CSC 5450 Randomness and Computation

Week 11 : Graph Algorithms

Plan : ① color coding

② minimum spanning tree

③ all pairs shortest paths

④ graph sparsification and applications

Color Coding [1]

This is a cool technique to design fixed parameter algorithms.

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

The brute-force algorithm takes $\theta(n^k)$ time.

We show how to use color coding to design a FPT algorithm for the k-path problem.

The algorithm is simple : randomly color each vertex by one of the colors in $\{1,2,\ldots,k\}$,
and then find a colorful path of length k in the resulting graph.

The clever observation is that it is much easier to solve the colorful path problem,
by dynamic programming.

Suppose we fix the starting vertex s.

For each vertex v, we remember all possible "color subsets" of colorful paths of length i,
starting at s and ending at v.

For a particular i, we store at most $\binom{k}{i}$ subsets on a vertex v, denote it by DP(v,i).

It is easy to compute all the states for i=1.

If we have the states for i, we can compute the states for i+1 efficiently.

For an edge uv, if a 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 $O(\binom{k}{i}) = O(2^k)$ time.

So all the states for i+1 can be computed in $O(2^k \cdot m)$ time, given the states for i.

Once we compute DP(v,k) for all v, we return YES if DP(v,k) is nonempty for some v,
otherwise return NO. It takes $O(k \cdot 2^k \cdot m)$ time.

By trying all starting vertex, it takes $O(k \cdot 2^k \cdot m \cdot n)$ time.

The probability that a k-path is colorful is $k!/k^k \geq e^{-k}$.

The probability that a k-path is colorful is $k!/k^k \geq e^{-k}$.

Therefore, by repeating the procedure for $O(e^k \log n)$ times, there is one execution with a colorful k-path with high probability, if a k-path exists.

So the overall complexity is $O(k \cdot (2e)^k \cdot m \cdot n \cdot \log n)$, considerably faster than brute force.

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 is a useful technique, and Prof. Cai has a closely related "random separation" technique for designing FPT algorithms (see notes of CSC 5320 if you are interested).

Minimum Spanning Tree [MR 10.3]

We will present a randomized linear time algorithm for MST, while there is a $O(m \log^* m)$ deterministic algorithm.

We assume without loss of generality that the edge weights are distinct.

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.

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 MST verification

Given a forest F (think of it as the current partial solution), we say an edge $e = (u,v)$ is <u>F-heavy</u> if u and v is connected in F by a path P_{uv} and uv is heavier than every edge in P_{uv}; otherwise, an edge is <u>F-light</u>.

The point of this definition is that an F-heavy edge must not belong to the MST. (Otherwise, some edge of P_{uv} will create a cycle with uv, and we can swap this edge with uv.) On the other hand, an F-light edge can be used to improve the current forest F.

There is a deterministic algorithm that verifies if a given spanning tree is the MST in linear time. And in fact it identifies all F-heavy edges.

Theorem Given a graph G and a forest F, all F-heavy edges can be identified in $O(n+m)$ time.

This is a strong result and we will use it (but not prove it) to remove all F-heavy edges.

Random sampling

The idea is simple. Consider $G(p)$ where each edge appears with probability p. Find a minimum spanning forest F in $G(p)$ (since $G(p)$ may not be connected), and use the MST verification algorithm to remove all F-heavy edges. Then we can "sparsify" the graph and find the MST in the remaining graph.

So the key is to analyse how many F-light edges in $G(p)$.

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

Proof By the principle of deferred decisions, we can assume that the edges are sorted in order of increasing weights, $e_1, e_2, \ldots, e_m$, and we construct $G(p)$ by flipping coins in this order.

(Note that this is only for the analysis; we don't need to sort the edges in the algorithm.)

An edge e_i is F-light when e_i is considered if and only if e_i connects two components in the beginning of step i, and it will remain to be F-light at the end.

We may not add e_i to the forest because e_i may not appear in $G(p)$, but it is added to the forest with probability p.

So how many F-light edges do we see until one is added to the forest?

This is just the expected value of a geometric variable with success probability p, and thus is $\frac{1}{p}$.

The process must stop when we have added $n-1$ edges, and by that time the expected number of F-light edges we have is at most n/p. ■

Linear time algorithm

Using random sampling, one can do the following:

- first construct $G(p)$, and find a minimum spanning forest F in $G(p)$
- then remove all heavy edges in G to form G', return the MST in G'.

In order for the total running time to be small, we need both $G(p)$ and G' to have few edges.

Heuristically, if we set $p = \sqrt{\frac{n}{m}}$, then both graphs would have $\sqrt{nm}$ edges, but it is not quite enough to give us an $O(m+n)$ time algorithm yet.

A golden principle in algorithm design is that when we have a good idea, we should recurse on it, and this is what we do to make it work.

Instead of solving the MST problem directly, we recursively apply the same procedure

- reduce the graph size by 1/8 by running three iterations of Borůvka's algorithm
- construct $G_1(p)$ from G_1, find minimum spanning forest F in $G_1(p)$ recursively
- remove all F-heavy edges from G_1 to form G_1', find MST in G_1' recursively.

Set $p=1/2$. Let $T(n,m)$ be the expected running time of the algorithm.

Then $T(n,m) \le T(n/8, m/2) + T(n/8, n/4) + c(m+n)$ for a constant c.

It is easy to check that $2c(m+n)$ is a solution, and we are done.

It can be proved that the running time is $O(m+n)$ with high probability, and the worst case time is $O(m \log n)$.

Two questions remain: ① deterministic $O(m+n)$ algorithm for MST

② simple (randomized) algorithm for MST verification

All Pairs Shortest Paths [MR 10.1]

Given an undirected graph $G_1=(V,E)$ where each edge is of the same length, the all-pairs shortest paths problem is to compute the shortest path between u and v for all $u,v \in V$.

This can be done easily in $O(mn)$ time, by doing n breadth first search.

We will present a $\tilde{O}(n^w)$ algorithm for the problem, where $w \approx 2.38$.

A key component is a $O(n^w)$ time algorithm for multiplying two $n \times n$ matrices.

But even given this powerful tool, we still need several clever ideas to make it work.

The proof is somewhat long, so let me give a quick overview first

① Use the matrix multiplication algorithm to compute the lengths of all shortest paths. (deterministic)

② Given two 0-1 matrices A, B, compute the witness matrix P of AB, where $P_{ij}=k$ if $A_{ik}=1$ and $B_{kj}=1$, if no such k exists then $P_{ij}=0$.

③ Use ① and ② to construct a succient data structure to store a shortest path for every pair.

Computing distances

Let A be the adjacency matrix of the graph G

Let A be the adjacency matrix of the graph G.

Consider the k-th power of A, denoted by A^k. Then A^k_{ij} is nonzero if and only if there is a path (not necessarily simple) from i to j of length k.

So, if we can compute $A, A^2, A^3, \ldots, A^n$, then we can solve the distance problem, but this is too slow.

Instead, let's try to compute $A, A^2, A^4, A^8, \ldots$ and see what can we deduce.

Given $G=(V,E)$, let $G'=(V,E')$ be the graph where $ij \in E'$ if and only if the distance between i and j is at most 2.

Given the adjacency matrix A of G, it is easy to construct the adjacency matrix of G' in one matrix multiplication time.

The base case is when G' is a complete graph, then we know the distance between i and j is is 2 if $ij \notin E(G)$; otherwise if $ij \in E(G)$ then their distance is 1.

Suppose we know the distance matrix D' for G'. Can we use it to compute the distance matrix D of G?

If so, then we can do "repeated squaring", and get the answer using $O(\log n)$ matrix multiplications.

Hence we study the relationship of D_{ij} and D'_{ij}.

It is easy to see that if $D'_{ij} = k$, then either $D_{ij} = 2k-1$ or $D_{ij} = 2k$. In other words,

- if D_{ij} is even, then $D_{ij} = 2D'_{ij}$
- if D_{ij} is odd, then $D_{ij} = 2D'_{ij} - 1$.

So, we can determine D_{ij} if we can determine the parity of D_{ij}.

For any i, j, it is easy to see that

- if k is a neighbor of i, then $D_{ij} - 1 \le D_{kj} \le D_{ij} + 1$
- there exists a neighbor k of i such that $D_{kj} = D_{ij} - 1$ (e.g. the one on a shortest path from i to j)

If $D_{ij} = 2\ell$, then $2\ell - 1 \le D_{kj} \le 2\ell + 1$ for any neighbor k of i.

This implies that $D'_{kj} \ge D'_{ij}$ for all $k \in N(i)$.

If $D_{ij} = 2\ell - 1$, then $2\ell - 2 \le D_{kj} \le 2\ell$ for any neighbor k of i, and $\exists k$ with $D_{kj} = 2\ell - 2$.

This implies that $D'_{kj} \le D'_{ij}$ and $\exists k$ with $D'_{kj} < D'_{ij}$.

Adding up the inequalities for all $k \in N(i)$ gives

- if D_{ij} is even, then $\sum_{k \in N(i)} D'_{kj} \ge d(i) \cdot D'_{ij}$

- if D_{ij} is odd, then $\sum_{k \in N(i)} D'_{kj} < d(i) \cdot D'_{ij}$

So this is a perfect test for the parity of D_{ij}.

And this can be checked efficiently (an important point), again by matrix multiplication.

Let $X = A \cdot D'$. Then $X_{ij} = \sum_{k=1}^{n} A_{ik} \cdot D'_{kj} = \sum_{k \in N(i)} D'_{kj}$.

Therefore, once we computed X in $O(n^\omega)$ time, we can check the parity of D_{ij} in $O(n^2)$ time (by checking whether $X_{ij} < d(i) \cdot D'_{ij}$), and then D can be constructed in $O(n^2)$ time.

So, the distance matrix D of G_1 can be computed in $O(n^\omega \log n)$ time.

Witnessing boolean matrix multiplication

Given two $n \times n$ boolean matrices A and B, we would like to compute a $n \times n$ matrix P such that if $(AB)_{ij} \neq 0$, then $P_{ij} = k$ for some k with $A_{ik} = 1$ and $B_{kj} = 1$; otherwise if $(AB)_{ij} = 0$, then $P_{ij} = 0$.

We say P is the witness matrix of AB and P_{ij} is the witness of $(AB)_{ij}$.

For example, if $A = B$ is the adjacency matrix of G_1, then the matrix P tells all the intermediate points of the paths of length two.

If we compute AB directly, then $(AB)_{ij}$ is the number of witnesses of $(AB)_{ij}$, but not who is the witness of $(AB)_{ij}$.

A brute-force algorithm would not work since it takes $\Omega(n^3)$ time.

An observation is that if there is a unique witness for $P_{ij} = k$, then we can compute P_{ij} easily.

The trick is to replace A by $\hat{A}$, where $\hat{A}_{ik} = k \cdot A_{ik}$, and compute $\hat{A} \cdot B$.

If there is a unique witness for P_{ij}, then $(\hat{A} \cdot B)_{ij} = \sum_k \hat{A}_{ik} \cdot B_{kj} = \sum_k k \cdot A_{ik} \cdot B_{kj} = k$.

The important idea is to guarantee a unique witness for an entry by random sampling.

Consider P_{ij}. Let w be the number of witnesses for P_{ij}.

How can we guarantee a unique witness?

The intuition is that if we sample each witness with probability $1/w$. Then the expected number of witnesses remained is one.

Of course, we don't know who are the witnesses, so we take a random set of indices $R \subseteq \{1, \ldots, n\}$

of cardinality r with $n/2 \le wr \le n$ (intuition was $r \approx n/w$).

Then one can prove formally that there is a unique witness for P_{ij} with constant probability.

Lemma 10.7: *Suppose an urn contains n balls of which w are white, and $n - w$ are black. Consider choosing r balls at random (without replacement), where $n/2 \le wr \le n$. Then*

$$\mathbf{Pr}[\textit{exactly one white ball is chosen}] \ge \frac{1}{2e}.$$

PROOF: By elementary counting, the desired probability can be bounded as follows.

$$\begin{aligned}
\frac{\binom{w}{1}\binom{n-w}{r-1}}{\binom{n}{r}} &= w\frac{r!}{(r-1)!}\frac{(n-w)!}{n!}\frac{(n-r)!}{(n-w-r+1)!} \\
&= wr\left(\prod_{i=0}^{w-1}\frac{1}{n-i}\right)\left(\prod_{j=0}^{w-2}(n-r-j)\right) \\
&= \frac{wr}{n}\left(\prod_{j=0}^{w-2}\frac{n-r-j}{n-1-j}\right) \\
&\ge \frac{wr}{n}\left(\prod_{j=0}^{w-2}\frac{n-r-j-(w-j-1)}{n-1-j-(w-j-1)}\right) \\
&= \frac{wr}{n}\left(\prod_{j=0}^{w-2}\frac{n-w-(r-1)}{n-w}\right) \\
&= \frac{wr}{n}\left(1-\frac{r-1}{n-w}\right)^{w-1} \\
&\ge \frac{1}{2}\left(1-\frac{1}{w}\right)^{w-1}
\end{aligned}$$

The last inequality follows from the observations that $wr/n \ge 1/2$ and $(r-1)/(n-w) \le 1/w$, which in turn follow from the assumption that $n/2 \le wr \le n$. Finally, applying Proposition B.3, the last expression is bounded by $1/2e$. □

(Obviously, it is a copy from Motwani-Raghavan's book.)

Let's come up with an efficient step to compute the entries with number of witnesses equal to w.

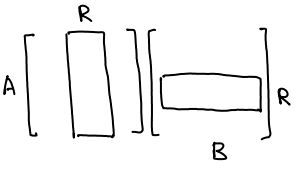
Pick any r such that $n/2 \le wr \le n$. Generate a random set of indices R with $|R| = r$.

Set $R_k = 1$ if $k \in R$, otherwise $R_k = 0$.

Let A^R be the matrix with $A^R_{ik} = k \cdot R_k \cdot A_{ik}$ and B^R with $R_k \cdot B_{kj}$

Then $(A^R \cdot B^R)_{ij} = \sum_k A^R_{ik} \cdot B^R_{kj} = \sum_k k \cdot R_k A_{ik} \cdot R_k \cdot B_{kj} = l$ if

l is the unique witness of P_{ij} in R.



After computing $A^R B^R$, we can check each entry to see if it is a witness of P_{ij} in $O(n^2)$ time.

So as long as R contains the unique witness for an entry, we can find it in $A^R B^R$.

This happens with constant probability for each entry.

So, if we construct $O(\log n)$ many R, then we can find a witness for P_{ij} if there are w witnesses for P_{ij}.

By setting $r = 1, 2, 4, 8, 16, \ldots, n$, each with $O(\log n)$ many R, then a witness for P_{ij} is

found with high probability.

The expected number of the remaining entries is small, i.e. $O(n)$ remaining entries, and we can find their witnesses in $O(n^2)$ time by brute-force algorithm.

The total running time is $O(n^\omega \log^2 n)$.

Determining shortest paths

In $\tilde{O}(n^\omega)$ time we could not return all paths, because the output size could be $\Omega(n^3)$.

Instead we will construct an implicit data structure such that any shortest path P can be constructed in $|P|$ steps. Informally, we just need to know what is the next step.

Definition. A successor matrix S for an n-vertex graph G is an $n \times n$ matrix such that S_{ij} is the index k of a neighbor of vertex i that lies on a shortest path from i to j.

Suppose we are given the distance matrix D. We can compute all S_{ij} with $D_{ij} = d$.

Let B be the $n \times n$ 0-1 matrix with $B_{ij} = 1$ if $D_{ij} = d-1$, which can be constructed in $O(n^2)$ time.

Then we can compute the witness matrix of AB to get all S_{ij} with $D_{ij} = d$.

But we could not afford to do this for all d.

The idea is that we can compute many d in parallel.

Recall that if k is a neighbor of i, then $D_{ij} - 1 \le D_{kj} \le D_{ij} + 1$ and k is an answer for S_{ij} if $D_{kj} = D_{ij} - 1$.

Thus we just need to do arithmetic mod 3 to distinguish $D_{kj} = D_{ij} - 1$, $D_{kj} = D_{ij}$ and $D_{kj} = D_{ij} + 1$.

So, any k with $A_{ik} = 1$ and $D_{kj} = D_{ij} - 1 \pmod 3$ is an answer for S_{ij}.

For $s \in \{0, 1, 2\}$, construct the $n \times n$ matrix $D^{(s)}$ with $D^{(s)}_{kj} = 1$ if $D_{kj} + 1 \equiv s \pmod 3$.

For i, j with $D_{ij} \equiv s \pmod 3$, we just need to compute the witness matrix P of $AD^{(s)}$ and set S_{ij} to be P_{ij}.

So we just need 3 matrix multiplications, and the successor matrix can be computed in $O(n^\omega)$ time.

The overall complexity of the algorithm is $O(n^\omega \log^2 n)$.

The big open question is whether there is a subcubic algorithm for all-pairs shortest paths when the edges can have arbitrary lengths.

Even better if you could solve it for directed graphs.

Graph Sparsification [2]

In week 3, we talked about graph sparsifiers. Given a graph G, a graph H is a $(1\pm\varepsilon)$-approximation of G if for every cut $\delta(S)$ for $S \subseteq V$, we have $(1-\varepsilon)w_G(\delta(S)) \le w_H(\delta(S)) \le (1+\varepsilon)w_G(\delta(S))$.

Note that H is not required to be a subgraph of G, and the weights of the edges in H could be different.

The following is the main theorem in week 3.

Theorem Let G be a graph in which each edge e has a probability of p_e being sampled. If the expected # edges of every cut in G is at least $q_\varepsilon = 3(d+2)(\ln n)/\varepsilon^2$ for some ε and d, then with probability $1-1/n^d$ every cut in G' has # edges $(1\pm\varepsilon)$ within its expectation.

This theorem can be applied to show that if the minimum cut in G is at least $c >> \ln n$, then we can construct a graph G' with $O(\frac{\log n}{c})\cdot|E(G)|$ edges while every cut is $(1\pm\varepsilon)$-approximation of G.

The limitation is that the minimum cut must be large, and if there is a small cut nothing can be said.

We present the following much stronger theorem.

Theorem For any graph G, there is a H with $O(n\log n)$ edges and is a $(1\pm\varepsilon)$-approximation of G.

To prove this, we need to do a nonuniform sampling on the edges. The idea is that edges with small connectivities are sampled with larger probabilities but a smaller weight if chosen, while edges with high connectivities are sampled with smaller probabilities but a larger weight if chosen.

Definitions - A graph is k-connected if every cut is of weight at least k.

- A k-strong component is a maximal k-connected induced subgraph of G.
- The strong connectivity of an edge, denoted by k_e, is the maximum k such that e is contained in a k-strong component.

The following is a more formal statement of the main theorem.

Theorem Given ε, let $q = 3(d+4)\ln n/\varepsilon^2$. For each edge, let $p_e = \min\{1, q/k_e\}$. Then, with probability $1-n^{-d}$, if we sample each edge with probability p_e and set its weight to be $1/p_e$ if e is chosen, then

① The resulting graph has $O(nq)$ edges.

② The resulting graph is a $(1\pm\varepsilon)$-approximation of the original graph.

We won't do the proof. See L11 of 2011 or [2].

Application To find an s-t min-cut, first sparsify the graph and find a min-cut, this will be an $(1-\varepsilon)$-approximate min-cut in the original graph. The fastest algorithm for min s-t cut runs in $O(mn)$ time, in the sparsified graph $m = n\log n$ and thus the running time is $\tilde{O}(n^2)$.

Actually, for the above application to work, we must also sparsify the graph quickly.
Computing strong connectivities takes too much time.
It turns out that an approximation of strong connectivities can be computed quickly in almost linear time, but it is not easy and we leave it to the paper.

Strong connectivities are easy for the proof but difficult for the algorithm.
It turns out that ordinary connectivities would also work, the proof becomes more difficult but the algorithm becomes simpler [3].

Minimum Cut in Near Linear Time [4]

It is an amazing result that uses graph sparsification to design faster exact algorithms.
We consider the problem of finding a global minimum cut in an undirected graph, i.e. to remove a minimum number of edges to disconnect the input graph.
Let k be the optimal value of this problem. So, the input graph is k-edge-connected.
Karger cleverly used a classical result on spanning tree packing.

Theorem (Tutte) If a graph is k-edge-connected, then it has $\lfloor k/2 \rfloor$ edge disjoint spanning trees.

Suppose we have edge disjoint spanning trees $T_1, T_2, \ldots, T_{k/2}$. Then, at least one tree T that crosses a minimum cut $S \subseteq V$ at most two times.
Karger observed that having such a T would help find a minimum cut efficiently, as one just needs to consider the cuts formed by removing at most two edges from T.
He showed that the minimum cut over these cuts can be computed in $\tilde{O}(m)$ time using dynamic programming (the case that T only crosses a minimum cut once is easier). [4].

Okay, but how to find such a tree T?
There is a known algorithm to find c edge disjoint spanning trees in $\tilde{O}(m \cdot c)$ time.
But this is not fast enough for this purpose.
The idea here, of course, is graph sparsification.
Just using the theorem proved in week 3, we can sparsify the graph so that its

min-cut is $O(\log n)$ while approximately preserving all the cuts.

Now, we can compute edge-disjoint spanning trees $T_1, \ldots, T_{O(\log n)}$ in the sparsifier, and one of these trees would cross a minimum cut at most two times.

So, doing dynamic programming on each tree would work, and total complexity is $\tilde{O}(m)$.

This is just a sketch of the main ideas.

References

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

[2] Benczur, karger. Randomized approximation schemes for cuts and flows in capacitated graphs.

[3] Fung, Hariharan, Harvey, Panigraph. A general framework for graph sparsification.

[4] Karger. Minimum cuts in near linear time.