optimal cut is at most $b_j + 2\binom{j}{2}$.

This upper bounds the sum of degrees of the j vertices of smallest degree, and hence the size of the cut produced. □

Our second algorithm is a modification of an approximation algorithm proposed for bisection in [12]. We assume that b_j is known to the algorithm.

Algorithm SV: Compute for all pairs of vertices u, v the size of the minimum cut between u and v. Merge (transitively) those pairs of vertices for which the minimum cut is greater than b_j. In the resulting graph, keep parallel edges but remove self loops. The resulting graph G' is composed of clusters, where a cluster is a set of vertices merged together. Each cluster has a weight equal to the number of vertices from which it is composed, and degree equal to the number of edges leaving the cluster.

Use dynamic programming to select a set of clusters whose total weight is exactly j and for which the sum of cluster degrees is smallest. This defines a cut of j vertices in the original graph.

4.3 Approximation within $O(\sqrt{n}\log n)$

In this section we wish to have an algorithm similar to that of Section 4.2, but to use the bound $2jb_j$ (from Lemma 5) on the number of edges cut in the balancing stage. We would like to argue that we can assume that $b_j \leq b$. An argument of this type was used in Section 4.2. However, now we will need to refine the argument. This will result in a different balancing stage, in which we do not only move vertices from the large side to the small side, but also move vertices from the small side to the large side. This becomes necessary because we move from side to side whole clusters, rather than individual vertices, and cluster sizes in the large side may not add up to give the exact number of vertices that need to be moved to create a bisection.

Cutting stage. Following Section 2, cut the graph into parts $S_1, \ldots, S_l$, using the approximation algorithm for min-ratio cut. As we have seen in Proposition 2, the total number of minority vertices in all parts is at most k. The number of edges cut in this stage is $O(b(n/k)(\log n)^2)$.

Marking stage. For each part S_i and every $0 \leq j \leq k$ create a version S_i^j in which exactly j vertices in S_i are *marked*. The j vertices are chosen using algorithm SV and satisfy Lemma 5 with $b_j = b$, namely, that the number of edges cut is at most $2jb$. (In fact, with some extra work we can find a cut of size at most $2jb_j$, where b_j is measured relative to S_i. Details omitted.) Note that for exactly one version, the number of vertices marked is exactly equal to the number of minority vertices in S_i. We call this version the *true* version.

Combining stage. Use dynamic programming to create two sets A and B with the following properties. From each part exactly one version is selected and put in either A or B. The number of vertices in A is n' where $n/2 - k \leq n' \leq n/2$, and the number of vertices in B is $n - n'$. The number of marked vertices in A is m_A and in B is m_B, satisfying $n' + m_B - m_A = n/2$ and $m_A + m_B \leq k$. The sum of the cut sizes of the marked vertices (in their corresponding parts) is at most $2bk$. Such a selection exists, by Proposition 11. It can be found by dynamic programming because after each part is considered, the parameters to remember are the current cardinalities of sets A and B, the number of marked vertices in each set, and sum of their cut sizes (only the minimum needs to be retained here), and each parameter ranges over polynomially many values.

Then in 2008 Harald Racke offered a new algorithm with O(log(n)) approximation. Unfortunately, I can’t google information so I didn’t find anything about it. But I found an article which was published in 2018.

Interchanging distance and capacity in probabilistic mappings

Reid Andersen* Uriel Feige†

October 4, 2018

Abstract

Harald Räcke [STOC 2008] described a new method to obtain hierarchical decompositions of networks in a way that minimizes the congestion. Räcke's approach is based on an equivalence that he discovered between minimizing congestion and minimizing stretch (in a certain setting). Here we present Räcke's equivalence in an abstract setting that is more general than the one described in Räcke's work, and clarifies the power of Räcke's result. In addition, we present a related (but different) equivalence that was developed by Yuval Emek [ESA 2009] and is only known to apply to planar graphs.

Räcke describes several applications to his results, with oblivious routing being a prominent example. Here we concentrate only on one of the applications, that of min-bisection that served as our motivating example.

Let G be a connected graph on an even number n of vertices, in which edges have nonnegative capacities. One wishes to find a bisection of minimum width (total capacity of edges with endpoints in different sides of the bipartization). We present here a polynomial time algorithm with approximation ratio $\tilde{O}(\log n)$.

Consider an arbitrary spanning tree T of G. Every edge $e = (i, j)$ of T partitions the vertices of G into two sets that we call T_i and T_j. Define the load $load_T(e)$ of edge e to be the sum of capacities of edges of G with one endpoint in T_i and the other endpoint in T_j. (This is consistent with Section 3.1.)

Consider now an arbitrary bipartization B of G. Let $E_T(B)$ be the set of edges of T that have endpoints in different sides of the bipartization. Then the width of the bipartization is at most $\sum_{e \in E_T(B)} load_T(e)$. (The load terms count every edge of G cut by the bipartition at least once and perhaps multiple times, and possibly also count edges of G not in the bipartization.) This is the *domination* property that we were referring to in Section 2.

By Theorem 1, whose proof is summarized in Section 5, one can find in polynomial time a distribution over spanning trees of G such that for every edge of G, its expected congestion (over choice of random spanning tree) is at most $\delta = \tilde{O}(\log n)$.

Consider an optimal bisection in G, and let b denote its width. For each edge cut by the bisection, its expected congestion over the probabilistic mapping into spanning trees is at most δ. Summing over all edges cut by the bisection and taking a weighted average over all spanning trees in the probabilistic mapping, we obtain that at least in one such tree T, the width of this bipartization (with respect to the load in that tree) is at most δb.

The above discussion gives the following algorithm for finding a bisection of small width in a graph G whose minimum bisection has width b.

1. Find a probabilistic mapping into spanning trees with congestion at most δ. (By the discussion above this step takes polynomial time, and δ can be taken to be $\tilde{O}(\log n)$. Furthermore, the set of spanning trees in the support of the probabilistic mapping has size polynomial in n.)

2. In each spanning tree, find an optimal bisection (with respect to the load) using dynamic programming. This takes polynomial time. Moreover, by the discussion above, in at least one tree the bisection found will have width at most δb.

3. Of all the bisections found (one per spanning tree), take the one that in G has smallest width. By the domination property, its width is at most δb.

The approximation ratio that we presented above for min-bisection is $\delta = \tilde{O}(\log n)$, rather than $O(\log n)$ as was done by Räcke. To get the $O(\log n)$ approximation, instead of probabilistic mappings into spanning trees one simply uses probabilistic mappings into (arbitrary but dominating) trees. Then one can plug in the bounds of [9] rather than the somewhat weaker bounds of [1] and obtain the desired approximation ratio. Details omitted.

The main aspects of this algorithm are zero sum games, superpolynomially many column strategies, superpolynomially many row strategies, weaker oracle models and faster algorithms; and implementation for probabilistic mappings. Õ(log(n)) is better than O(log(n)) and Uriel Feige continued his work =)

7. Используя возможности вывода графов в networkx, а также matplotlib.pyplot, я вывел граф из задания на связность (оно же первое на странице).

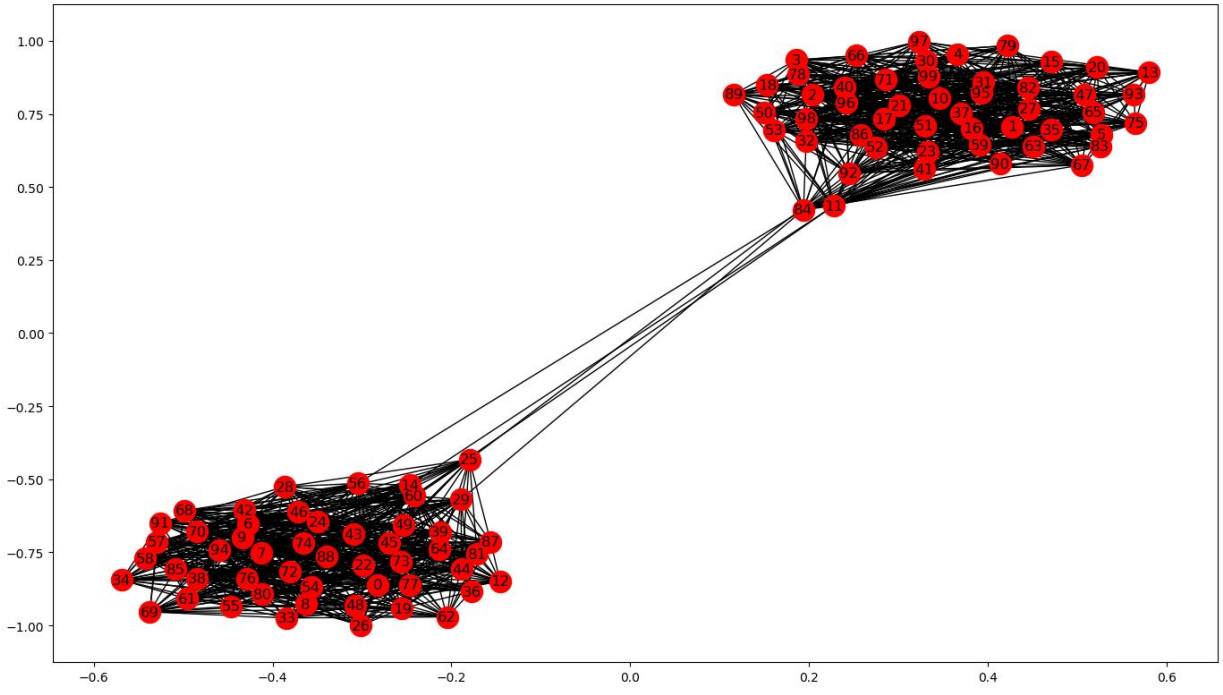
Минимальная ширина бисекции для сети

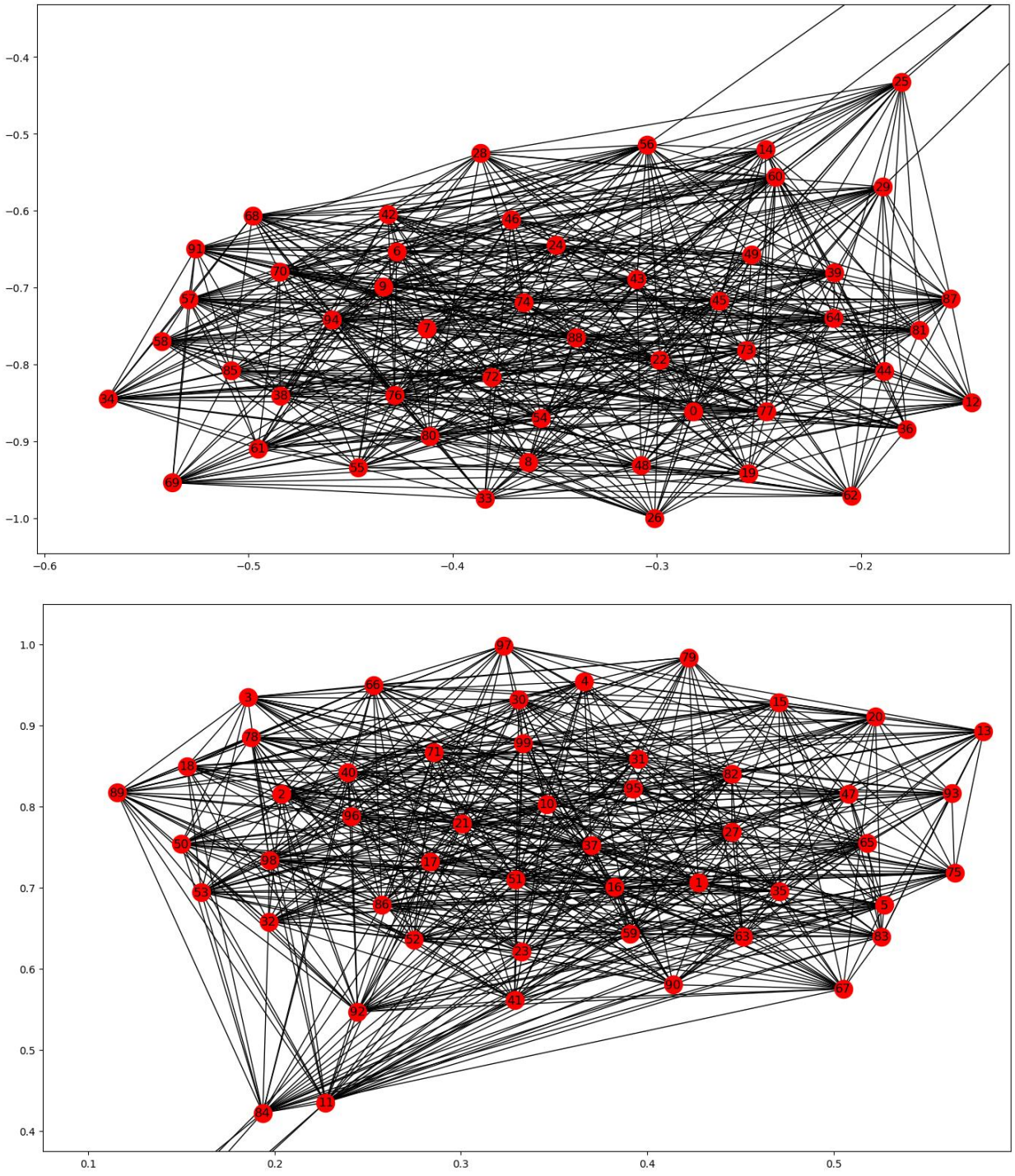
0/1 point (graded)

[Опциональное задание]

Отметьте правильные утверждения о минимальной ширине бисекция сети из первого задания на этой странице:

Получил следующее:





Ручками пересчитав количество вершин в первом и во втором подмножестве, а затем, вводя имена-номера вершин и проверяя длины множеств, я убедился, что их длины равны и на 1-ой картинке отчётливо приведена бисекция графа, причем минимальная. Потому что густое переплетение связей как в первом, так и во втором подграфе, дает понять, что надо удалять достаточно много связей, чтобы сделать другую бисекцию. А по 1-ой картинке ширина нашей бисекции равна 5. Таким образом, связность

получилась равной минимальной ширине бисекции. Теперь порассуждаем о категории общих вопросов. Логично, что если в графе некая вершина N будет иметь лишь 1 связь (т.е. 1 инцидентное ребро или, можно сказать, степень вершины равна 1), то, удалив эту самую связь, мы получим связность графа равную 1. Но при этом совсем необязательно, что это будет минимальной бисекцией, потому что минимальная бисекция графа это фактически разрез максимальной мощности, тогда логично, что в общем случае ширина минимальной бисекции графа больше или равна (>=) связности графа. Ну а 3-я категория про минимальную/максимальную связность узла сети (вершины графа) – « это троллинг» © А.А.Дюмин

8. Задание:

Ура! До сих пор Вы отлично справлялись с изучением курса "Сети и телекоммуникации"!

Заметив Ваш недюжинный талант, заведующий кафедрой "Компьютерные системы и технологии" поручает Вам спроектировать сеть для разрабатываемого на кафедре суперкомпьютера МИФИ-2018.

Исходя из потребностей приложений, которые будут работать на данном суперкомпьютере, Вы определили максимально допустимую задержку передачи данных между узлами суперкомпьютера, которая составила **1.52772683695e-06** секунд.

При этом в суперкомпьютере будет использоваться экспериментальная сетевая инфраструктура с максимально допустимым размером кадра в **2059** октет и размером поля адреса узла в **7** октет. Пропускная способность данной сети составляет **10** Гбит/сек. Коммутаторы должны быть объединены по топологии **fat-tree** с коэффициентом переподписки **1:1**. Адрес узла получателя находится в самом начале пакета. Преамбулы у пакета нет.

Вам доступны коммутаторы двух фирм **Bisco** и **Crocade**. Коммутаторы фирмы **Bisco** работают по технологии **store-and-forward** и содержат максимум **360** порта, а коммутаторы фирмы **Crocade** - по технологии **cut-through** и содержат максимум **38** порта.

Вам необходимо рассчитать, какое максимальное количество узлов может быть в проектируемом суперкомпьютере, если Вы выберете оборудование одной из указанных выше фирм.

При решении задачи учитывайте только задержку передачи данных. Другие задержки, в т.ч. задержку распространения сигнала, не учитывайте.

Решение. Пропускную способность пересчитаем в октет/с. 10 Гбит/с = 1.25*10**9 октет/с.

Производим расчет для оборудования фирмы Bisco. Оно работает по технологии store-and-forward, по которой весь кадр буферизуется и проходит дальше. Задержка через 1 коммутатор:

```
In [2]: 2059/(1.25*10**9)
Out[2]: 1.6472e-06
```

Получается, что с этим оборудованием даже через один коммутатор задержка больше допустимой)))

Уровней будет 0.

How many nodes could you have in a network?

If edge switches have *Pe* ports, and core switches have *Pc* ports, then the largest two-level fat-tree that can be built will connect $Nmax=Pe*Pc/2$ servers (smaller trees can also be built if necessary). Suppose we use one of the biggest currently produced commodity switch with $P=648$ ports, and employ the same switch model on both edge and core levels ($Pe=Pc=P$), then $Nmax=P*P/2=648*648/2=209,952$. This is more than enough for many supercomputers! If even more servers need to be connected, a three-level topology can be built.

Exercise. How many nodes will have the largest two-level fat-tree that you can build using 36-port switches? Spoiler: ( )

For every additional level that you add to your network, you multiply the number of nodes by $P/2$. Therefore, a three-level network can connect $Nmax=P*(P/2)*(P/2)=2*(P/2)^3$ nodes, and for L layers $Nmax=2*(P/2)^L$ *nodes.*

However, many cluster supercomputers only have on the order of several thousands compute nodes, and that is where variety comes into effect: you could use many small switches for your design, or a few large ones instead. There are many possible variants that would differ in their cost, possibility for future expansion, volume of equipment (how much space it takes in racks), power consumption and other important characteristics.

In fact, there are more variants that human designers tend to consider. Hence, it is beneficial to use a tool that automates the design process of fat-trees.

А узлов соответственно этой статье $N_{max} = 2(\frac{P}{2})^L = 2*180^0=2$

Теперь будем считать для оборудования типа Crocade. Оно работает по технологии Cut-And-Through. Это значит, что только часть с полем адреса узла (так как преамбулы нет) будет проходить через коммутатор (нужна для подсчета задержки).

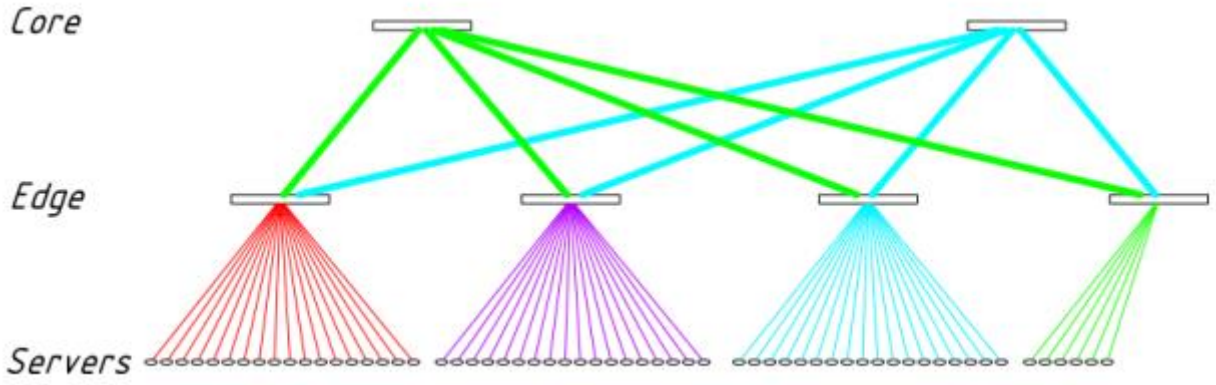
Посчитаем задержку прохождения через 1 коммутатор:

```
In [3]: 7/(1.25*10**9)
Out[3]: 5.6e-09
```

Теперь посчитаем количество уровней в сети. Фактически это будет «сколько раз 5.6*10**(-9) уложится в 1.5*10**(-6)», так как мы учитываем задержку прохождения ТОЛЬКО через коммутаторы.

```
In [4]: 1.52772683695e-06/(7/(1.25*10**9))
Out[4]: 272.80836374107145
```

Получили вот такой результат. Округляем его в меньшую сторону, чтобы уложиться в заданную задержку. То есть количество уровней в сети будет 272. Теперь рассчитаем количество узлов. Мы имеем право воспользоваться формулой $N_{max} = 2(\frac{P}{2})^L$, так как коэффициент переподписки сети 1:1, то есть количество портов, идущих «вниз», равно количеству портов, идущих «вверх»



Blocking

We noted above the property of fat-tree networks in their primary meaning: on each intermediate level, the number of links that go to the upper level (in two-level networks, this is from the edge level to the core level) is equal to the number of links going to the lower level (here, from the edge level down to nodes). On this intermediate switch, the number of uplinks and downlinks are in proportion of 1:1. Such networks are called non-blocking. You can arbitrarily divide nodes into pairs, and if all pairs start communicating, still every pair of nodes will be able to communicate at full link bandwidth.

$N_{max} = 2(\frac{38}{2})^{272}=2 * 19^{272} =$
1324370367163477867146013121742510502528493079916844359564
3300383658976470068440364750283357283992730471846733769791
6222525742507042852590297514930839358511633062363358758744
2922363823139733784031040255173430509095380713084179440532
5309007181863846375355119621048194047879556968620487077797
1989685161182969437691084879584719591978218877623702403392
2

Ну как бэ… Много)))