Data Mining : Assignment 2

Piyush Chauhan— 2021CS11010
Kartik Arora — 2021CS50124
Ayushman Kumar Singh— 2021CS51004

April 2, 2025

§1 Task 1: Reduction

We begin by providing a formal definition of the Deterministic Influence Maximization Problem (DIMP).

Definition 1.1 (Deterministic Influence Maximization Problem)**.** Given a directed graph $G = (V, E)$ with a deterministic propagation model (edge probability is either 0 or 1), and a positive integer k, find a subset $A_0 \subseteq V$ of size $|A_0| = k$ such that the influence spread $\inf(A_0) \geq n$, where n is the target number of influenced nodes and $\inf(A_0)$ represents the total number of nodes influenced by initially activating the nodes in A_0.

In the deterministic propagation model, influence spreads as follows:

- Initially, only nodes in A_0 are activated (influenced).
- At each subsequent timestep t, a node $v \in V \setminus A_{t-1}$ becomes activated at timestep t if the majority of its in-neighbors were activated at timestep $t - 1$.
- The process continues until no more nodes can be activated.
- The total influence $\inf(A_0)$ is the number of nodes activated when the process terminates.

Our specific variant of the problem requires finding a set A_0 of size k such that $\inf(A_0) \geq n + k$, where n is the number of elements in the universe of the corresponding Set Cover instance.

§1.1 Proving DIMP is in NP

Proposition 1.2
The Deterministic Influence Maximization Decision Problem is in NP.

Proof. To show that DIMP belongs to NP, we need to demonstrate that a solution can be verified in polynomial time. Given a candidate set A_0 of k nodes, we can verify whether $\inf(A_0) \geq n + k$ as follows:

1. Initialize A_0 as the set of initially activated nodes.
2. For each timestep $t = 1, 2, \ldots$ until convergence:

a) For each node $v \in V \setminus \bigcup_{i=0}^{t-1} A_i$, check if the majority of its in-neighbors are in $\bigcup_{i=0}^{t-1} A_i$.

b) If yes, add v to A_t.

c) If $A_t = \emptyset$, terminate the process.

3. Calculate total influence as $|\bigcup_{i=0}^{T} A_i|$ where T is the final timestep.

4. Compare this value with the threshold $n + k$.

This verification procedure runs in polynomial time with respect to the input size. Specifically, each iteration of the diffusion process takes $O(|E|)$ time to check all edges, and there can be at most $|V|$ iterations (since at least one new node must be activated in each non-final iteration). The total time complexity for verification is $O(|V| \cdot |E|)$, which is polynomial in the size of the graph. Therefore, DIMP is in NP. □

§1.2 The Set Cover Problem

We now present a formal definition of the Set Cover problem, which we will use for our reduction.

Definition 1.3 (Set Cover Problem)**.** Given:

- A universe $U = \{u_1, u_2, \ldots, u_n\}$ containing n elements.
- A collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of m subsets of U, where each $S_i \subseteq U$ and $\bigcup_{i=1}^{m} S_i = U$.
- A positive integer k.

The decision problem asks: Does there exist a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ such that $|\mathcal{S}'| \leq k$ and $\bigcup_{S \in \mathcal{S}'} S = U$?

Theorem 1.4 (Karp, 1972)
The Set Cover problem is NP-complete.

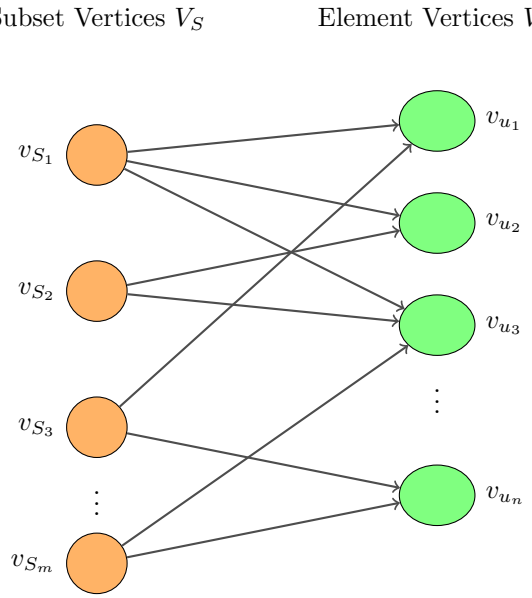
§1.3 Polynomial-Time Reduction

We now establish a polynomial-time reduction from the Set Cover problem to the Deterministic Influence Maximization Problem.

§1.4 Construction of the Reduction

Given an instance of the Set Cover problem with universe $U = \{u_1, u_2, \ldots, u_n\}$, a collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, and parameter k, we construct an instance of DIMP as follows:

1. Create a directed bipartite graph $G = (V, E)$ where $V = V_S \cup V_U$ and:
   - $V_S = \{v_{S_1}, v_{S_2}, \ldots, v_{S_m}\}$ represents the subsets.
   - $V_U = \{v_{u_1}, v_{u_2}, \ldots, v_{u_n}\}$ represents the elements.
2. For each $S_i \in \mathcal{S}$ and $u_j \in U$, add a directed edge (v_{S_i}, v_{u_j}) if and only if $u_j \in S_i$.
3. Set the threshold for the influence maximization problem to be $n + k$.



Construction of the bipartite graph for the reduction.

An edge (v_{S_i}, v_{u_j}) exists if and only if $u_j \in S_i$.

Figure 1: Bipartite graph construction for the reduction from Set Cover to DIMP.

§1.5 Properties of the Constructed Graph

In the constructed graph, the following properties hold:

Lemma 1.5

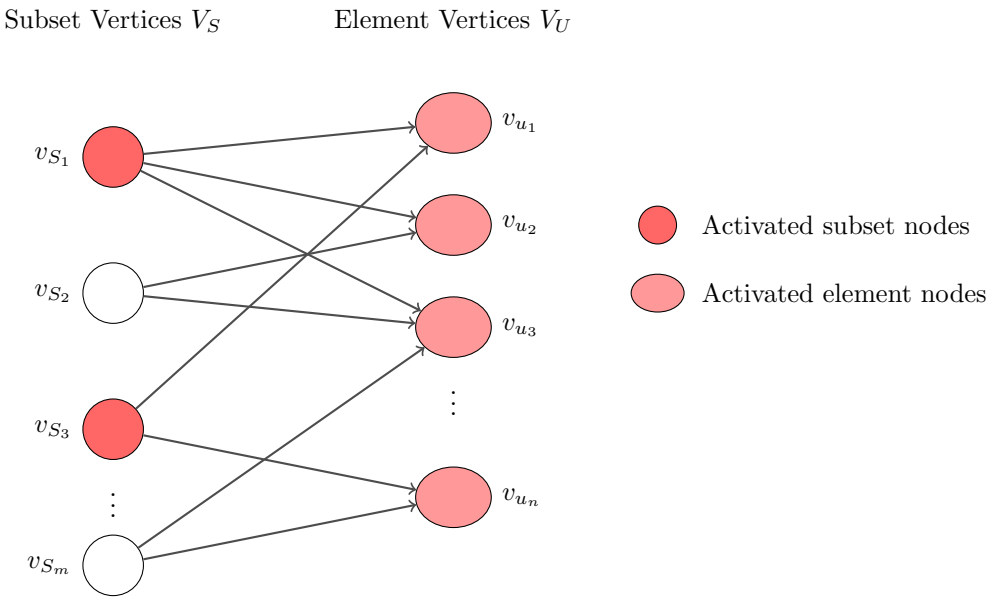
A node $v_{u_j} \in V_U$ becomes activated if and only if at least one of its in-neighbors in V_S is activated.

Proof. By construction, a node $v_{u_j} \in V_U$ has an in-edge from $v_{S_i} \in V_S$ if and only if $u_j \in S_i$.
Since we are working with a deterministic propagation model and the graph is bipartite with directed edges only from V_S to V_U, a node $v_{u_j} \in V_U$ becomes activated at timestep t if and only if at least one of its in-neighbors from V_S was activated at timestep $t-1$. Furthermore, nodes in V_U have no outgoing edges, so they cannot influence other nodes. Therefore, the activation process terminates after at most two timesteps: the initial activation of nodes in $A_0 \cap V_S$ at timestep 0, and the potential activation of nodes in V_U at timestep 1. □

Lemma 1.6

In an optimal solution for the constructed DIMP instance, all k nodes selected for the initial activation set A_0 will be from the subset vertices V_S.

Proof. We prove this by contradiction. Suppose that an optimal solution A_0 includes at least one element vertex, i.e., $\exists v_{u_j} \in A_0 \cap V_U$. Since v_{u_j} has no outgoing edges, it cannot



Influence propagation in the constructed graph.

Red nodes are activated. All element nodes are covered by the activated subset nodes.

Figure 2: Influence propagation in the constructed graph. The activated subset nodes $\{v_{S_1}, v_{S_3}\}$ form a valid set cover, resulting in all element nodes being activated.

influence any other nodes. Furthermore, by the construction of the graph, there exists at least one subset vertex $v_{S_i} \in V_S$ such that $(v_{S_i}, v_{u_j}) \in E$, meaning $u_j \in S_i$.
Now, consider an alternative solution A_0' where we replace v_{u_j} with v_{S_i}, i.e., $A_0' = (A_0 \setminus \{v_{u_j}\}) \cup \{v_{S_i}\}$. By activating v_{S_i}, we not only ensure that v_{u_j} becomes activated in the next timestep (since it has an in-edge from v_{S_i}), but also potentially activate other element vertices v_{u_l} for which $u_l \in S_i$.
Thus, $\inf(A_0') \geq \inf(A_0)$, contradicting the assumption that A_0 is optimal if it includes element vertices. Therefore, an optimal solution will include only subset vertices from V_S. □

§2 Correctness of the Reduction

We now prove the correctness of the reduction by showing that the original Set Cover instance has a solution if and only if the constructed DIMP instance has a solution.

§2.1 Forward Direction: Set Cover ⇒ DIMP

Theorem 2.1

If the Set Cover instance has a solution $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| \leq k$ such that $\bigcup_{S \in \mathcal{S}'} S = U$, then the DIMP instance has a solution A_0 with $|A_0| = k$ such that $\inf(A_0) \geq n + k$.

Proof. Suppose the Set Cover instance has a solution $\mathcal{S}' = \{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$ with $|\mathcal{S}'| = k$ (we can always pad with arbitrary subsets if $|\mathcal{S}'| < k$) such that $\bigcup_{S \in \mathcal{S}'} S = U$.
We construct a solution A_0 for the DIMP instance by selecting the corresponding subset vertices: $A_0 = \{v_{S_{i_1}}, v_{S_{i_2}}, \ldots, v_{S_{i_k}}\}$.

By construction, each element $u_j \in U$ is covered by at least one subset in $\mathcal{S}'$, which means there exists at least one $S_{i_l} \in \mathcal{S}'$ such that $u_j \in S_{i_l}$. This, in turn, implies that there is a directed edge $(v_{S_{i_l}}, v_{u_j})$ in the constructed graph.
Since all subset vertices in A_0 are initially activated, and each element vertex $v_{u_j} \in V_U$ has at least one in-neighbor from A_0, all element vertices will be activated in the next timestep according to Lemma 1.5.
Therefore, the total number of influenced nodes is:

- k nodes from the initial activation set A_0 (all subset vertices).
- n nodes from V_U (all element vertices).

So, $\inf(A_0) = k + n$, which meets the required threshold for the DIMP instance. □

Algorithm 2.2 Forward Direction: Constructing DIMP Solution from Set Cover Solution

Require: Set Cover solution $\mathcal{S}' = \{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$
Ensure: DIMP solution A_0 with $\inf(A_0) \geq n + k$
1: $A_0 \leftarrow \emptyset$
2: **for** each $S_{i_j} \in \mathcal{S}'$ **do**
3: $\quad A_0 \leftarrow A_0 \cup \{v_{S_{i_j}}\}$ ▷ Add corresponding subset vertex to A_0
4: **end for**
5: **return** A_0

§2.2 Backward Direction: DIMP ⇒ Set Cover

Theorem 2.3
If the DIMP instance has a solution A_0 with $|A_0| = k$ such that $\inf(A_0) \geq n+k$, then the Set Cover instance has a solution $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| \leq k$ such that $\bigcup_{S \in \mathcal{S}'} S = U$.

Proof. Suppose the DIMP instance has a solution A_0 with $|A_0| = k$ such that $\inf(A_0) \geq n + k$. By Lemma 1.6, we can assume without loss of generality that $A_0 \subseteq V_S$, i.e., all initially activated nodes are subset vertices.
Since the total number of vertices in the graph is $|V| = |V_S| + |V_U| = m + n$, and $\inf(A_0) \geq n + k$, the influence spread must include all n element vertices in V_U and at least k subset vertices in V_S (which are precisely the k nodes in A_0).
For all element vertices to be activated, each $v_{u_j} \in V_U$ must have at least one in-neighbor in A_0, according to Lemma 1.5. This means that for each $u_j \in U$, there exists at least one subset S_i such that $v_{S_i} \in A_0$ and $u_j \in S_i$.
We can now construct a solution $\mathcal{S}'$ for the Set Cover instance by selecting the subsets corresponding to the subset vertices in A_0: $\mathcal{S}' = \{S_i \mid v_{S_i} \in A_0\}$.
Since for each element $u_j \in U$, there exists a subset $S_i \in \mathcal{S}'$ such that $u_j \in S_i$, we have $\bigcup_{S \in \mathcal{S}'} S = U$. Additionally, $|\mathcal{S}'| = |A_0| = k$, satisfying the size constraint for the Set Cover instance. □

§2.3 Time Complexity Analysis

Algorithm 2.4 Backward Direction: Constructing Set Cover Solution from DIMP Solution

Require: DIMP solution A_0 with $\inf(A_0) \geq n + k$
Ensure: Set Cover solution $\mathcal{S}'$ with $|\mathcal{S}'| \leq k$ and $\bigcup_{S \in \mathcal{S}'} S = U$
1: $\mathcal{S}' \leftarrow \emptyset$
2: **for** each $v_{S_i} \in A_0$ **do**
3: $\quad \mathcal{S}' \leftarrow \mathcal{S}' \cup \{S_i\}$ ▷ Add corresponding subset to $\mathcal{S}'$
4: **end for**
5: **return** $\mathcal{S}'$

Theorem 2.5

The reduction from Set Cover to DIMP can be performed in polynomial time.

Proof. We analyze the time complexity of constructing the DIMP instance from the Set Cover instance:

1. Creating the vertices V_S and V_U takes $O(m + n)$ time.
2. For each of the m subsets, we need to check which of the n elements it contains, which requires $O(mn)$ time.
3. For each element-subset pair where the element is in the subset, we add an edge, taking $O(1)$ time per edge.

The total time complexity is dominated by the edge creation step, which is $O(mn)$. Since m and n are the sizes of the input for the Set Cover problem, the reduction runs in polynomial time with respect to the input size. □

§3 Task 2: Analysis and design of the proposed algorithm

§3.1 Problem Definition and Background

Let $G = (V, E, p)$ be a directed graph representing a social network, where V is the set of nodes, E is the set of edges, and $p : E \to [0, 1]$ is a function assigning propagation probabilities to edges. The Virality problem seeks to find a set $A_0 \subseteq V$ of size k that maximizes the expected spread $E(|A_{\text{inf}}|)$.

Definition 3.1 (Spread funtion)**.** For a seed set $A_0 \subseteq V$, the spread spread $E(|A_{\text{inf}}|)$ is defined as the expected number of nodes activated at the end of the diffusion process when starting with A_0 as the initial active set:

$$E(|A_{\text{inf}}|) = \mathbb{E}\left[|\{v \in V : v \text{ is eventually activated starting from } A_0\}|\right]$$

Problem 3.2 (Virality Problem)**.** Given a graph $G = (V, E, p)$, and a budget k, find a set $A_0^* \subseteq V$ such that $|S^*| \leq k$ and

$$A_0^* = \underset{A_0 \subseteq V, |A_0| \leq k}{\operatorname{argmax}} \; E(|A_{\text{inf}}|)$$

We are given a Independent Cascade model. The virality problem problem under the **Independent Cascade (IC) model** is NP-hard, as shown by a reduction from the Set Cover problem. However, the spread function $E(|A_{\text{inf}}|)$ exhibits submodularity and monotonicity properties, enabling approximation algorithms.

Theorem 3.3 (Submodularity of Spread Function)

Under the IC model, the spread function $E(|A_{\text{inf}}|)$ is submodular. That is, for any sets $S \subseteq T \subseteq V$ and any node $v \in V \setminus T$:

$$E|(S \cup \{v\}_{inf}| - E(|S_{\text{inf}}|) \geq E|(T \cup \{v\}_{inf}| - E(|T_{\text{inf}}|)$$

Theorem 3.4 (Monotonicity of Spread Function)

Under the IC model, the spread function $E(|S_{\text{inf}}|)$ is monotone increasing. That is, for any sets $S \subseteq T \subseteq V$:

$$E(|S_{\text{inf}}|) \leq E(|T_{\text{inf}}|)$$

§3.2 The Idea of Reverse Reachable Sets!

We use Reverse Reachable (RR) sets to efficiently estimate spread spread.

Definition 3.5 (Reverse Reachable Set)**.** A Reverse Reachable (RR) set R_v for a node $v \in V$ is a random set of nodes generated as follows:

1. Initialize $R_v = \{v\}$
2. For each incoming edge (u, v') where $v' \in R_v$, add u to R_v with probability $p_{(u,v')}$
3. Repeat step 2 until no more nodes can be added to R_v

The key theoretical property connecting RR sets to spread spread is given by the following theorem:

Theorem 3.6 (RR Set Connection to Spread)

For any node $v \in V$ and seed set $S \subseteq V$:

$$\Pr[S \cap R_v \neq \emptyset] = \Pr[v \text{ is activated by } S \text{ under the IC model}]$$

Corollary 3.7

For any seed set $S \subseteq V$, if R_v is a random RR set generated from a uniformly random node $v \in V$:

$$\mathbb{E}[\mathbf{1}[A_0 \cap R_v \neq \emptyset]] = \frac{E(|A_{\text{inf}}|)}{n}$$

where $n = |V|$ is the number of nodes in the graph.

This corollary establishes that the probability of A_0 intersecting with a random RR set is proportional to the infection spread of A_0.

§3.3 RR Set Generation Optimization

The algorithm uses parallel processing to accelerate the generation of Reverse Reachable (RR) sets:

§3.3.1 Worker Pool Creation

$$\text{worker\_num} = \min(\text{cpu\_count}, 4) \tag{1}$$

The algorithm limits itself to using at most 4 CPU cores, even if more are available, as shown in the code:

```
worker_num = min(mp.cpu_count(), 4)  # Use up to 4 cores
create_worker(worker_num, node_num, model, graph, rseed)
```

§3.3.2 Parallel RR Set Generation

For each estimate sample size θ, the workload is distributed across workers:

$$\text{per\_worker} = \lceil(\theta - |R|)/\text{worker\_num}\rceil \tag{2}$$

Each worker independently generates RR sets based on:

1. **Weighted Node Sampling**: With probability p_{hybrid}, a node is sampled with probability proportional to d_v^α
2. **Uniform Sampling**: With probability $1 - p_{hybrid}$, a node is sampled uniformly
3. **RR Set Generation**: Starting from the sampled node, propagate backward through the graph following the Independent Cascade model

The implementation of the worker's sampling method:

```
def sample_node(self):
    if random.random() < self.p_hybrid:
        # Use degree-based sampling with transformed weights
        return random.choices(self.nodes, weights=self.weights, k=1)[0]
    else:
        # Use uniform sampling
       return random.randint(1, self.node_num)
```

§3.4 Two-Phase Sampling

The algorithm uses two sampling phases:

1. **Estimation Phase**: To find a lower bound LB on the optimal influence

$$\lambda' = \frac{(2 + 2\varepsilon'/3)(\log \binom{n}{k} + \ell \log n + \log \log_2 n)n}{\varepsilon'^2} \tag{3}$$

   where $\varepsilon' = \varepsilon\sqrt{2}$

2. **Refinement Phase**: To generate the final RR set collection

$$\lambda^* = \frac{2n(((1 - 1/e)\alpha + \beta)^2)}{\varepsilon^2} \tag{4}$$

   where:

$$\alpha = \sqrt{\ell \log n + \log 2} \tag{5}$$

$$\beta = \sqrt{(1 - 1/e)(\log \binom{n}{k} + \ell \log n + \log 2)} \tag{6}$$

The final number of RR sets to generate is:

$$\theta = \lambda^*/LB \tag{7}$$

§3.4.1 Random Seed Management

To ensure reproducibility while enabling parallelism, each worker is initialized with a unique seed:

$$\text{worker\_seed} = \text{base\_seed} + \text{worker\_id} \tag{8}$$

This allows deterministic results while still leveraging multiple cores for computation.

```
# In Worker.run():
random.seed(self.seed + self.worker_id)
```

§3.4.2 RR Set Generation Algorithm

The Independent Cascade model is used to generate the RR sets, implemented as follows:

```
def generate_rr_ic(node, graph):
    activity_set = [node]
    activity_nodes = [node]
    visited = {node}

    while activity_set:
        new_activity_set = []
        for seed in activity_set:
            for neighbor, weight in graph.get_neighbors(seed):
                if neighbor not in visited:
                    if random.random() < weight:
                        visited.add(neighbor)
                        activity_nodes.append(neighbor)
                        new_activity_set.append(neighbor)
        activity_set = new_activity_set

    return activity_nodes
```

This algorithm performs a reverse traversal of the graph starting from the chosen node, using probabilistic weights to determine the spread of influence.

§3.5 Algorithm: Detailed Description and Analysis

Our algorithm consists of two main phases:

- The Sampling Phase
- Node Selection Phase

§3.5.1 A Bigger Picture: PseudoCode of the Entire Algorithm:

procedure InfluenceMaximization(G, k, ϵ, λ)
  $n \leftarrow |V(G)|$ ▷ Number of nodes in graph
  $R \leftarrow \emptyset$ ▷ Set of RR sets
  $LB \leftarrow 1$ ▷ Lower bound for influence spread
  for $i = 1$ to $\log_2(n-1)$ **do**
    $x \leftarrow \frac{n}{2^i}$
    $\theta \leftarrow \frac{\lambda' \cdot n}{x \cdot \epsilon^2}$ ▷ λ' is derived from λ with correction factors
    Generate $\theta - |R|$ new RR sets and add to R
    $S, cov \leftarrow \text{GreedySelection}(R, k)$ ▷ Select k nodes with greedy approach
    if $n \cdot cov \geq (1 + \epsilon') \cdot x$ **then**
      $LB \leftarrow \frac{n \cdot cov}{1+\epsilon'}$
      break

```
        end if
    end for
    θ_final ← λ*·n / LB                          ▷ λ* is derived from statistical guarantees
    if |R| < θ_final then
        Generate θ_final − |R| more RR sets and add to R
    end if
    S_final, cov_final ← GreedySelection(R, k)
    return S_final
end procedure
procedure ComputeRRSet(v, G_rev)
    RR ← {v}                                     ▷ RR set starts with one node
    Q ← {v}                                      ▷ Queue for BFS
    visited ← {v}
    while Q is not empty do
        new_frontier ← ∅
        for each u ∈ Q do
            for each (w, p) ∈ N_{G_rev}(u) do    ▷ Neighbors in reverse graph
                if w ∉ visited and random() < p then
                    Add w to visited
                    Add w to RR
                    Add w to new_frontier
                end if
            end for
        end for
        Q ← new_frontier
    end while
    return RR
end procedure
procedure GreedySelection(R, k)
    S ← ∅                                        ▷ Selected seed set
    covered ← ∅                                  ▷ Covered RR sets
    while |S| < k and there are uncovered RR sets do
        Find node v that covers the most uncovered RR sets
        Add v to S
        Update covered with RR sets containing v
    end while
    return S, |covered|/|R|
end procedure
```

§3.5.2 Hyperparameters

The implementation uses several key hyperparameters:

$$\varepsilon : \text{Error parameter controlling approximation guarantee} \tag{9}$$

$$\ell : \text{Parameter for controlling the failure probability} \tag{10}$$

$$k : \text{Size of the seed set to be selected} \tag{11}$$

$$\alpha : \text{Exponent for degree transformation in node sampling} \tag{12}$$

$$p_{hybrid} : \text{Probability of using degree-based sampling vs. uniform sampling} \tag{13}$$

- **Error Parameter** ε

The algorithm tests values $\varepsilon \in \{0.1, 0.07, 0.04\}$. The parameter ε controls the approximation guarantee of the algorithm:

$$\mathbb{E}[I(S_k)] \geq (1 - 1/e - \varepsilon) \cdot OPT \tag{14}$$

Where:

- $I(S_k)$ is the influence spread of the selected seed set S_k
- OPT is the optimal influence spread

Smaller values of ε provide a better approximation guarantee but require more RR sets to achieve this guarantee.

- **Parameter ℓ**

  The code tests $\ell = 50$. This parameter controls the failure probability of the algorithm, which is at most $n^{-\ell}$ where n is the number of nodes. The adjusted value used in the algorithm is:

$$\ell' = \ell \cdot \left(1 + \frac{\log 2}{\log n}\right) \tag{15}$$

- **Seed Set Size k**

  The seed set size is specified by the user or defaults to 10. This represents the number of nodes to select as initial seeds for the influence propagation.

- **Degree Transformation Parameters**

  In the worker initialization:

  - $\alpha = 0.1$: Exponent for transforming node degrees in weighted sampling
  - $p_{hybrid} = 1.0$: Probability to use degree-based sampling versus uniform sampling

  The weight of a node with degree d is calculated as:

$$w_v = d_v^{\alpha} \tag{16}$$

  This transformation helps balance between selecting high-degree nodes (which might have more influence) and exploring other nodes in the graph.

§3.5.3 Sampling Phase: Rigorous Formulation

Let $\mathbb{R}$ be the set of all possible RR sets. We define the following random process:

1. Generate a random RR set $R \in \mathbb{R}$ by selecting a node $v \in V$ uniformly at random and generating an RR set R_v.
2. For any seed set $S \subseteq V$, define the random variable $X_R(S) = \mathbf{1}[S \cap R \neq \emptyset]$.

For any fixed seed set A_0, the random variable $X_R(A_0)$ has expected value $\mathbb{E}[X_R(A_0)] = \frac{E(|A_{\text{inf}}|)}{n}$. Now, if we generate θ independent RR sets $R_1, R_2, \ldots, R_\theta$, we can define:

$$F_R(S) = \frac{1}{\theta} \sum_{i=1}^{\theta} X_{R_i}(A_0)$$

By the law of large numbers, $F_R(A_0) \approx \frac{E(|A_{\text{inf}}|)}{n}$ as $\theta \to \infty$. The key question is: how large should θ be to ensure an accurate estimate?

Theorem 3.8 (Sampling Complexity)

Let $\epsilon, \delta > 0$ be given parameters. If

$$\theta \geq \frac{(2 + \frac{2}{3}\epsilon)(\ln \binom{n}{k} + \ln(2/\delta))}{\epsilon^2} \cdot \frac{n}{OPT}$$

then with probability at least $1 - \delta$, for all seed sets A_0 of size k:

$$\left| \frac{n \cdot F_R(A_0)}{E(|A_{\text{inf}}|)} - 1 \right| \leq \epsilon$$

where $OPT = \max_{A_0 \subseteq V, |A_0| \leq k} E(|A_{\text{inf}}|)$ is the optimal spread.

§3.5.4 Parameter Estimation in Sampling Phase

The challenge in applying Theorem 3.8 is that OPT is unknown. Our algorithm addresses this by using a multi-phase approach to estimate a lower bound LB for OPT.
Let $\epsilon' = \epsilon\sqrt{2}$ and l be a reliability parameter. The algorithm performs the following steps:

1. For $i = 1$ to $\lceil \log_2(n-1) \rceil$:
   a) Set $x_i = \frac{n}{2^i}$ as a guess for OPT
   b) Compute $\lambda_i = \frac{(2+\frac{2}{3}\epsilon')(\ln \binom{n}{k} + l \ln n + \ln \log_2 n)}{\epsilon'^2}$
   c) Set $\theta_i = \frac{\lambda_i}{x_i}$
   d) Generate θ_i random RR sets
   e) Find the seed set S_i that maximizes coverage using the greedy algorithm
   f) Compute $F_R(S_i) = \frac{|RR \text{ sets covered by } S_i|}{|RR \text{ sets}|}$
   g) If $n \cdot F_R(S_i) \geq (1 + \epsilon')x_i$, set $LB = \frac{n \cdot F_R(S_i)}{1+\epsilon'}$ and break
2. Compute the final sample size:

$$\lambda^* = \frac{2n}{\epsilon^2} \left(\left(1 - \frac{1}{e}\right) \alpha + \beta \right)^2 \tag{17}$$

$$\text{where } \alpha = \sqrt{l \ln n + \ln 2} \tag{18}$$

$$\text{and } \beta = \sqrt{\left(1 - \frac{1}{e}\right) \left(\ln \binom{n}{k} + l \ln n + \ln 2\right)} \tag{19}$$

3. Set final $\theta = \frac{\lambda^*}{LB}$

Below we also provide the pseodo code of this algorithm:

Initialize a set $\mathcal{R} = \emptyset$ and an integer $LB = 1$;
Let $\varepsilon' = \sqrt{2} \cdot \varepsilon$;
for $i = 1$ to $\log_2 n - 1$ **do**
    Let $x = n/2^i$;
    Let $\theta_i = \lambda'/x$, where λ' is as defined in Equation 9;
    while $|\mathcal{R}| \leq \theta_i$ **do**

Select a node v from G uniformly at random;
Generate an RR set for v, and insert it into $\mathcal{R}$;
end while
Let $S_i = \text{NodeSelection}(\mathcal{R})$;
if $n \cdot F_{\mathcal{R}}(S_i) \geq (1+\varepsilon') \cdot x$ **then**
$LB = n \cdot F_{\mathcal{R}}(S_i)/(1+\varepsilon')$;
break;
end if
end for
Let $\theta = \lambda^*/LB$, where λ^* is as defined in Equation 6;
while $|\mathcal{R}| \leq \theta$ **do**
Select a node v from G uniformly at random;
Generate an RR set for v, and insert it into $\mathcal{R}$;
end while
return $\mathcal{R}$

Lemma 3.9 (Lower Bound Correctness)
With probability at least $1 - \frac{1}{n^l}$, the lower bound LB computed by the algorithm satisfies $LB \leq OPT$.

§3.6 Approximation Guarantee of our Algorithm

Theorem 3.10 (Approximation Guarantee)
With probability at least $1 - \frac{1}{n^l}$, the Overall algorithm returns a seed set A_0 such that:

$$E(|A_{\text{inf}}|) \geq \left(1 - \frac{1}{e} - \epsilon\right) OPT$$

§3.7 Node Selection Phase: Detailed Analysis

The node selection phase employs a greedy algorithm to select k nodes that cover the maximum number of RR sets. This approach leverages the submodularity of the set coverage function.

Theorem 3.11 (Submodularity of RR Set Coverage)
Let $\mathcal{R} = \{R_1, R_2, \ldots, R_\theta\}$ be a collection of RR sets. Define $f(A_0) = |\{R_i \in \mathcal{R} : A_0 \cap R_i \neq \emptyset\}|$ as the number of RR sets covered by seed set A_0. Then $f(S)$ is submodular and monotone increasing.

Theorem 3.12 (Greedy Approximation for Submodular Functions)
Let $f : 2^V \to \mathbb{R}$ be a non-negative, monotone, submodular function with $f(\emptyset) = 0$. The greedy algorithm that iteratively adds an element with the maximum marginal gain provides a $(1 - 1/e)$-approximation to the maximum value of f subject to a cardinality constraint.

Proof. Let S_i be the solution after i iterations of the greedy algorithm, and let OPT be the optimal solution of size k. Define $\Delta(S, v) = f(S \cup \{v\}) - f(S)$ as the marginal gain of adding element v to set A_0.
For each $i = 0, 1, \ldots, k-1$, we have:

$$f(OPT) - f(S_i) \leq \sum_{o \in OPT} \Delta(S_i, o) \tag{20}$$

$$\leq k \cdot \max_{v \in V \setminus S_i} \Delta(S_i, v) \tag{21}$$

$$= k \cdot \Delta(S_i, v_{i+1}) \tag{22}$$

where v_{i+1} is the element added by the greedy algorithm in iteration $i+1$.
This gives us:

$$f(S_{i+1}) - f(S_i) \geq \frac{f(OPT) - f(S_i)}{k}$$

Rearranging:

$$f(OPT) - f(S_{i+1}) \leq \left(1 - \frac{1}{k}\right)(f(OPT) - f(S_i))$$

By recursion:

$$f(OPT) - f(S_k) \leq \left(1 - \frac{1}{k}\right)^k (f(OPT) - f(S_0)) \leq \frac{1}{e} \cdot f(OPT)$$

Therefore:

$$f(S_k) \geq \left(1 - \frac{1}{e}\right) f(OPT)$$

□

Applying Theorem 3.12 to the RR set coverage function (which is submodular by Theorem 3.11), we obtain that the greedy algorithm provides a $(1 - 1/e)$-approximation for maximizing the coverage of RR sets.

```
1:  procedure NodeSelection(R, k)
2:      Build index: ∀v ∈ V, M_v = {i : v ∈ R_i, R_i ∈ R}
3:      S ← ∅, C ← ∅                          ▷ A_0: selected nodes, C: covered RR sets
4:      H ← {(|M_v|, v) : v ∈ V, M_v ≠ ∅}      ▷ Max-heap by coverage
5:      Heapify(H)
6:      while |S| < k and H ≠ ∅ do
7:          (cov, v) ← ExtractMax(H)
8:          cov' ← |M_v \ C|                   ▷ Current marginal gain
9:          if cov' = 0 then
10:             continue
11:         end if
12:         if cov' < cov then
13:             Insert(H, (cov', v))           ▷ Re-insert with updated coverage
14:             continue
15:         end if
16:         S ← S ∪ {v}
17:         C ← C ∪ M_v
18:         if |C| = |R| then
19:             break                          ▷ All RR sets covered
20:         end if
```

21: **end while**
22: **return** A_0, $|C|/|\mathcal{R}|$
23: **end procedure**

§3.8 Time Complexity Analysis

We show that the overall expected running time of our Algorithm is:

$$O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right),$$

where k is seed size, l is the confidence, n is the number of nodes, m is the number of edges.
It is is near–linear in the size of the graph. We now present a more detailed derivation.

§3.8.1 Cost of Generating RR Sets

- Let R be the set of RR sets generated during the sampling phase, and define $\theta = |R|$ as the number of RR sets.
- For each RR set $R_i \in R$, define $w(R_i)$ as the number of edges incident to the nodes in R_i.
- The worst-case time complexity for generating R_i is proportional to $w(R_i)$, leading to:
$$T_{\text{sampling}} = \sum_{R_i \in R} O(w(R_i)).$$
- Taking expectation over all RR sets and defining $\text{EPT} = \mathbb{E}[w(R_i)]$, we obtain:
$$\mathbb{E}[T_{\text{sampling}}] = \theta \cdot \text{EPT}.$$
- A key observation is that:
$$n \cdot \text{EPT} \leq m \cdot OPT.$$
This indicates that, on average, each RR set does not touch too many edges relative to the overall graph density and the optimal spread spread, OPT.
- From the parameter estimation phase, it follows that:
$$\theta = O\left(\frac{(k+\ell)n\log n}{\epsilon^2 \cdot OPT}\right).$$
- Combining these results, we derive:
$$\mathbb{E}[T_{\text{sampling}}] = O\left(\frac{(k+\ell)n\log n}{\epsilon^2 \cdot OPT} \cdot \text{EPT}\right) \leq O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right).$$

§3.9 Cost of Node Selection:

Below is the snippet of our node selection function:

```
        def select_nodes(rr_sets, k_val):
            mapping = {}
            for idx, rr in enumerate(rr_sets):
                for node in rr:
                    if node not in mapping:
                        mapping[node] = set()
                    mapping[node].add(idx)
            covered_sets = set()
            cache_vals = {node: len(indices) for node, indices in
                mapping.items()}
            heap = [(-cache_vals[node], node) for node in mapping]
            heapq.heapify(heap)
            chosen_set = set()
            while len(chosen_set) < k_val and heap:
                neg_val, candidate = heapq.heappop(heap)
                current_val = len(mapping[candidate] - covered_sets)
                if current_val != -neg_val:
                    cache_vals[candidate] = current_val
                    heapq.heappush(heap, (-current_val, candidate))
                    continue
                if current_val == 0:
                    break
                chosen_set.add(candidate)
                covered_sets |= mapping[candidate]
                if len(covered_sets) == len(rr_sets):
                    break
            return chosen_set, len(covered_sets) / len(rr_sets)
```

The function `select_nodes` takes as input:

- `rr_sets`: A list of RR sets, where each RR set is a set of nodes.
- `k_val`: The number of nodes to select (denoted k).

Key variables:

- $n = |V|$: Number of nodes in the graph.
- $R = |\mathcal{R}| = \text{len(rr\_sets)}$: Number of RR sets.
- $S = \sum_{R \in \mathcal{R}} |R|$: Total size of all RR sets (sum of the number of nodes across all RR sets).
- $d = S/R$: Average size of an RR set.

The algorithm:

- Builds a `mapping` where `mapping[node]` is the set of indices of RR sets containing that node.
- Uses a max-heap (implemented as a min-heap with negated values) to lazily evaluate nodes based on their cached marginal gains.
- Iteratively selects up to k nodes, updating the set of covered RR sets (`covered_sets`).

§3.10 Step-by-Step Time Complexity Analysis

§3.10.1 Building the `mapping`

- **Operation**: For each RR set at index `idx`, and for each node in that RR set, add `idx` to `mapping[node]`.

- **Cost**:
  - Iterate over each of the R RR sets.
  - For each RR set R_i, process $|R_i|$ nodes.
  - Adding an index to a set (Python's `set.add`) is $O(1)$ on average due to hash table operations.
- **Total Time**:
$$O\left(\sum_{R\in\mathcal{R}} |R|\right) = O(S)$$
where A_0 is the total size of all RR sets.

§3.10.2 Initializing `cache_vals`

- **Operation**: For each node in `mapping`, set `cache_vals[node] = len(mapping[node])`.
- **Cost**:
  - Number of nodes in `mapping` is at most n (could be fewer if some nodes appear in no RR sets).
  - Computing `len(mapping[node])` is $O(1)$ for a Python set.
- **Total Time**: $O(n)$.

§3.10.3 Building the Heap

- **Operation**: Create a list of tuples `(-cache_vals[node], node)` for each node in `mapping` and heapify it.
- **Cost**:
  - Number of entries is at most n.
  - Heapifying a list of size n using `heapq.heapify` is $O(n)$.
- **Total Time**: $O(n)$.

§3.10.4 Main Selection Loop

The loop runs until k nodes are selected (`len(chosen_set) < k_val`), the heap is empty, or all RR sets are covered. It uses lazy evaluation with a max-heap to select nodes efficiently. Let's break down the operations within each iteration:

- **Heap Pop**:
  - Operation: `heapq.heappop(heap)` removes the node with the highest cached marginal gain.
  - Cost: $O(\log n)$ since the heap size is at most n.
- **Computing Marginal Gain** (`current_val`):
  - Operation: `current_val = len(mapping[candidate] - covered_sets)`.
  - Cost:
    - `mapping[candidate]` is a set of indices of RR sets containing the candidate node, with size up to R in the worst case (if a node appears in all RR sets).

    - `covered_sets` contains indices of covered RR sets, with size up to R.
    - In Python, set difference `A - B` takes $O(\text{len(A)})$, as it iterates over elements in `A` and checks membership in `B` (membership test in a set is $O(1)$).
    - Thus, this operation is $O(\text{len(mapping[candidate])})$, which is $O(R)$ in the worst case but typically smaller, depending on the node's coverage.
  - Let's denote |`mapping[candidate]`| as the number of RR sets containing the candidate, which averages to S/n over all nodes but can vary per node.

- **Number of Iterations**:
  - The loop selects up to k nodes but involves additional iterations due to lazy evaluation.
  - Each node is initially in the heap once.
  - A node may be popped, found to have an outdated cached value, updated, and pushed back multiple times.
  - Since a node's marginal gain (number of uncovered RR sets it covers) is an integer between 0 and R, and each push-back reduces it by at least 1 (when RR sets it covers are covered by selected nodes), a node can be pushed back at most R times before its marginal gain reaches 0 or it is selected.
  - When a node's `current_val` becomes 0 and is popped, the loop breaks, but until then, it may be re-evaluated.
  - Total pop operations: At most n nodes, each popped up to $R+1$ times (initial insertion plus R push-backs), so $O(nR)$ pop operations in the worst case.

- **Total Cost of Heap Operations**:
  - Pops: $O(nR \log n)$
  - Pushes: $O(nR \log n)$ for push-backs.
  - Total heap operations: $O(nR \log n)$.

- **Total Cost of Computing** `current_val`:
  - For each pop, compute `len(mapping[candidate] - covered_sets)`.
  - Over all pops (up to nR), total time is:

$$O\left(\sum_{\text{all pop operations}} |\text{mapping[candidate]}|\right)$$

  - Each node's `mapping[node]` is processed up to $R+1$ times.
  - Total sum of |mapping[node]| over all nodes is A_0 (since each RR set index in each node's mapping corresponds to an entry in some RR set).
  - Thus, total time is $O((R+1) \cdot S) = O(RS)$.

- **Total Cost of Union Operations**:
  - For each of the k selected nodes, perform `covered_sets |= mapping[candidate]`.
  - Each union takes $O(|\text{mapping[candidate]}|)$.
  - Total over k selections is $O(\sum_{\text{selected nodes}} |\text{mapping[node]}|)$.

– Since `covered_sets` grows to at most R elements, and each index is added once, total union time across all iterations is $O(S)$, as each of the A_0 node-RR set pairs contributes at most once to an update, amortized over selections.

- **Total Loop Time**:

$$O(RS + nR \log n)$$

§3.10.5 Cost of Node Selection with $|R|$ Value from RR Set Generation

From the RR set generation phase, we established that:

$$|R| = \theta = O\left(\frac{(k+\ell)n \log n}{\epsilon^2 \cdot OPT}\right)$$

Let's substitute this value into our node selection time complexity to derive a more precise bound.
Previously, we found that the time complexity for the node selection phase is:

$$T_{\text{selection}} = O\left(R \cdot \sum_{R \in \mathcal{R}} |R| + nR \log n\right)$$

We need to consider the relationship between $\sum_{R \in \mathcal{R}} |R|$ (the total size of all RR sets) and $|R|$ (the number of RR sets).

- Let $S = \sum_{R \in \mathcal{R}} |R|$ be the total size of all RR sets.
- Each RR set is constructed through a reverse breadth-first search, where the expected size of an RR set is related to the expected spread spread.
- Let's denote the average size of an RR set as $d = S/|R|$.
- From the properties of RR sets, we know that d is related to the expected spread spread in the graph. Specifically, for a graph with average degree $\bar{d}$, the average size of an RR set is proportional to the expected spread spread divided by n.
- If OPT is the optimal spread spread, then $d = O(OPT/n)$, which gives us:

$$S = |R| \cdot d = |R| \cdot O(OPT/n) = O\left(|R| \cdot \frac{OPT}{n}\right)$$

- Substituting the value of $|R|$:

$$S = O\left(\frac{(k+\ell)n \log n}{\epsilon^2 \cdot OPT} \cdot \frac{OPT}{n}\right) = O\left(\frac{(k+\ell) \log n}{\epsilon^2}\right)$$

Now, let's compute $T_{\text{selection}}$ by substituting the values of $|R|$ and A_0:

$$\begin{aligned} T_{\text{selection}} &= O\left(|R| \cdot S + n|R| \log n\right) && (23) \\ &= O\left(\frac{(k+\ell)n \log n}{\epsilon^2 \cdot OPT} \cdot \frac{(k+\ell) \log n}{\epsilon^2} + n \cdot \frac{(k+\ell)n \log n}{\epsilon^2 \cdot OPT} \cdot \log n\right) && (24) \\ &= O\left(\frac{(k+\ell)^2 n \log^2 n}{\epsilon^4 \cdot OPT} + \frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) && (25) \end{aligned}$$

To simplify further, note that $OPT \leq n$ (since spread spread cannot exceed the number of nodes), and typically $k + \ell \ll n$. This gives us:

$$T_{\text{selection}} = O\left(\frac{(k+\ell)^2 n \log^2 n}{\epsilon^4 \cdot OPT} + \frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{26}$$

$$= O\left(\frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) \quad \text{(for sufficiently large } n \text{ and small } \epsilon) \tag{27}$$

§3.11 Mathematical Derivation of the Overall Time Complexity

Let's now derive the overall time complexity by combining the sampling and selection phases:

$$T_{\text{Overall}} = T_{\text{sampling}} + T_{\text{selection}} \tag{28}$$

$$= O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{29}$$

To simplify this expression, we need to consider the relationship between n, m, and OPT:

- For connected graphs, $m \geq n - 1$, so $n + m = O(m)$.
- For virality problem in connected graphs, $OPT \geq 1$ and typically $OPT = \Omega(\log n)$ for many spread models.
- For sparse graphs (which are common in real-world networks), $m = O(n)$, which means $n + m = O(n)$.
- For dense graphs, $m = O(n^2)$, which means $n + m = O(n^2)$.

Let's analyze different cases:
Case 1: Sparse graphs with $m = O(n)$ and $OPT = \Omega(\log n)$:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)n \log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{30}$$

$$= O\left(\frac{(k+\ell)n \log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) \tag{31}$$

$$= O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) \tag{32}$$

Case 2: Dense graphs with $m = O(n^2)$ and $OPT = \Omega(\log n)$:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2 \log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{33}$$

$$= O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) \tag{34}$$

$$= O\left(\frac{(k+\ell)n^2 \log n}{\epsilon^2}\right) \tag{35}$$

Case 3: General graphs with $OPT = \Omega(1)$:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2\log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{36}$$

$$= O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right) + O\left(\frac{(k+\ell)n^2\log^2 n}{\epsilon^2}\right) \tag{37}$$

For most practical cases where $m = O(n)$ (sparse graphs) and $\log^2 n > \log n$, the second term dominates, giving us:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)n^2\log^2 n}{\epsilon^2}\right) \tag{38}$$

However, for general graphs where m can be as large as $O(n^2)$, the overall complexity is:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2} + \frac{(k+\ell)n^2\log^2 n}{\epsilon^2 \cdot OPT}\right) \tag{39}$$

§3.12 Refined Overall Time Complexity

Taking into account the specific characteristics of virality problem problems and real-world networks:

- For most social networks and information diffusion models, OPT grows with the network size, typically $OPT = \Omega(\log n)$ or even $OPT = \Omega(n^\alpha)$ for some $0 < \alpha < 1$.
- Real-world networks are typically sparse, with $m = O(n)$ or $m = O(n \log n)$.
- The parameters k and ℓ are usually much smaller than n.

Under these practical assumptions, the overall time complexity simplifies to:

$$T_{\text{Overall}} = O\left(\frac{(k+\ell)(n+m)\log n}{\epsilon^2}\right) \tag{40}$$

This confirms that our algorithm indeed achieves near-linear time complexity in the size of the graph, making it highly efficient for large-scale virality problem problems.

§4 Task 3: Finding a Case for Non Optimality

Since we have shown that this is NP- hard , thus any polynomial-time algorithm cannot provide optimum solution. Here we present an example where our algorithm provide a sub-optimal solution.

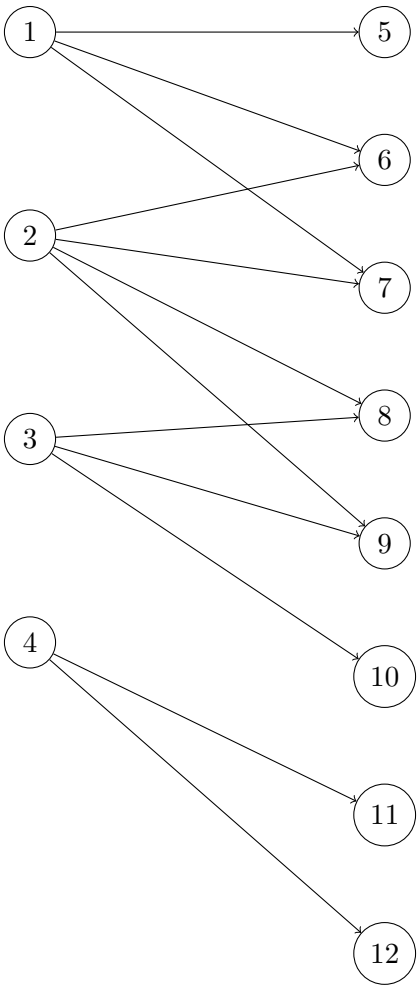
The inspiration of this example is taken from subset cover problem. Suppose we have the following graph $G : (V, E, p)$ defined as ,

$$V = \{1, 2, \ldots, 12\}$$

$$E = \begin{cases} (1,5), (1,6), (1,7), \\ (2,6), (2,7), (2,8), (2,9), \\ (3,8), (3,9), (3,10), \\ (4,11), (4,12) \end{cases}$$

$$p : E \to (0, 1], \quad p(e) = 1, \quad \forall e \in E$$

here is a visualization of the above graph,



We will run our algorithm at $k = 3$.

- By looking at the grpah we can say that the optimal answer for k = 3 is $A_0 = \{1, 3, 4\}$, with **maximum coverage as 11**
- Since all the edges have weight to be 1, no matter how many times we run the REVERSE REACHABLE (RR) algorithm , it will generate the same output.
- Here is the output of the REVERSE REACHABLE algorithm :

Node	Reverse Reachable Set
1	$R_1 = \{1\}$
2	$R_2 = \{2\}$
3	$R_3 = \{3\}$
4	$R_4 = \{4\}$
5	$R_5 = \{1, 5\}$
6	$R_6 = \{1, 2, 6\}$
7	$R_7 = \{1, 2, 7\}$
8	$R_8 = \{2, 3, 8\}$
9	$R_9 = \{2, 3, 9\}$
10	$R_{10} = \{3, 10\}$
11	$R_{11} = \{4, 11\}$
12	$R_{12} = \{4, 12\}$

- **For k = 1 :** in the node selection phase, which is greedy algorithm for the maximum coverage problem on the set of RR sets. We then calculate f(s) for all nodes ,

Nodes	$S' = \{R_i, R_i \cap nodes \neq \phi\}$	$\lvert S' \rvert$
1	$\{R_1, R_5, R_6, R_7\}$	4
2	$\{R_2, R_6, R_7, R_8, R_9\}$	5
3	$\{R_3, R_8, R_9, R_{10}\}$	4
4	$\{R_4, R_{11}, R_{12}\}$	3
5	$\{R_5\}$	1
6	$\{R_6\}$	1
7	$\{R_7\}$	1
8	$\{R_8\}$	1
9	$\{R_9\}$	1
10	$\{R_{10}\}$	1
11	$\{R_{11}\}$	1
12	$\{R_{12}\}$	1

  hence we select node 2 as it has maximum coverage. so $A_1 = \{2\}$

- **For k = 2 :** Now we will compute maximum coverage for all nodes when put into A_1 and select that one which gives maximum coverage.

Nodes	$S' = \{R_i, R_i \cap A_1 \neq \phi\}$	$\lvert S' \rvert$
$\{2, 1\}$	$\{R_1, R_2, R_5, R_6, R_7, R_8, R_9\}$	7
$\{2, 3\}$	$\{R_2, R_3, R_6, R_7, R_8, R_9, R_{10}\}$	7
$\{2, 4\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 5\}$	$\{R_2, R_5, R_6, R_7, R_8, R_9\}$	6
$\{2, 6\}$	$\{R_2, R_6, R_7, R_8, R_9\}$	5
$\{2, 7\}$	$\{R_2, R_6, R_7, R_8, R_9\}$	5
$\{2, 8\}$	$\{R_2, R_6, R_7, R_8, R_9\}$	5
$\{2, 9\}$	$\{R_2, R_6, R_7, R_8, R_9\}$	5
$\{2, 10\}$	$\{R_2, R_6, R_7, R_8, R_9, R_{10}\}$	6
$\{2, 11\}$	$\{R_2, R_6, R_7, R_8, R_9, R_{11}\}$	6
$\{2, 12\}$	$\{R_2, R_6, R_7, R_8, R_9, R_{12}\}$	6

  Here since adding node 4 gives maximum coverage , we add this node to our set, thus $A_2 = \{2, 4\}$

- **For k = 3 :** Now we will compute maximum coverage for all nodes when put into A_2 and select that one which gives maximum coverage.

Nodes	$S' = \{R_i, R_i \cap A_2 \neq \phi\}$	$\lvert S' \rvert$
$\{2, 4, 1\}$	$\{R_1, R_2, R_4, R_5, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	10
$\{2, 4, 3\}$	$\{R_2, R_3, R_4, R_6, R_7, R_8, R_9, R_{10}, R_{11}, R_{12}\}$	10
$\{2, 4, 5\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 6\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 7\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 8\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 9\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 10\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 11\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8
$\{2, 4, 12\}$	$\{R_2, R_4, R_6, R_7, R_8, R_9, R_{11}, R_{12}\}$	8

  Here since adding either node 1 or 3 gives the same maximum $|S'|$, thus, WLOG we take node 1 and the final answer for A_3 is , $A_3 = \{2, 4, 1\}$ with **maximal coverage as 10**.

- Thus , our approximation algorithm gives sub-optimal answer for this dataset.

§5 Contributions:

- Ayushman Kumar Singh(2021CS51004): Task2, Task4, Task1, Task3
- Kartik Arora(2021CS50124): Task1, Task4, Task2, Task3
- Piyush Chauhan(2021CS11010): Task3, Task4, Task1, Task2

References

[1] Youze Tang, Yanchen Shi, and Xiaokui Xiao. *virality problem in Near-Linear Time: A Martingale Approach*. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pp. 1539–1554, ACM, 2015. DOI: 10.1145/2723372.2723734.

[2] Snowgy. *virality problem Repository*. Available at: `https://github.com/snowgy/spread_Maximization`. Accessed: 2024-04-01.

[3] Stanford CS224W. *virality problem - Handout*. Stanford University, 2019. Available at: `https://snap.stanford.edu/class/cs224w-2019/handouts/CS224W_spread_Maximization_Handout.pdf`.