1 Hitting set algorithms

Given a collection Σ of subsets of V, the hitting set problem is to find the smallest subset $S \subseteq V$ which intersects (hits) every set in Σ. If we regard Σ as defining a hypergraph on V (where each set in Σ constituting a hyperedge) then we see that the hitting set problem is equivalent to the vertex cover problem on hypergraphs; this problem is NP-hard. Here we will show two simple algorithms for finding reasonably small hitting sets.

Theorem 1.1. *Let $\Sigma = (S_1, \dots, S_n)$ where S_i is a subset of $V \equiv \{1, \dots, n\}$ of size $|S_i| \geq R$. There is a deterministic algorithm which runs in $O(nR)$ time and finds a subset $S \subseteq V$ with $|S| \leq (n/R) \ln n$ and $S \cap S_i \neq \emptyset$ for all i.*

Proof. Assume without loss of generality that $|S_i| = R$ for all i; otherwise drop all but the first R elements from every S_i before searching for S. Run the natural greedy algorithm: start with $S = \emptyset$, and for each $1 \leq j \leq n$, keep a counter $c(j) = |\{S_i \in \Sigma : j \in S_i\}|$. While $\Sigma \neq \emptyset$, let v be a maximizer of $c(v)$: update $S \leftarrow S \cup \{v\}$ and remove any subsets $S_i \ni v$ from Σ, decrementing all the appropriate counters $c(x)$ $(x \in V)$ whenever a subset containing x is removed from Σ (so in particular $c(v)$ will decrease to zero).

To obtain the runtime, we store the counts $c(j)$ in a data structure (e.g. a binary search tree) that can support the following operations in $O(\log n)$ time where n is the number of entries stored:

(a) insert an element

(b) return the element j of maximum value $c(j)$,

(c) decrement a given count $c(j)$.

The total number of decrements done by the algorithm to reach $\Sigma = \emptyset$ is nR which explains the runtime.

We still need to upper bound $|S|$. To this end, let T_j denote the number of sets remaining in Σ after j passes through the while loop, i.e. after j elements have been added to S. Then, $T_0 = |\Sigma| = n$, and $|S| = \min\{s \geq 0 : T_s = 0\}$.

Let u_j be the j-th element added to S, so $T_j = T_{j-1} - c(u_j)$. Just before we add u_j, the sum of counts $c(v)$ over $v \in V \setminus \{u_1, \dots, u_{j-1}\}$ must be precisely $T_{j-1}R$, so $c(u_j)$ must be at least the average count, which is $T_{j-1}R/(n-j+1)$ since there are $n-j+1$ elements with nonzero counts. Therefore

$$T_j \leq \Big(1 - \frac{R}{n-j+1}\Big) T_{j-1} \leq n \prod_{\ell=0}^{j-1} \Big(1 - \frac{R}{n-\ell}\Big) < n\Big(1 - \frac{R}{n}\Big)^j \leq n e^{-Rj/n},$$

and taking $j = (n/R) \ln n$ gives $T_j < 1$, therefore $T_j = 0$. Hence, $|S| \leq n/R \ln n$. □

Theorem 1.2. *Let $\Sigma = (S_1, \dots, S_n)$ with each S_i a subset of $V \equiv \{1, \dots, n\}$ of size $|S_i| \geq R$. For any constant $c > 0$, there is a randomized algorithm which runs in time $O(n)$ and finds a subset $S \subseteq V$ with $|S| \leq (n(1+c)/R) \log n$, such that $S \cap S_i \neq \emptyset$ for all i holds with probability $\geq 1 - n^{-c}$. The algorithm does not need to know Σ.*

Proof. Let $C \equiv 1 + c$, $s \equiv (nC/R) \ln n$. To return a hitting set having the correct size s *in expectation*, randomly add each element $v \in V$ to S with probability s/n. The probability for S to miss a particular set $S_i \in \Sigma$ is $\prod_{v \in S_i} \mathbb{P}(v \notin S) = (1 - s/n)^{|S_i|} \leq (1 - (C/R) \ln n)^R \leq n^{-C} = n^{-1-c}$, and taking the union bound over Σ proves that S fails to be a hitting set with probability at most n^{-c}.

To return a hitting set of exactly the correct size, choose a random subset of V of size s (that is, sample s elements without replacement). The probability for S to miss a particular subset $S_i \in \Sigma$ is

$$\prod_{j=1}^{s} \frac{n-R-(j-1)}{n-(j-1)} \leq (1-R/n)^s \leq n^{-C} \leq n^{-1-c}$$

where the j-th factor in the product is the probability for the j-th element added to S to avoid S_i. Taking a union bound over Σ as before proves that S fails to be a hitting set with probability at most n^{-c}. □

In the previous lecture, we showed that if we use the deterministic way of obtaining a small hitting set, we can obtain a deterministic $\tilde{O}(m\sqrt{n}+n^2)$ time algorithm that approximates the diameter of a graph. On the other hand, if we use the randomized way to obtain a hitting set, then the $O(n^2)$ time step of the algorithm that essentially produces the sets in Σ above can be avoided, since the randomized algorithm does not need to know Σ. Hence we would obtain an $\tilde{O}(m\sqrt{n})$ time algorithm which is faster for sparse graphs. The disadvantage is, of course, that the algorithm may fail to obtain a good estimate, albeit with very small probability.

2 Approximate APSP

Another application of hitting sets is given in the following combinatorial ("without matrix multiplication") algorithms devised by Aingworth et al. [1] and Dor et al. [2], giving +2-approximations to the all-pairs-shortest-paths (APSP) problem. Throughout the following, d denotes graph distance on the input graph.

2.1 Runtime $O(n^{5/2}\log n)$

Theorem 2.1 ([1])**.** *There is an algorithm which, given an n-vertex undirected unweighted graph, G, runs in time $O(n^{5/2}\log n)$ and computes estimates $d'(u,v)$ satisfying $d(u,v) \leq d'(u,v) \leq d(u,v)+2$ for all $u,v \in V$.*

The rough idea is as follows: Computing exact APSP (by running Dijkstra/BFS from all sources) is affordable only on a fairly sparse graph. The high-degree vertices are the computational bottleneck. To circumvent this, we use the low-degree high-degree technique. We partition the vertex set into low-degree (L) and high-degree vertices (H). High-degree vertices have large neighborhoods, and we can hit all their neighbors with a small hitting set S. A path in G either goes only through low-degree vertices, or passes within distance one of S. Thus, to estimate distances in G, it suffices to compute distances within L and distances from S, which can be done quickly since L has low-degree nodes and S is small.

To explicitly describe the algorithm, fix a parameter R (we will later set $R \approx n^{1/2}$ to optimize runtime). Let $N(v) \equiv \{v' \in V : d(v,v') \leq 1\}$, the depth-one neighborhood of vertex v.

Proof of Thm. 2.1. We first show that Algorithm 1 with general R returns the desired approximation, that is, $d(u,v) \leq d_K(u,v) \leq d(u,v)+2$ for all pairs $u,v \in V$. The lower bound is trivial: every path in K corresponds to a path in G of equal weight, so $d_K(u,v) \geq d(u,v)$. For the upper bound, let γ be the shortest path in G joining vertices u,v. If all vertices in γ are in L then γ is a path in K as well, so $d_K(u,v) = d(u,v)$. If not, we can find a high-degree vertex $h \in \gamma \backslash L$. Then, by construction, some $s \in S$ lies in $N(h)$, and $(s,u),(s,v) \in E_K$, therefore $d_K(u,v) \leq d_K(u,s)+d_K(s,v) = d(u,s)+d(s,v) \leq d(u,h)+d(h,v)+2 = d(u,v)+2$, where the $+2$ term arises because $s \in N(h)$ and the triangle inequality.

The runtime of Algorithm 1 is as follows: Computation of S takes time $O(nR\log n)$. Running BFS from all $s \in S$ takes time $O(|S|n^2)$. Forming the graph K has two steps: adding the L-incident edges takes time $O(|L|R) = O(nR)$, and adding the edges $((s,v) : s \in S, v \in V)$ takes time $O(|S|n)$. Recall that Dijkstra's algorithm on an n-vertex, m-edge graph runs in time $O(m+n\log n)$ using a Fibonacci heap. The graph K has $|E_K| = O(n(R+|S|))$, so solving APSP on K via Dijkstra from all sources takes time $O(n(|E_K|+n\log n)) = O(n^2(R+|S|+\log n))$. Summing these gives overall runtime $O(n^2(R+|S|+\log n))$ which is minimized by taking $R \approx n^{1/2}$, for runtime $O(n^{5/2}\log n)$ as claimed. □

Algorithm 1: AAPSP-ACIM($G = (V, E)$)

$L \leftarrow \{v \in V : |N(v)| \leq R\}$; $H \leftarrow V \setminus L$;
$S \leftarrow$ hitting set for $(N(v) : v \in H)$, $|S| = O((n/R) \log n)$;
foreach $s \in S$ **do**
  BFS(s) to compute $d(s, v)$ for each $v \in V$;
form new graph $K = (V, E_K)$ with edge weights w:
foreach $u \in L$ **do**
  add to E_K all edges $(u, v) \in E$, setting $w(u, v) = 1$;
  / in fact it suffices here to add (u, v) with both $u, v \in L$ */*
foreach $s \in S$ **do**
  foreach $v \in V$ **do**
    add edge (s, v) to E_K and set $w(s, v) = d(s, v)$
compute (exact) APSP on K (Dijkstra) to output $(d_K(u, v) : u, v \in V)$

Next time we will improve the algorithm obtaining the following theorem:

Theorem 2.2 ([2])**.** *There is an algorithm which, given an n-vertex undirected unweighted graph, runs in time $\tilde{O}(n^{7/3})$ and computes estimates $d'(u, v)$ satisfying $d(u, v) \leq d'(u, v) \leq d(u, v) + 2$ for all $u, v \in V$.*

References

[1] Aingworth, D., Chekuri, C., Indyk, P., and Motwani, R. (1999). Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4), 1167-1181.

[2] Dor, D., Halperin, S., and Zwick, U. (2000). All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5), 1740-1759.