1 Hitting Sets

"Given a collection $\Sigma$ of subsets of $V$, the hitting set problem is to find the smallest subset $S \subseteq V$ which intersects (hits) every set in $\Sigma$. If we regard $\Sigma$ as defining a hypergraph on $V$ ([with] each set in $\Sigma$ 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." [2]

Some algorithms follow for finding reasonably small hitting sets.

**Theorem 1.1.** There is an $O(n \times R)$ time algorithm that given $\Sigma$ computes a hitting set $T$ of size $\leq \frac{n}{R} \times \ln n$

**Theorem 1.2.** For all $c > 0$, there is an $O(n)$ time algorithm that computes $T$ of size $\leq (c + 1) \frac{n}{R} \ln n$ s.t. with probability $> 1 - \frac{1}{n^c}$ for all $i$, $S_i \cap T \neq \emptyset$. Algorithm does not need to know $\Sigma$.

Fact: $(1 - \frac{1}{R})^x \leq \frac{1}{e}$

Proof of Theorem 1: Drop all but the first $R$ elements from each $S_i$. Now $|S_i| = R$. A greedy algorithm that can satisfy theorem 1 is below:

**Algorithm 1:** Hitting-Set Greedy($n$)

\[
\begin{align*}
&\text{foreach } j \in 1, \ldots, n \text{ do} \\
&\quad \text{sets}(j) = \{S_i | j \in S_i\}; \\
&\quad \text{count}(j) = |\text{sets}(j)| \\
&\text{Insert all } j \in [n] \text{ into } F; \\
&\text{while } \Sigma \neq \emptyset \text{ do} \\
&\quad j = F.\text{extract-max}; \\
&\quad \text{foreach } S_i \in \text{sets}(j) \text{ do} \\
&\quad\quad \Sigma = \Sigma \setminus S_i; \\
&\quad\quad \text{foreach } k \in S_i \text{ do} \\
&\quad\quad\quad F.\text{decrease-key}(k) \\
&\quad T = T \cup \{j\} \\
&\text{return } T
\end{align*}
\]

$F$ is a data structure supporting:

1. $\text{insert}(e, k)$
2. $\text{extract-max}$: finds largest key element, removes it from $F$ and returns it.
3. $\text{decrease-key}(j)$: decrement $j$’s key by 1.

It has $O(\log n)$ time per operation using a binomial heap.

**Claim 1.** $|T| \leq \frac{n \ln n}{R}$

**Proof of Claim 1.**

Let $W_j$ be the number of sets in $\Sigma$ after $j$-th element is added to $T$. $(W_0 = n)$

Let $u_j$ be the $j$-th element added to $T$.

$W_j = W_{j-1} - \text{count}(u_j)$. (*)

1
Consider the sum of counts of the elements in \( U = \{1, 2, ..., n\}/T \) after \( j - 1 \) elements are added to \( T \).

This is exactly \( W_{j-1} \times R \).

\[
|U| = n - (j - 1)
\]

The average count of an element in \( U \) (after \( j - 1 \) elements added to \( T \)) is \( \frac{W_{j-1} \times R}{n - (j - 1)} \).

\[
\text{count}_{U_j} \geq \frac{W_{j-1} \times R}{n - (j - 1)}.
\]

By (*) \( W_j \leq W_{j-1}(1 - \frac{R}{n - (j - 1)}) \).

\[
\Rightarrow W_j \leq W_0 \times < PI > j = 1((1 - \frac{R}{n - (j - 1)}) \leq 1 - \frac{R}{n}) < n \times (1 - \frac{R}{n})^j.
\]

\[
W(\frac{R}{n}) \ln n < n \times [(1 - \frac{R}{n}) \frac{n}{1} \ln n
\]

< \( n \times (\frac{1}{2})^{\ln n} = \frac{n}{2} = 1 \)

So after \( \frac{n}{2} \ln n \) elements are added to \( T \), all sets are covered.

\[\square\]

**Proof of Theorem 1.2.**

\( \Sigma = \{S_1, ..., S_n\} \) (may be unknown)

Let \( T \leq \{1, ..., n\} \) be a uniformly random subset of size \((c + 1)\frac{R}{n} \ln n\).

We’ll show that with probability \( \geq 1 - \frac{1}{n}c, \forall i, S_i \cap T \neq \emptyset \).

Fix some \( S_i \).

\[
Pr[T \cap S_i = \emptyset] = \prod_{j=1}^{T} [(n-2(j-1)) \frac{n-2(j-1)}{n-2j}] = \prod_{j=1}^{T} (1 - \frac{R}{n-2j}) \leq ((1 - \frac{R}{n})^T)^{(c+1)\ln n}.
\]

(The initial term within the multiplicative sum expression is the probability that \( j \)-th element of \( T \) misses \( S_i \).)

Union Bound: \( \text{Prob[some } S_i \text{ has } T \cap S_i = \emptyset] \leq n \times \text{Pr[fixed } S_i \text{ has } T \cap S_i = \emptyset] \leq n \times \frac{1}{n^{c+1}} = \frac{1}{n^c} \)

\( \Rightarrow \) With probability \( \geq 1 - \frac{1}{n^c}, T \) is a hitting set.

\[\square\]

**Definition 1.1.** Union bound: If we have \( n \) bad events each occurring with probability \( \leq P \), then the probability that some bad event happens is at most \( n \times P \).

"In the previous lecture, we showed that if we use the deterministic way of obtaining a small hitting set, we can obtain a deterministic \( \tilde{O}(m^2 \sqrt{n} + n^2) \) 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 \( \Sigma \) above can be avoided, since the randomized algorithm does not need to know \( \Sigma \). 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]

2 Approximate Shortest Paths

All pairs shortest paths in undirected, unweighted graphs \( G = (V, E), |V| = n \): The problem is to compute \( \forall u, v d(u, v) \). Seidel: \( \tilde{O}(n^{2.373}) \) time result uses fancy matrix multiplication. This is not currently practical. It can be shown that a ‘practical’, combinatorial algorithm can’t be done for exact solutions to APSP.

**Claim 2.** If APSP is in \( O(n^c) \) time then one can multiply \( m \times n \) matrices in \( O(n^c) \) time.

**Proof of Claim 2.** Boolean matrices \( A, B \) (entries in \{0, 1\}) \( C(i, j) = \sum_{k=1}^{n} (A(i, k) \land B(k, j)) \) ← the best known algorithm runs in \( O(n^{2.373}) \).

If matrix contents described as a graph s.t. we have three sections of nodes for values of \( i, j \), and \( k \) respectively, and place an edge \( i, k \) if \( A(i, k) = 1 \) and an edge \( k, j \) if \( B(k, j) = 1 \), there are paths s.t. \( \text{dst}(i, j) = 2 \) if \( \sum_{k=1}^{n} (A(i, k) \land B(k, j)) = 1 \), and \( \geq 4 \) otherwise. Thus an APSP algorithm could be used to solve Boolean matrix multiplication. Since this would be an implausible breakthrough in matrix multiplication we cannot expect to find a "practical" APSP algorithm.

\[\square\]
Theorem 2.1. There is a ("simple") $O(n^{2.5} \log n)$ time algorithm that computes a +2-approximation to APSP. (Also, there is an $O(n^3)$ time algorithm for a +2-approximation.)

"The rough idea is as follows. Computing exact APSP (by running Dijkstra/BFS from all sources) is affordable only on a fairly sparse graph, so the high-degree vertices are the computational bottleneck. To circumvent this, the algorithm partitions the vertex set into low-degree and high-degree vertices. High-degree vertices have large neighborhoods, and we can hit all these neighborhoods 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 $L$ is low-degree and $S$ is small." [2]

Algorithm 2: +2-Approx-APSP

Compute a hitting set $S$ for $\Sigma = \{ N(x) | \forall v : \text{deg}(v) > \sqrt{n} \}$

$|S| \leq O(\sqrt{n} \log n)$

For each $s \in S$, run $BFS(s) \rightarrow \frac{\sqrt{d(s,v)}}{O(\sqrt{n} \log n)}$

For $u, v \in V$ set $d_1(u,v) = \min(d(u,s) + d(s,v))$ (min over $s \in S$).

$G_{low} = (V, E')$ (add only edges incident to nodes of low degree, so $|E'| \leq n^{1.5}$) is the subgraph of $G$ where $(u,v) \in E'$ iff $u$ or $v$ has degree $< \sqrt{n}$.

For each $v \in V$, run $BFS(v)$ in $G_{low}$

$\rightarrow \forall u,v \ d'(u,v) \in G_{low}$

For $u, v \in V$ return $\min\{d_1(u,v), d'(u,v)\} \leftarrow d(u,v)$.

In high degree case, the algorithm’s running time is $O(\frac{R^3}{5} \log n)$ since $|S| = O(\frac{R}{5}) \log n$. In the low-degree case, it is $O(n^2 D_2)$. To minimize these, set $D_1 = D_2$, giving $D = \sqrt{n}$.

Claim 3. For all $u, v$, $d(u,v) \leq \bar{d}(u,v) \leq d(u,v) + 2$.

Proof of Claim 3. Fix $u, v$, let $P$ be the shortest path from $u$ to $v$. Case 1: $P$ does not have any high-degree nodes. Then $P$ is in $G_{low}$, thus $d'(u,v) = d(u,v)$. Case 2: $P$ has some high degree node $h$. Let $s \in S$ be a neighbor of $h$. We know that for this $s$ the distance estimate between $u$ and $v$ is $\bar{d}(u,v) \leq d(u,s) + d(s,v)$. But by triangle inequality, $d(u,s)$ is at most $d(u,h) + d(h,s)$, and $d(s,v)$ is at most $d(s,h) + d(h,v)$. $d(h,s) = 1$, as $s$ is a neighbor, so this gives $\bar{d}(u,v) = d(u,v) + 2$. $\square$

"Proof of Thm. 2.1. We first show that Algorithm 2 with general $R$ returns the desired approximation, that is, $d(u,v) \leq \bar{d}_K(u,v) \leq d(u,v) + 2$ 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 $\gamma$ be the shortest path in $G$ joining vertices $u, v$. If all vertices of $\gamma$ are in $L$ then $\gamma$ is a path in $K$ as well, so $d_K(u,v) = d(u,v)$. If not, we can find a high-degree vertex $h \in \gamma \setminus L$: 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 2.1 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 $O(n^{7/2} \log n)$ as claimed." $\square$

[1] by way of [2]

2.1 Runtime $O(n^{7/3})$

(This coverage of the $O(n^{7/3})$ algorithm from last year’s notes [2])."
Theorem 2.2 ([3]). There is an algorithm which, given an n-vertex undirected unweighted graph, runs in time $O(n^{7/3})$ and computes estimates $d'(u, v)$ satisfying $d(u, v) \leq d'(u, v) \leq d(u, v) + 2$ for all $u, v \in V$.

Algorithm 3: AAPSP-DHZ($G = (V, E)$)

\begin{verbatim}
L ← \{v ∈ V : |N(v)| ≤ n^{1/3}\}; H ← \{v ∈ V : |N(v)| > n^{2/3}\}; M ← V \setminus (L \cup H);
S ← hitting set for (N(v) : v ∈ H), |S| = O(n^{1/3} \log n);
T ← hitting set for (N(v) : v ∈ M), |T| = O(n^{2/3} \log n);

foreach s ∈ S do
    BFS(s) to compute $d(s, v)$ for each $v ∈ V$;

G_{mid} ← G with only edges incident to $L \cup M$;

foreach t ∈ T do
    BFS_{mid}(t) (BFS in $G_{mid}$) to compute $d_{mid}(t, v)$ for all $v ∈ V$;

form new graph $K_o = (V, E_o)$ with edge weights $w$:

foreach u ∈ L do
    add to $E_o$ all edges $(u, v) ∈ E$, setting $w(u, v) = 1$;

foreach s ∈ S do
    
    for each $v ∈ V$ do
        add edge $(s, v)$ to $E_o$ and set $w(s, v) = d(s, v)$;

foreach x ∈ V do
    form new graph $K_x = (V, E_x)$ with edge weights $w$: initialize $K_x ← K_o$

foreach t ∈ T do
    add edge $(x, t)$ to $E_x$ and set $w(x, t) = d_{mid}(x, t)$;

foreach z ∈ M do
    add a single edge $(z, t)$ to $E_x$ for some $t ∈ T \cap N(z)$ and set $w(z, t) = 1$;

compute (exact) sssp from $x$ in $K_x$ (Dijkstra) to obtain distances $d_x(x, v)$ for all $v ∈ V$;

output $(d'(x, y) : x, y ∈ V)$ where $d'(x, y) = d_x(x, y) ∧ d_y(y, x)$
\end{verbatim}

Proof of Thm. 2.2. We first show the approximation bounds $d(u, v) ≤ d'(u, v) ≤ d(u, v) + 2$. As before, the lower bound is trivial. For the upper bound, in view of the proof of Thm. 2.1, it suffices to consider the case that the shortest path $γ$ joining vertices $x, y ∈ V$ involves only vertices of $L \cup M$, but not $L$ alone. In this case, let $z$ be the vertex in $γ \cap M$ which is furthest from $x$, and let $t$ be the chosen vertex of $T \setminus N(z)$ for which $(z, t)$ was added to $E_x$. By the triangle inequality for $d_x$ we have $d_x(x, y) ≤ d_x(x, t) + d_x(t, z) + d_x(z, y)$; the middle term $d_x(t, z)$ is simply 1. The path from $z$ to $y$ involves only $L$-incident edges, so $d_x(z, y) = d(z, y)$. Lastly $d_x(x, t) ≤ d_{mid}(x, t) ≤ d_{mid}(x, z) + d_{mid}(z, t)$ (by the triangle inequality for $d_{mid}$), and this equals $d(x, z) + d(z, t) = d(x, z) + 1$ by our assumption that $γ$ involves only $L \cup M$. Combining these inequalities, $d_x(x, y) ≤ d_{mid}(x, t) + d_x(t, z) + d_x(z, y) ≤ d_{mid}(x, t) + 1 + d(z, y) ≤ d(x, z) + d(z, y) + 2 = d(x, y) + 2$, where the last equality holds because $P$ was assumed to be a shortest path.

We now check that the algorithm has the claimed runtime: Computation of $S$ takes time $O(n^{5/3} \log n)$; computation of $T$ takes time $O(n^{4/3} \log n)$. Running BFS from all $s ∈ S$ takes time $O(|S|^2) = O(n^{7/3} \log n)$. The graph $G_{mid}$ has $|E_{mid}| = O(n^{5/3})$ edges, so running BFS in $G_{mid}$ from all $t ∈ T$ takes time $O(|T||E_{mid}|) = O(n^{7/3} \log n)$. Forming $K_o$ takes time $O(|L|n^{1/3} + |S||V|) = O(n^{4/3} \log n)$. For each $x ∈ V$, forming $K_x$ takes time $O(|T| + |M|) = O(n)$ if we do not re-compute $K_o$ (although we can afford to). Then $|E_x| = O(|L|n^{1/3} + |S|n + |T| + |M|) = O(n^{4/3} \log n)$, so running Dijkstra from $x$ in $K_x$ takes time $O(n^{4/3} \log n)$. Running over $x ∈ V$ gives final runtime $O(n^{7/3} \log n)$ as claimed.  

"[2]
References

