FULLY-DYNAMIC ALL-PAIRS OF SHORTEST PATHS

Fully Dynamic(=Online) APSP

- N vertices
- Directed
- weighted(no negative cycles) or unweighted
- Begins with empty graph
- Inserts or delete vertices in each update
- Answers queries of the form "distance between shortest path from s to t" in $O(1)$
- (optional but usually possible) Find first k edges of a shortest path in $O(k)$

Randomization

- I will skip all parts about randomization because too much…

- The use of any kind of **randomness** is prohibited. Any solution seen using randomness (whether it provably works or not) will be disqualified. This includes writing your own pseudorandom generators instead of using the rand() function.

Previous Results

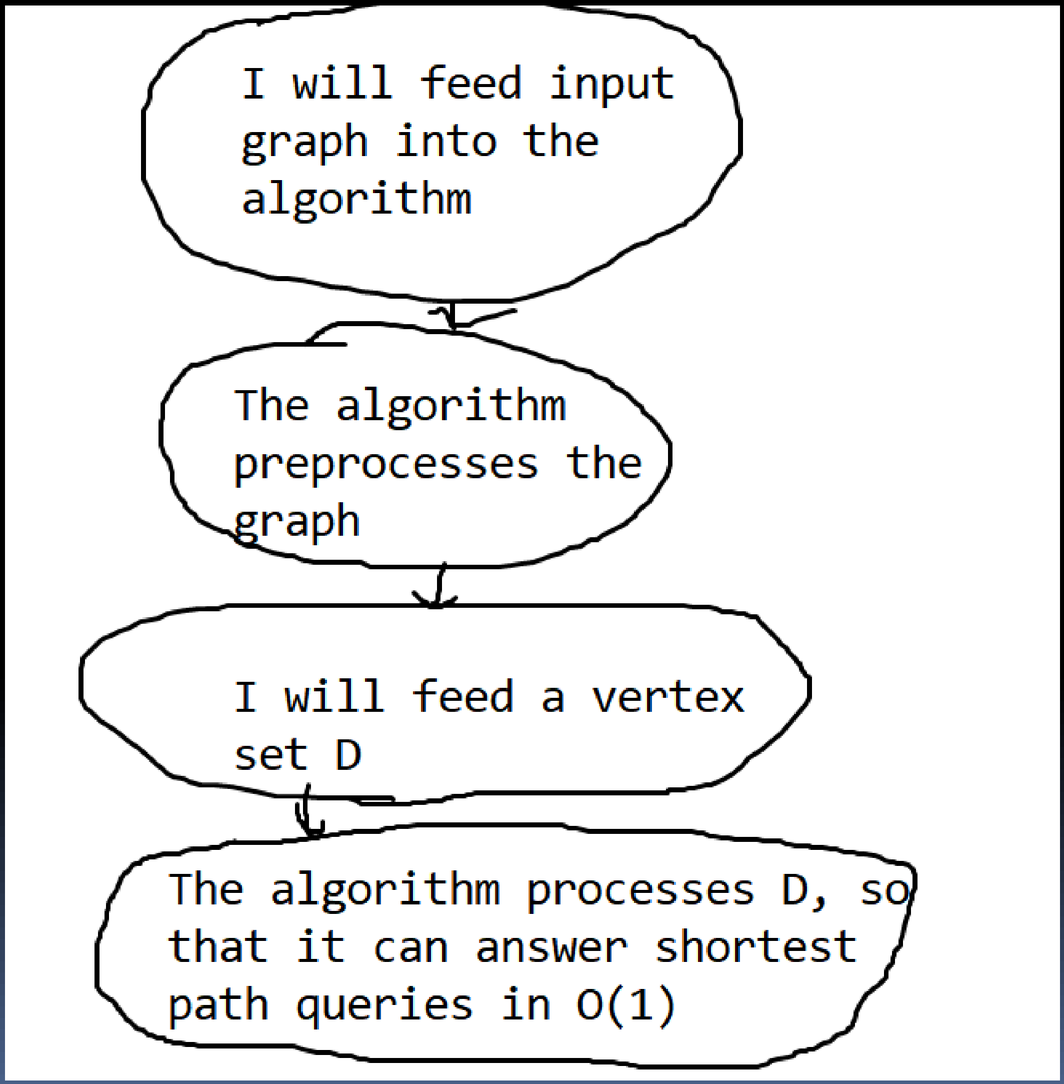
	Amortized update time	Worst case update time	Weighted	Space
Brute	$O(n^3)$	$O(n^3)$	Yes	$O(n^2)$
DI04,THO04	$O(n^2 \log^2 n)$	$O(n^3)$	Yes	$O(n^3)$
THO05	$\tilde{O}(n^{2+3/4})$	$\tilde{O}(n^{2+3/4})$	w>=0	Super-cubic
Gutenberg, Nilsen 2020	$O(n^{\frac{19}{7}} \log^{8/7} n)$	$O(n^{\frac{19}{7}} \log^{8/7} n)$	Yes	Sub-cubic
Gutenberg, Nilsen 2020	$O(n^{2.6}\, log\, n)$	$O(n^{2.6}\, log\, n)$	No	Sub-cubic
Gutenberg, Nilsen 2020	$O(n^{\frac{11}{4}} \log^{2/3} n)$	$O(n^{\frac{11}{4}} \log^{2/3} n)$	Yes	$O(n^2)$
Gutenberg, Nilsen 2020	$O(n^{\frac{8}{3}} \log^{2/3} n)$	$O(n^{\frac{8}{3}} \log^{2/3} n)$	No	$O(n^2)$

Definitions

- h-hop path: Path with at most h edges (not same as distance)
- $hop(p)$: number of edges on path p
- $dist_H\ (s, t)$: shortest path from s to t in the induced subgraph regarding H
- Improving path from s to t: path rom s to t with distance at most $dist(s, t)$
- Improving path from s to t with regards to H: path from s to t (the path doesn't have to be in H) with distance at most $dist_H\ (s, t)$

Reduction to batch deletion

- "Batch Deletion Problem"



Reduction to batch deletion

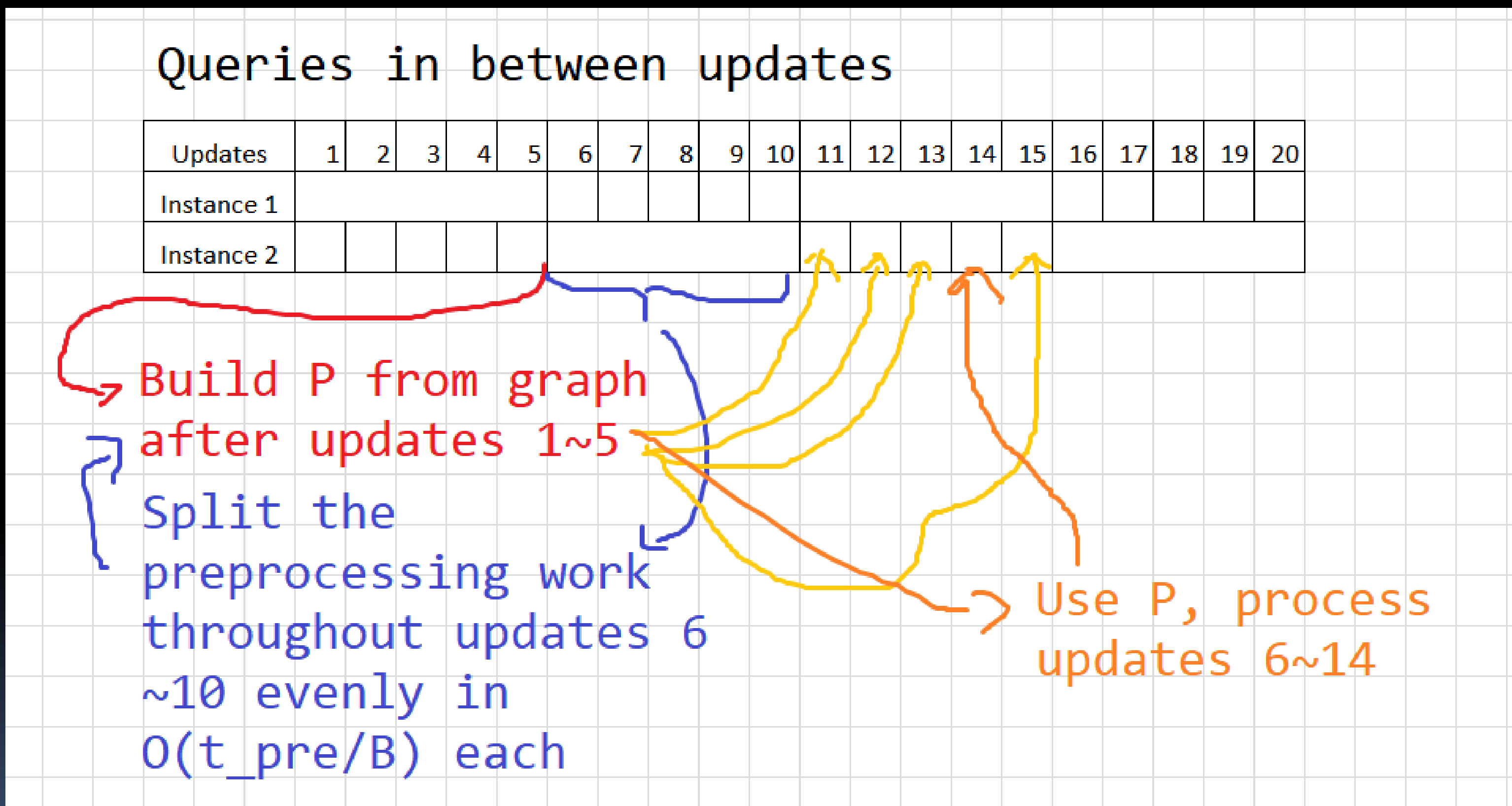
- Imagine we have a data structure (call it P) that:
  - Inputs a graph and preprocesses in time $O(t_{pre})$
  - Can handle one single batch deletion of a vertex set D with size $\leq 2B$ in $O(t_{del})$
  - Returns shortest path between all pairs of nodes in new graph (after deletion)
- We can use it to solve the fully dynamic APSP in worst update time $O\left(\frac{t_{pre}}{B} + t_{del} + Bn^2\right)$
- Standard deamorization techniques

Reduction to batch deletion

- Insertion = Easy
- Modified Floyd-Warshall
  - Compute distance from each original node to new node
  - Compute distance from new node to each original node
  - Update all $dist(s,t)$ with $dist(s,new) + dist(new,t)$
- $O(kn^2)$ for inserting k vertices

Reduction to batch deletion

Assume B=5



Reduction to batch deletion

- Note that in the 11^{th} update, you answer queries using P and process update 6~11
- Then for the 12^{th} update , you discard the processing and start over using P and process updates 6~12
- A bit confusing, took me a while to understand

Reduction to batch deletion

- The preprocessing time is spread across B updates and therefore worst update time for this step is $O\left(\frac{t_{pre}}{B}\right)$
- When you use P and process updates
  - Process deletions first in $O(t_{del})$
  - Use the results and apply modified Floyd-Warshall to process insertion updates in $O(Bn^2)$
- $O\left(\frac{t_{pre}}{B} + t_{del} + Bn^2\right)$ worst update time total

Powerful Blackbox

- Given graph G
- Given a parameter h
- Given n^2 paths $p_{s,t}$ such that
  - If shortest path (if many, the one with least edges) from s to t in G consists of at most h edges (h-hop), then $p_{s,t}$ is this path.
- Returns all pairs of shortest path in G
- Time Complexity: $O(\frac{n^3 logn}{h} + n^2 \log^2 n + n^2 h)$

Powerful Blackbox

- In other words
- If we already found all h-hop improving shortest paths with regards to G
  - That is, if there exist path p such that
  - Starts at s and ends at t
  - $hop(p) \leq h$
  - $dist(p) = dist(s, t)$
  - Then $p_{s,t}$ will be one of the valid p
  - Otherwise $p_{s,t}$ can store any valid path between s, t
- We can find all pairs of shortest path

Powerful Blackbox

- Will describe near the end of the presentation
- Lets find all h-hop improving shortest paths first

Slow Deletion

- Motivation
  - If we only maintain shortest paths that consist of little edges
  - Handling deletion is easier because we only need to recompute paths that includes at least one of the elements in the deleted set
  - As the paths are short, the probability of recomputation is small
  - We can then use our powerful blackbox to extend our short paths to all pairs of shortest path

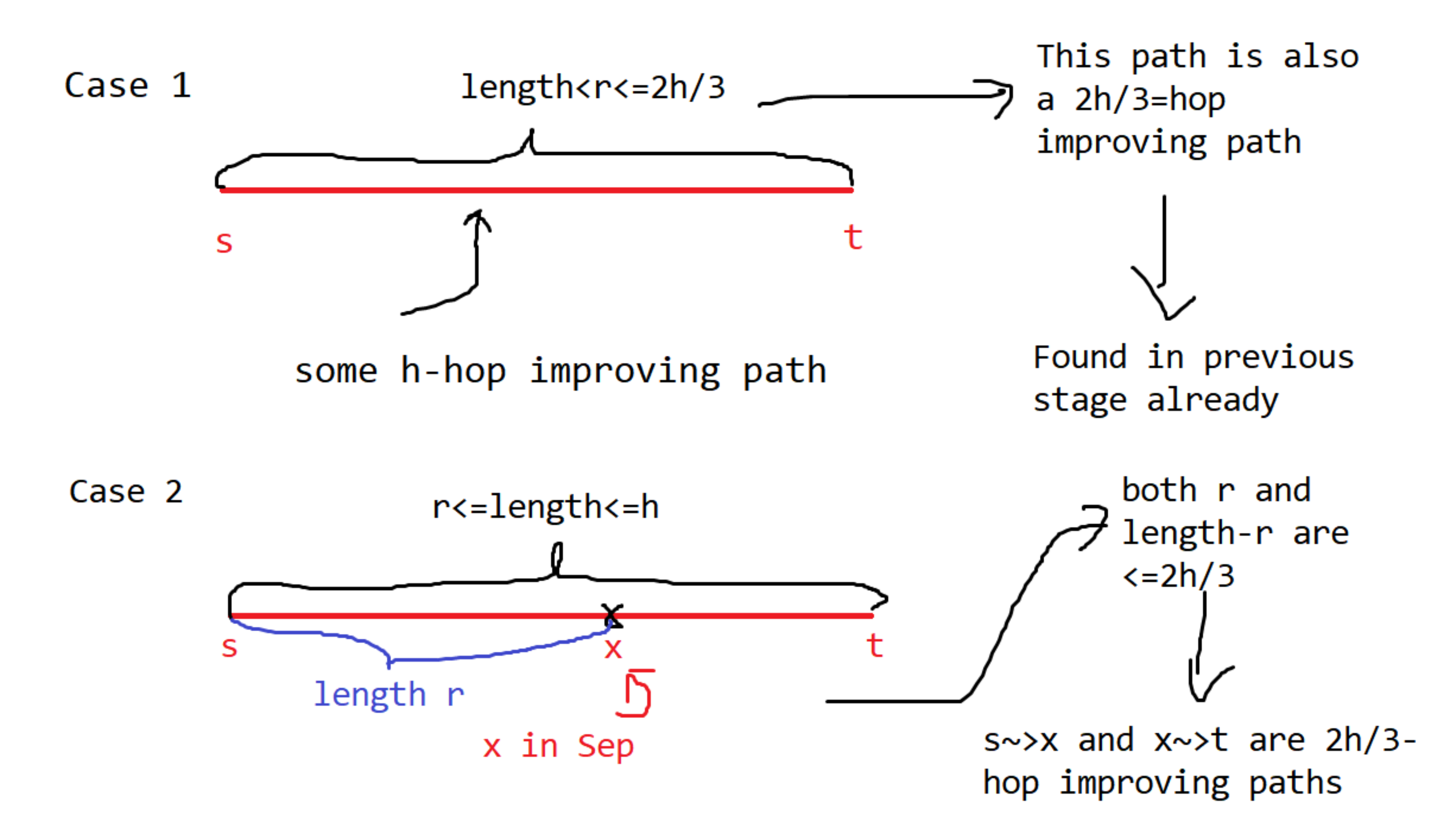
Slow Deletion

- Objective: Find all h-hop improving paths in $\mathrm{G} \backslash \mathrm{D}$
  - D is the set to be deleted
  - Well, not exactly “all”, we only need to find one path between each pair of vertices
  - Very important to understand what it means
  - If the shortest path from s to t that consist of least edges has <=h edges, it is an h-hop improving path
  - We don’t have to know which are h-hop improving paths and which aren’t, we just have to make sure that every such path in our set of paths
- Method: Recursion/Induction

Slow Deletion

- Lets say we are finding all h-hop improving paths that starts from s
- Imagine we have all $\frac{2h}{3}$-hop improving paths between all pairs of vertices already
- Pick some integer $r\epsilon[\frac{h}{3},\frac{2h}{3}]$ and let Sep be the set of all vertices x such that the shortest path (which is already found) from s to x consists of <u>exactly</u> r edges

Slow Deletion



Slow Deletion

- So, to find the h-hop improving path from s to t
- We will enumerate x from Sep, and see if the concatenation of s~>x and x~>t is shorter than the current path we have from s to t
- As there is $O(h)$ choices for r, and there are at most $O(n)$ paths we have starting from s
- We can pick r such that $|Sep| = O(\frac{n}{h})$

Slow Deletion

- Some Notations for convenience
  - $i_h = \lceil \log_{1.5} h \rceil$ is the number of phases of "binary lifting" we will do (in fact it is 1.5-lifting)
  - $1 \leq i \leq i_h$ is the phase number of the current phase
  - $h_i = 1.5^i$ is the target hop number that you want to find all shortest path in $V \backslash \mathrm{D}$ of hops $\leq h_i$, during phase i

Slow Deletion

```
path p[log(N)/log(1.5)][N][N];
int adj[N][N];//adjacency matrix
int cnt[N];
bool better(path x,path y){
    if(dist(x)!=dist(y)) return dist(x)<dist(y);
    return hop(x)<hop(y);
}
void slowDelete(vector<int>D, int h){
    for(s in V\D)
        for(t in V\D)
            p[0][s][t]=adj[s][t];
    i_h=ceil(log(h)/log(1.5));
    for(int i=1; i<=i_h ;i++){
        int h_i=pow(1.5,i);//in this iteration, we are computing h_i-hop shortest paths
        for(s in V\D){
            int r;vector<int>Sep;
            {//finding the best r
                /*initialize cnt*/
                for(x in V\D) cnt[hop(p[i-1][s][x])]++;
                r=h_i/3;
                for(int j in [h_i/3,2*h_i/3]) if(cnt[j]<cnt[r]) r=j;
                for(x in V\D) if(hop(p[i-1][s][x])==r) Sep.push_back(x);
            }
            for(t in V\D){
                p[i][s][t]=p[i-1][s][t];
                for(x in Sep){
                    if(better(p[i-1][s][x]+p[i-1][x][t],p[i][s][t])) p[i][s][t]=p[i-1][s][x]+p[i-1][x][t];
                }
            }
        }
    }
}
```

fast

O(n/h_i)

Remember the definition of i_h and h_i, will be used frequently later

Less Slow Deletion

- Why previous algorithm slow?
  - $O(n/h_i)$ part is computed many times
  - When h_i is small, will take $O(n^3)$
  - Doesn't use preprocessing
- How to improve?
  - Preprocess all h-hop improving paths in G
  - We can skip $O(\frac{n}{h_i})$ part if $p_{s,t}$ doesn't not contain elements in D already

Less Slow Deletion

```
void badPreprocess(int h){
    i_h=ceil(log(h)/log(1.5));
    vector<int>X=V;
    while(!X.empty()){
        int s=X.back();X.pop_back();
        for(int i=1; i<=i_h ;i++){
            int h_i=pow(1.5,i);//in this iteration, we are computing h_i-hop shortest paths
            bellmanford(s,V,h_i);//finds all shortest path from s to t with hop at most h_i
            /*stores result in p[i][s][t]*/
        }
    }
}
```

Bellmanford is $O(n^3 h_i)$

Preprocessing is $O(n^3 h)$ due to geometric sum

$$\sum h_i = 1 + 1.5 + 2.25 + \cdots + \mathrm{h} = \frac{1.5\mathrm{h} - 1}{1.5 - 1} = \mathrm{O(h)}$$

Less Slow Deletion

```
void lessSlowDelete(vector<int>D, int h){
//of course, we don't actually rewrite the content in p[] because we have to use them multiple times
//we will make a copy instead
    for(s in V\D)
        for(t in V\D)
            p[0][s][t]=adj[s][t];
    i_h=ceil(log(h)/log(1.5));
    for(int i=1; i<=i_h ;i++){
        int h_i=pow(1.5,i);//in this iteration, we are computing h_i-hop shortest paths
        for(s in V\D){
            int r;vector<int>Sep;
            /*compute r,Sep like in slowDelete*/
            for(t in V\D){
O(h_i)              if(p[i][s][t] and D does not intersect) continue;
                //to check this, we need to maintain the intermediate nodes in p[i][s][t], not just only hop and dist
                p[i][s][t]=p[i-1][s][t];
                for(x in Sep){
O(n/h_i)                best=t;
                    if(better(p[i-1][s][x]+p[i-1][x][t],p[i-1][s][best]+p[i-1][best][t])) best=x;
                }
                p[i][s][t]=p[i-1][s][best]+p[i-1][best][t];
O(h_i)          if(hop(p[i][s][t])>h_i) p[i][s][t]=NULL;//to keep the hop of paths in O(h_i)
            }
        }
    }
}
```

The $O(h_i)$ parts contributed to $O(n^2h)$ due to geometric sum, similar to previous slide

Fast Deletion

- Why is previous algorithm slow?
  - It does not improve asymptotically, the $O\left(\frac{n}{h_i}\right)$ parts still is computed many times
- How to improve?
  - We can make use that insertion is fast
  - We don't want some vertex to appear in our p[][][] too frequently, as it would be costly to delete
  - We maintain a set C that appears in h-hop improving shortest path very frequently, and ignore them during preprocessing, then after we finish doing delete(), we add the C vertices back using modified Floyd-Warshall

Fast Deletion

Threshold (represented as τ later) is $\geq 2n^2$

```
int congestion[N];
void Preprocess(int threshold, int h){
    i_h=ceil(log(h)/log(1.5));
    vector<int>X=V;
    vector<int>C;//initially empty
    while(!X.empty()){
        int s=X.back();X.pop_back();
        for(int i=1; i<=i_h ;i++){
            int h_i=pow(1.5,i);//in this iteration, we are computing h_i-hop shortest paths
            bellmanford(s,V\C,h_i);
            //finds all shortest path from s to t with hop at most h_i in the induced subgraph regarding h_i
            for(t in V\C){
                for(vertex v in p[i][s][t]){
                    congestion[v]+=ceil(n/h_i);
                }
            }
            for(vertex v not in C){
                if(congestion[v]>=threshold) C.push_back(v);
            }
            /*stores result in p[i][s][t]*/
        }
    }
}
```

Each occurrence of vertex v in p[i] will cause O(n/h_i) time in Delete()

Fast Deletion

- So in our process, when vertices occurred too much in the paths we found, we will stop considering it
- This way, the paths we get are h_i-hop improving paths regarding $G\backslash C$
  - dist(p[i][s][t]) is at most the distance from s to t with h_i hop in $G\backslash C$
  - p[i][s][t] may contain vertices from C, but it is ok

Fast Deletion

- $Congestion(v) \leq 2\tau$ for all vertices
  - After each iteration of bellman-ford, at most $\mathrm{n}(\mathrm{h_i})(\frac{\mathrm{n}}{\mathrm{h_i}}) = n^2$ is added to all vertices in total, and congestion never increases once it exceeds τ
- Sum of $Congestion(v)$ is $O(n^3 \log h)$
  - After each iteration of bellman-ford, at most $\mathrm{n}(\mathrm{h_i})(\frac{\mathrm{n}}{\mathrm{h_i}}) = n^2$ is added to all vertices in total, and bellman-ford is ran for $O(n \log h)$ times.
- $|C|$ is $O(\frac{n^3 \log h}{\tau})$
  - Obvious using above results

Fast Deletion

```
void Delete(vector<int>D, int h){//basically just lessSlowDelete but V\(D Union C) instead of V\D
//of course, we don't actually rewrite the content in p[] because we have to use them multiple times
//we will make a copy instead
    vector<int>Ban=D union C;
    for(s in V\Ban)
        for(t in V\Ban)
            p[0][s][t]=adj[s][t];
    i_h=ceil(log(h)/log(1.5));
    for(int i=1; i<=i_h ;i++){
        int h_i=pow(1.5,i);//in this iteration, we are computing h_i-hop shortest paths
        for(s in V\Ban){
            int r;vector<int>Sep;
            /*compute r,Sep like in slowDelete*/
            for(t in V\Ban){
                if(p[i][s][t] and D does not intersect) continue;//still D here
                //to check this, we need to maintain the intermediate nodes in p[i][s][t], not just only hop and dist
                p[i][s][t]=p[i-1][s][t];
                /*update p[i][s][t] using Sep*/
            }
        }
    }
    ////////////
    for(vertex v in C\D) insert(v);//insert in p[i_h][s][t] like floyd warshall
}
```

Units of time taken here is bounded by sum of congestion of vertices in D

Fast Deletion

- Basically it is almost the same as lessSlowDelete, except that we might miss some paths that pass through elements in C
- In phase i, the algorithm will find all h_i-improving paths in $G\setminus \mathrm{D}$, if the path does not contain elements in C
- After the algorithm, we will use modified floyd-warshall to insert elements in C and find all paths that we missed
- Lastly we will plug into the blackbox and get our desired distance matrix

Fast Deletion

- Everytime we need to recompute a path in stage i, it takes $O(\frac{n}{h_i})$ time and also gives $\frac{n}{h_i}$ contribution to the congestion of the vertex that is in D and in the path
- Time complexity of the path recomputation path is $O(|D|\tau) = O(B\tau)$
- Time complexity of the insertion of vertices in C is $\mathrm{O}(|\mathrm{C}|\mathrm{n}^2) = O(\frac{n^5 \log h}{\tau})$

Complexity Analysis

- $\mathrm{O}\left(\frac{t_{pre}}{B} + t_{del} + Bn^2\right)$
- $t_{pre} = O(n^3 h)$
- $t_{del} = O\left(B\tau + \frac{n^5 \log h}{\tau}\right) + O(\frac{n^3 logn}{h} + n^2 \log^2 n + n^2 h)$
- Choose $\tau = n^{2.25} \log^{0.5} n, h = n^{0.25} \log^{0.5} n, B = n^{0.5}$
- Complexity $O(n^{2.75} \log^{0.5} n)$

Complexity Analysis

- When unweighted, the bellman-ford during preprocessing can be replaced by BFS
- Complexity can be reduced to Complexity $O(n^{\frac{8}{3}} \log^{\frac{2}{3}} n)$
- Choice of parameters is left as exercise to readers (wasn't given in the paper)

Powerful Blackbox

- Given graph G
- Given a parameter h
- Given n^2 paths $p_{s,t}$ such that
  - If shortest path (if many, the one with least edges) from s to t in G consists of at most h edges (h-hop), then $p_{s,t}$ is this path.
- Returns all pairs of shortest path in G
- Time Complexity: $O(\frac{n^3 logn}{h} + n^2 \log^2 n + n^2 h)$

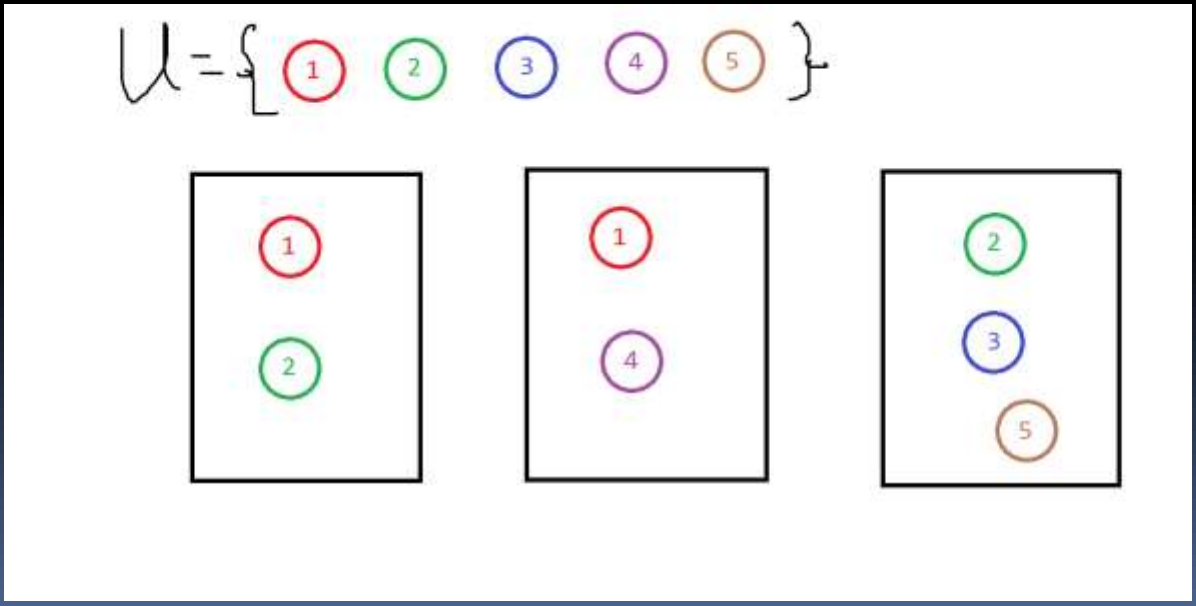
Powerful Blackbox

- Intuition
  - Previously we had short paths
  - This time we have long paths, what now..?
  - If we have some set of vertices Sep, such that many paths of a certain length will include at least one vertex from Sep
  - As paths are long, maybe we can bound $|Sep|$?

Powerful Blackbox

Lemma A.1 (see [TZ05, RTZ05]). *Let $N_1, N_2, \ldots, N_n \subseteq U$ be a collection of subsets of U, with $u = |U|$ and $|N_i| \geq s$ for all $i \in [1, n]$. Then, we can implement a procedure* $\textsc{Separator}(\{N_i\}_{i \in [1,n]})$ *that returns a set A of size at most $O(\frac{u \log n}{s})$ with $N_i \cap A \neq \emptyset$ for all i, deterministically in $O(u + \sum_i |N_i|)$ time.*

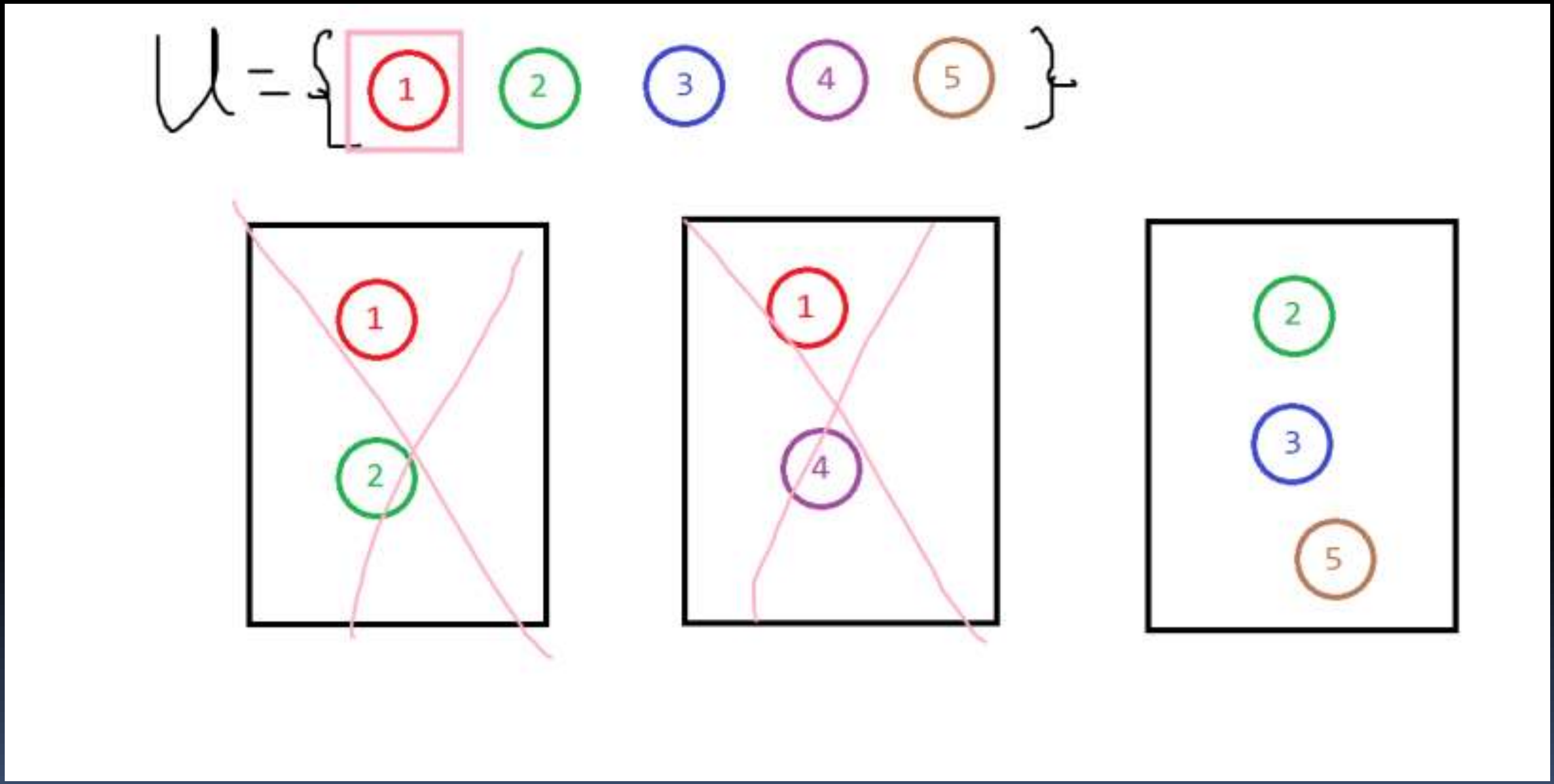
- Example: $u = 5, s = 2, n = 3$



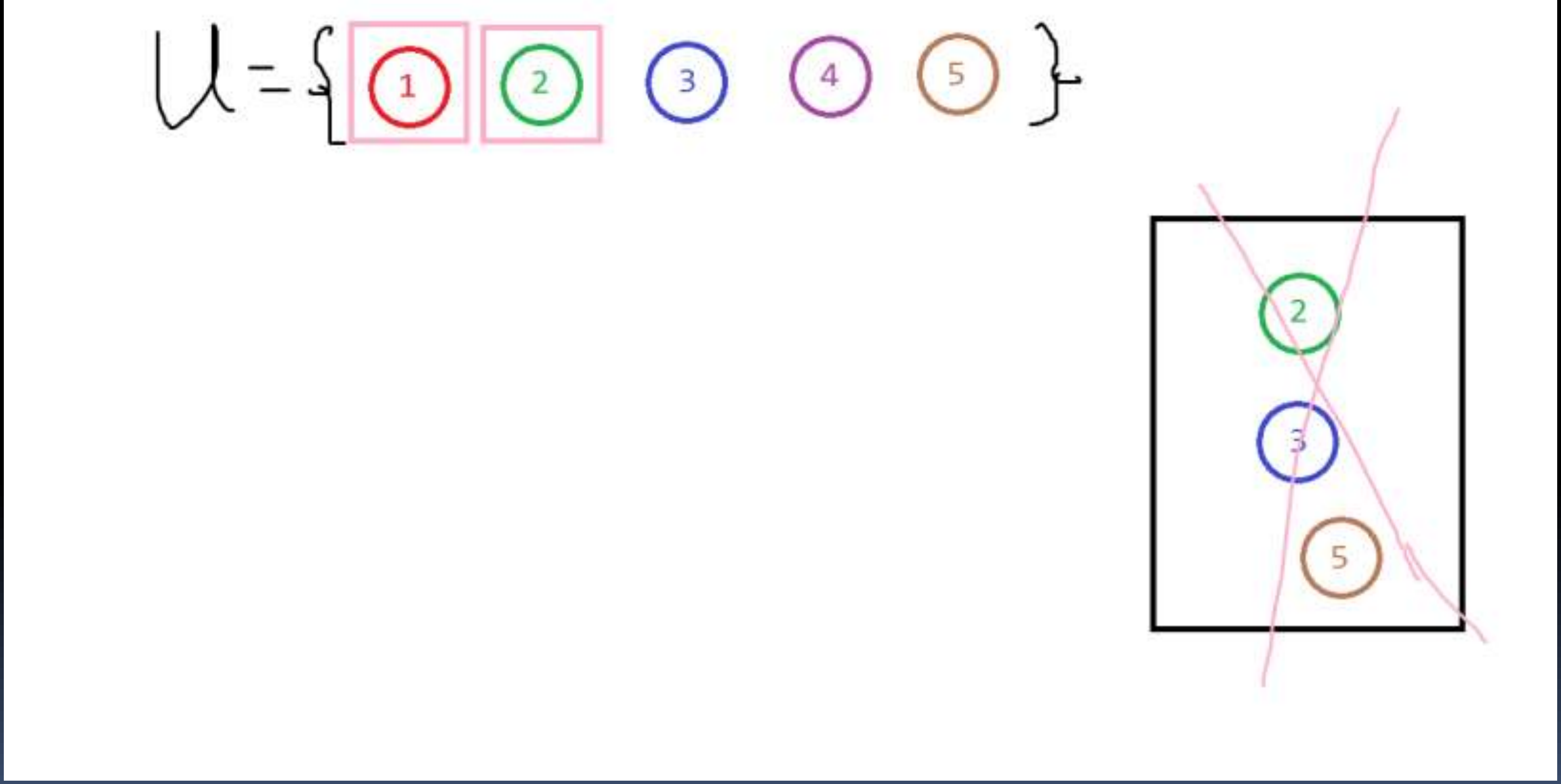
Powerful Blackbox

- Proof of lemma: Just repeatedly pick the element that appears in most sets, then delete all sets that have
- In each step n is multiplied by a factor of at most $(1 - s/u)$ (by pigeon-hole principle)
- We know that $\left(1 - \frac{1}{x}\right)^x < 1/e$ for $x \geq 1$
- So after $\frac{u}{s} \lceil \ln n \rceil$ moves, $n < 1$ and thus we obtained Sep of size $O(\frac{u \log n}{s})$
- It is very cool because the number of sets only contributes to the size by a log factor

Powerful Blackbox



Powerful Blackbox



Powerful Blackbox

- Back to our problem
- Idea
  - Again, we approach the problem in inductive manner
  - In every step, we extend h by a factor of 1.5, so the idea is that for each path that is than current h but shorter than new h, we want to have some vertices in Sep that somewhat is close to the middle of the path
  - We can then run modified floyd-warshall on only vertices in Sep
  - We know that for every path that is a shortest path, any subsegment of it is also a shortest path.

Powerful Blackbox

Algorithm 5: $\textsc{DeterministicExtendDistances}(\Pi = \{\pi_{i_h}(s,t)\}_{s,t}, h)$

Input: A collection of paths Π, that contains a path for each tuple $(s,t) \in V \times V$.

Output: Returns the set of distances $\{(\textsc{Dist}_{i_{max}}(s,t)\}_{s,t \in V \times V}$.

foreach $(s,t) \in V \times V$ **do**
- $\textsc{Dist}_{i_h}(s,t) \leftarrow w(\pi_{i_h}(s,t))$

for $i \leftarrow i_h + 1$ *to* i_{max} **do**
- Compute a set $\textsc{Separator}$ of size $O(n \log n / h_i)$ that contains a vertex from each path in Π of hop at least $\lfloor \frac{1}{4} h_i \rfloor$.;
- **foreach** $(s,t) \in V \times V$ **do**
  - $\textsc{Dist}_i(s,t) \leftarrow \textsc{Dist}_{i-1}(s,t)$;
  - **foreach** $x \in \textsc{Separator}$ **do**
    - $\textsc{Dist}_i(s,t) \leftarrow \min\{\textsc{Dist}_i(s,t), \textsc{Dist}_{i-1}(s,x) + \textsc{Dist}_{i-1}(x,t)\}$

return $\{(\textsc{Dist}_{i_{max}}(s,t)\}_{s,t \in V \times V}$

Hard

Modified Floyd warshall

Powerful Blackbox

- Again, $h_i = 1.5^i$ is the hop size we target for in phase i
- To make sure our algorithm works, we need to figure out these details
  - How to find separator of size $O(\frac{n \log n}{h_i})$ fast?
  - Is it correct? Will we find a shortest path between a pair of nodes if it has hop $\leq h_i$?

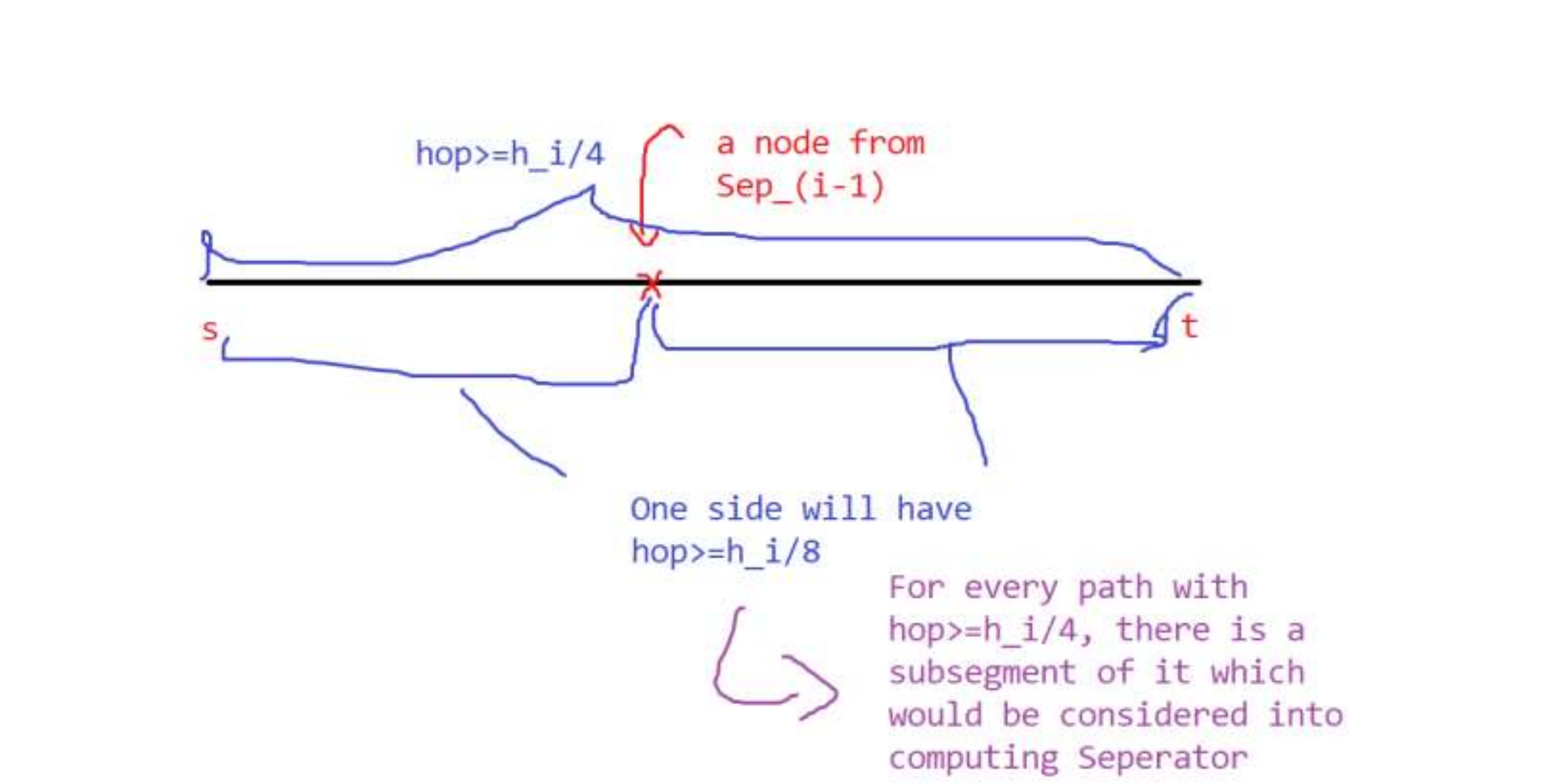
Powerful Blackbox

- How to find Sep of size $O\left(\frac{n \log n}{h_i}\right)$ fast?
- Firstly, we should understand the size and it is straightforward by plugging every shortest path with hop $\geq \lfloor\frac{h_i}{4}\rfloor$ into Lemma A.1
- In the first phase, we will find our Sep by considering all n^2 pairs, so it will take $O(n^2 h)$ time.
- However, we cannot check all n^2 pairs every time or else the complexity will explode

Powerful Blackbox

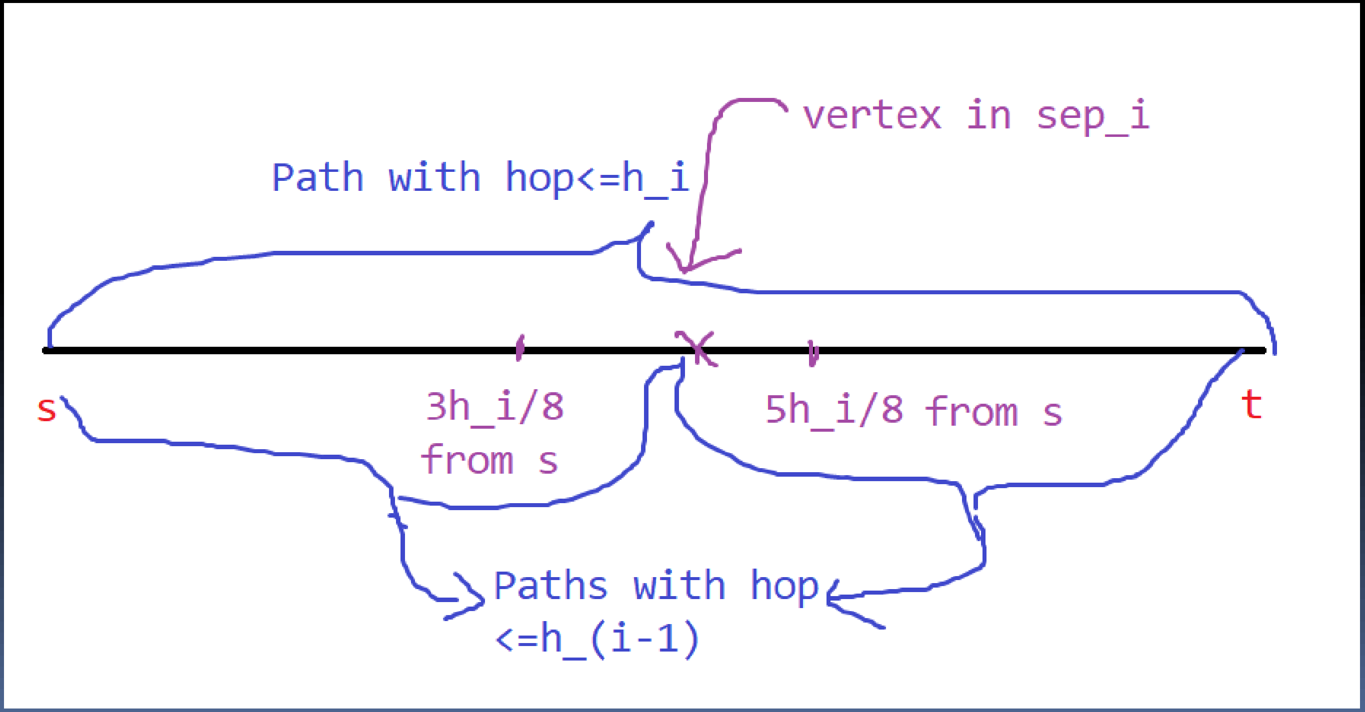
- Lets only check paths that
  - Starts from a vertex in Sep_{i-1} (Sep_{i-1} is the Seperator from previous iteration)
  - Ends from a vertex in Sep_{i-1} (Sep_{i-1} is the Seperator from previous iteration)
- We take paths that are at least $\frac{h_i}{8}$ hop
- Seperator size remains $O(\frac{n \log n}{h_i})$
- Time complexity = $O(\log n)$ phases × $O\left(\frac{n \log n}{h_i} n h_i\right) = O(n^2 \log^2 n)$

Powerful Blackbox



Powerful Blackbox

- Is it correct? Will we find a shortest path between a pair of nodes if it has hop $\leq h_i$?
- Same idea as the one in Delete()



Powerful Blackbox

- Lets analyze complexity
- $O(n^2 h)$ for computing first seperator
- $O(n^2 \log^2 n)$ for computing the later seperators
- $O\left(\frac{n \log n}{h_i} \times n^2\right) = O(\frac{n^3 \log n}{h_i})$ for doing modified floyd warshall
  - Again, analyze with with geometric sum
- $O(\frac{n^3 logn}{h} + n^2 \log^2 n + n^2 h)$ total

Further Optimizations

- Optimizes Time Complexity by combining methods with another paper ACK17
- Optimizes Space Complexity by replacing bellman-ford with an inductive style shortest path (similar as the one in delete), and storing paths in some funny ways