CS 466/666 Algorithm Design and Analysis, Spring 2019. Waterloo.

Lecture 3 : Graph algorithms

We will study some interesting randomized algorithms for graph problems, including longest path, minimum cuts, and minimum spanning trees. We will see that elementary probabilistic ideas are already very powerful.

Color coding [1]

Color coding is a cool technique to design "fixed parameter tractable" (FPT) algorithms.

Consider the k-path problem where the objective is to find a path of length at least k.

The brute force algorithm takes $O(n^k)$ time.

Since the longest path problem is NP-complete, we do not expect that this problem can be solved in polynomial time for every k, but we could hope to solve the problem when k is small.

We say a problem is fixed parameter tractable if there is an algorithm with running time $O(f(k) \cdot n^c)$, where c is an absolute constant, i.e. the parameter k does not appear in the exponent of n.

It is not clear how to design such an algorithm, and the color coding technique is quite creative.

Algorithm is simple to describe.

- Color each vertex independently with a uniform random color from $\{1, 2, \ldots, k\}$.
- Search for a "colorful" k-path (a path of length k with k distinct colors).
  If such a path can be found, return the k-path found; otherwise return "NO".

Dynamic programming : The clever observation is that it is easier to solve the colorful k-path problem.

We sketch the dynamic programming algorithm below (anything non-probabilistic is not the main focus of this course.)

Suppose we fix the starting vertex s (at most n possibilities).

For each vertex v and for each $1 \le i \le k$, we remember all possible "color subsets" of colorful path of length $i-1$, starting from s and ending at v.

Notice that since the path is colorful, it is guaranteed to have no repeated vertices.

For a particular i, we need to store at most $\binom{k}{i} \le 2^k$ subsets at a vertex v.

Let us denote all these subsets by $DP(v,i)$. It is obvious to compute $DP(v,1)$.

Once we have computed $DP(v,i)$ for all v, we can compute $DP(v,i+1)$ efficiently as follows.

For an edge uv, if a color subset S is in $DP(u,i)$ and $color(v)$ is not in S, then we add $S+\{color(v)\}$ to $DP(v,i+1)$. So for an edge it takes at most $O(2^k)$ time.

Hence, we can compute $DP(v,i+1)$ for all v in time $O(2^k \cdot m)$ time, given $DP(u,i)$ for all u.

Once we computed $DP(v,k)$ for all v, we return "YES" if $DP(v,k)$ is non-empty for some v, otherwise we return "NO". This takes time $O(k \cdot 2^k \cdot m)$.

Finally, we try all starting vertices, and the total runtime is $O(k \cdot 2^k \cdot m \cdot n)$.

Success probability: What is the probability that the algorithm succeeds?

Suppose there is a path P of length k in the graph.

When we color the vertices of P independently and uniform randomly, the probability that P becomes colorful is $k!/k^k \geq e^{-k}$ (recall that $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$).

Therefore, by repeating the algorithm $O(e^k)$ times, the failure probability is at most $(1-e^{-k})^{O(e^k)} \leq 10^{-10}$.

That is, with probability $\geq 1-10^{-10}$, there is one execution with P being colorful, and the dynamic programming algorithm will find it, if such a P exists.

The overall complexity is $O(k \cdot (2e)^k \cdot m \cdot n)$, fixed parameter tractable.

Derandomization: It is possible to derandomize this algorithm by using almost k-wise independent variables, and it leads to a deterministic algorithm that is difficult to come up with otherwise.

Color coding and its variants (e.g. random separation) is a useful technique in designing FPT algorithms.

Randomized minimum cut [MU 1.4, MR 10.2]

Problem: In the minimum cut problem, we are given an unweighted undirected graph $G=(V,E)$, and the objective is to find a minimum subset of edges $F \subseteq E$ so that $G-F$ is disconnected.

Simple ideas: A simple observation is that a minimum cut is also a min s-t cut for some s,t.

So, one can compute min s-t cut for all s,t and take a minimum one, and this naive algorithm requires n^2 calls of maximum s-t flow (which can be solved in polynomial time).

Then, one observes that it is enough to just fix an arbitrary vertex as s and just compute min s-t cut for all possible t, and this gives an algorithm in n maxflow time.

Deterministic algorithm: For a while this global min-cut problem seems more difficult than min s-t cut.

Deterministic algorithm: For a while, this global min-cut problem seems more difficult than min s-t cut. Then, Matula (1987) gave a very nice algorithm that computes min-cut in $O(|V||E|)$ time. His new idea is to do a graph search and identify an edge that can be contracted and repeat.

Randomized algorithms: Karger came up with the idea of random contraction and improved the runtime to $\tilde{O}(|V|^2)$, which is faster than computing min s-t cut. Eventually, he gave a near-linear time $\tilde{O}(|E|)$ algorithm for the minimum cut problem, which is very interesting and surprising.

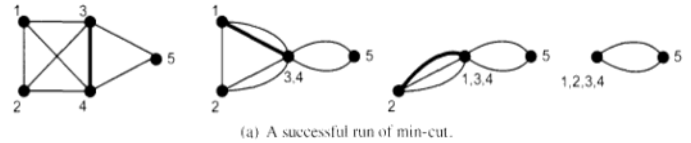
Here, the $\tilde{O}$ notation hides some polylog factor in the runtime. e.g. $O(n^2 \log^3 n) = \tilde{O}(n^2)$.

We will present an $O(|V|^4)$-time algorithm and mention how it can be improved to $\tilde{O}(|V|^2)$.

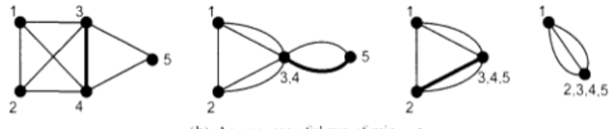
Karger's algorithm is very simple.

```
While there are more than two vertices in the graph
    Pick a random edge and contract the two endpoints
Output the edges between the remaining vertices.
```

By "contracting" an edge, we mean to identify the two endpoints as a single vertex, putting both on the same side. See the picture below from the book by Mitzenmacher and Upfal.



(a) A successful run of min-cut.



(b) An unsuccessful run of min-cut.

Observation: Each vertex in an intermediate graph is a subset of vertices in the original graph.

So, each cut in an intermediate graph corresponds to a cut in the original graph.

Hence, a min-cut in an intermediate graph is at least a min-cut in the original graph.

Theorem The probability that the algorithm outputs a minimum cut is at least $2/n(n-1)$, where n is the number of vertices in the input graph.

Proof Let F be a minimum cut and let $k = |F|$ be the number of edges in F.

where n is the number of vertices in the input graph.

Proof Let F be a minimum cut and let $k = |F|$ be the number of edges in F.

If we never contract an edge in F until the algorithm ends, then the algorithm succeeds.

What is the probability that an edge in F is contracted in the i-th iteration?

By the observation, the min-cut value in the i-th iteration is at least k.

Note that it implies that every vertex is of degree at least k, as otherwise we can disconnect a single vertex from the graph by removing less than k edges.

Therefore, the number of edges in the i-th iteration is at least $(n-i+1)\cdot k/2$.

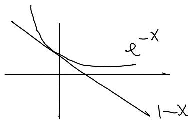
Since we pick a random edge to contract, the probability that we pick an edge in F is at most $k/\left((n-i+1)k/2\right) = \frac{2}{n-i+1}$.

So, the probability that F survives until the end is at least

$$\left(1-\frac{2}{n}\right)\cdot\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right) = \frac{\cancel{n-2}}{n}\cdot\frac{\cancel{n-3}}{n-1}\cdot\frac{\cancel{n-4}}{\cancel{n-2}}\cdots\frac{2}{\cancel{4}}\cdot\frac{1}{\cancel{3}} = \frac{2}{n(n-1)}. \quad \square$$

Improving success probability A simple way is to repeat the whole procedure many times.

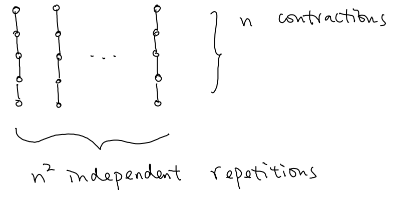
Suppose we repeat for t times. Then the failure probability is at most $\left(1-\frac{2}{n(n-1)}\right)^t$

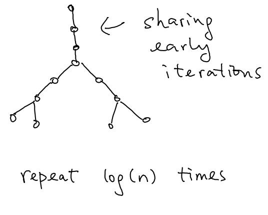
Recall that $1-x \le e^{-x}$, 

which can be derived from Taylor expansion (good to review some basic calculus, e.g. Stirling's approx)

Therefore, $\left(1-\frac{2}{n(n-1)}\right)^t \le e^{-\frac{2t}{n(n-1)}}$. So, if we set $k = 10n(n-1)$, then failure probability $\le e^{-20}$.

Running time : One execution can be implemented in $O(n^2)$ time (an exercise in data structures), and thus the total time complexity is $O(n^4)$.

Improving running time: The observation is that the failure probability is only large near the end of one executation, while it is very small in the beginning. So, the idea is that we only repeat the later iterations but not the early iterations. Pictorically, the $O(n^4)$ algorithm is like 

, while it is much better to do 

See more details in [MR 10.2]. This is called the Karger-Stein algorithm (1993).

Combinatorial Structure: An interesting corollary of the theorem is that there are at most

Combinatorial structure: An interesting corollary of the theorem is that there are at most $O(n^2)$ min-cuts in an undirected graph, because each min-cut survives with probability $\Omega(\frac{1}{n^2})$ and the events that two different min-cut survive are disjoint. This is a non-trivial statement to prove using other arguments.

minimum k-cut The algorithm can be extended to give a $n^{O(k)}$ algorithm for finding a minimum k-cut, which is nontrivial to do by a deterministic algorithm.

Notice that the minimum k-cut problem is NP-hard if k is given as an input.

Question: Can you modify the algorithm for the minimum s-t cut problem?

Open question: Can you design a fast algorithm for computing global vertex connectivity?

Remark: After we studied graph sparsification, we can discuss Karger's near linear time min-cut algorithm.

Minimum spanning trees [MR 10.3]

We will present a randomized linear time algorithm for the classical minimum spanning tree problem.

We assume without loss of generality that all edge weights are distinct; in particular there is a unique minimum spanning tree.

Boruvka's algorithm

First, we revisit a classical algorithm for MST, known as the Borůvka's algorithm.

In each iteration, for each vertex v, we add the edge vw to the solution, where vw is the edge with minimum weight among the edges incident on v.

Observe that each edge added must be in the MST (why?).

Also, it is easy to see that in each iteration we add at least $n/2$ edges to the solution.

Then we can contract the components in the current solution and repeat this procedure.

Since every iteration decreases the number of vertices by half, there are at most $\log n$ iterations, and the total running time is $O(m \log n)$.

Heavy edge and light edge with respect to a forest: This is an important definition for the algorithm.

Given a forest F (think of it as a candidate partial solution of MST), we say an edge $e=(u,v)$ is

F-heavy if u and v is connected by a (unique) path P_{uv} in F, and every edge in P_{uv} has a weight smaller than the weight of uv. Otherwise, we say the edge $e=(u,v)$ is F-light.

The point of the definition is that an F-heavy edge does not belong to any MST, no matter what F is (as otherwise we can add some edge in P_{uv} to create a cycle with the edge uv, and we can swap these two edges to form another spanning tree with lower weight). So, if we can find some F for which an edge e is F-heavy, we can safely delete the edge e from the graph.

On the other hand, if an edge $e \notin F$ is F-light, then it means that e can be used to improve the current partial solution F, by either decreasing the weight of the forest or decreasing the number of connected components of F.

MST verification algorithm : The main tool behind the linear time algorithm is a deterministic linear time algorithm to verify whether a tree is an MST. Moreover, it can report all the F-heavy edges of the proposed solution F.

Theorem [Dixon-Rauch-Tarjan ; Komlós ; King] Given a weighted graph $G=(V,E)$ and a forest F, all F-heavy edges can be identified in $O(|V|+|E|)$ time.

This is a strong result. We will not prove it.

We will just use it as a black box to remove all F-heavy edges to "sparsify" the graph.

Random sampling : The main idea in the randomized algorithm is to use random sampling.

Suppose we sample each edge with probability p (i.e. keep the edge with prob p, and delete the edge with prob $1-p$) and consider the resulting subgraph $G(p)$ of G.

The expected number of edges in $G(p)$ is $p \cdot |E(G)|$.

We find a minimum spanning forest F of $G(p)$ (as $G(p)$ may not be connected), which can be done faster than in G as $G(p)$ has fewer edges.

Then, we run the MST verification algorithm with F as the input, and remove all F-heavy edges in G to obtain G', and then we find an MST in G' which hopefully has much fewer edges than G.

The hope is that there is a choice of p so that both $G(p)$ an G' have very few edges.

Analysis : The key of the analysis is to analyze how many F-light edges are in G, as these are exactly the edges remained in G'.

Lemma The expected number of F-light edges in G is at most n/p.

Proof In the analysis, by the principle of deferred decision (that the random choices are revealed when we need them), we can assume that the edges $e_1, e_2, \ldots, e_m$ are sorted by increasing weight, and we construct $G(p)$ by flipping coins in this order.

Note that this is only for the analysis, and we cannot afford to sort the edges in the algorithm.

Whether an edge e_i is F-light depends only on which subset of edges in $e_1, \ldots, e_{i-1}$ appears in $G(p)$.

Let the subset of edges in $e_1, \ldots, e_{i-1}$ appearing in $G(p)$ be E_{i-1}.

An edge e_i is F-light if and only if e_i connects two connected components in E_{i-1}, regardless of whether e_i is chosen in $G(p)$ or not.

If e_i is chosen in $G(p)$, then e_i will be one of the edges in the minimum spanning forest F; if e_i is not chosen in $G(p)$, then e_i will be an F-light edge in G as it can be used to improve the forest F.

Now, if $n-1$ F-light edges are chosen in $G(p)$, then $G(p)$ will be connected and there will be no more F-light edges after that.

So, the question is how many F-light edges are "considered" before we choose $n-1$ F-light edges in $G(p)$?

For every F-light edge, it is chosen in $G(p)$ with probability p. So, the expected number of F-light edges that we "considered" is $\frac{1}{p}$ before we choose one such F-light edge in $G(p)$. This is the expected number of a geometric random variable where the success probability of each trial is p.

Therefore, before we choose $n-1$ F-light edges in $G(p)$, the expected number of F-light edges that we "considered" is at most $(n-1)/p$, and these are exactly the F-light edges in G. $\square$

First algorithm : ① Construct $G(p)$ by random sampling.

② Find a minimum spanning forest F in $G(p)$.

③ Use the MST verification algorithm to remove all F-heavy edges in G to form G'.

④ Find a minimum spanning tree in G'.

The expected number of edges in $G(p)$ is pm and the expected number of edges in G' is n/p.

The best choice of p is $p = \sqrt{\frac{n}{m}}$, so that both graphs have $\sqrt{nm}$ edges.

This gives us a nontrivial $O(m + \sqrt{nm} \log m)$ algorithm, which is linear when $m \geq n \log^2 m$.

Recursive algorithm : A golden principle in algorithm design is that when we have a good idea, we should apply it recursively, so that the good idea is used over and over again.

The following is the final algorithm.

① Reduce the graph size by 1/8 by running three iterations of Borůvka's algorithm.

② Set $p = \frac{1}{2}$ and construct $G(p)$. Find a minimum spanning forest F of $G(p)$ recursively.

③ Run the MST verification algorithm on G using F and remove all F-heavy edges in G to form G'. Find a minimum spanning tree of G' recursively.

Time complexity: Let $T(m,n)$ be the expected running time on a graph with m edges and n vertices.

Then we have the recurrence relation $T(m,n) \leq \underbrace{T(\frac{m}{2}, \frac{n}{8})}_{\text{MSF in } G(\frac{1}{2})} + \underbrace{T(\frac{n}{4}, \frac{n}{8})}_{\text{MST in } G' \text{ which has } (n/8)/\frac{1}{2} = \frac{n}{4} \text{ edges}} + c(m+n)$.

It is easy to check that $T(m,n) \leq 2c(m+n)$ is a solution.

This linear time algorithm is by Karger, Klein and Tarjan (1995).

References [1] Alon, Yuster, Zwick. Color-coding.