Maximum Coverage with Path Constraint

Anna Fariha, Larkin Flodin, Raj Kumar Maity

1 Introduction

The maximum coverage problem is a classical and fundamental problem in the study of approximation algorithms. In the classical unweighted version of the maximum coverage problem, we are given a universe $\mathcal{U} = \{e_1, e_2, \ldots, e_n\}$ of elements, a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, where each S_i *covers* certain elements of $\mathcal{U}$, i.e., $S_i \subseteq \mathcal{U}$, and an integer k. The goal is to find $S' \subseteq \mathcal{S}$ such that $|S'| \leq k$ and the number of covered elements, i.e. $|\bigcup_{S_i \in S'} S_i|$, is maximized. The maximum coverage problem is NP-hard, but there is a greedy algorithm with approximation factor $1 - \frac{1}{e} + o(1)$. Several variants of the maximum coverage problem have been studied in the approximation algorithms literature and we highlight a few notable ones below:

Weighted version [16]: In the weighted version of the maximum coverage problem, each element $e_i \in \mathcal{U}$ has a weight $w(e_i)$, and the objective is to maximize the total weights of the covered elements, i.e. $\sum_{e_i : \exists S_j \in S' \; e_i \in S_j} w(e_i)$.

Budgeted version [11]: Instead of simply a constraint on the maximum number of sets to pick, in the budgeted version the constraint is on the *cost* of the picked sets. Similar to the weighted version, each element e_i has a weight $w(e_i)$, but additionally, each set S_j has an associated cost $c(S_j)$. The modified constraint states that the total cost of the picked sets, $\sum_{S_j \in S'} c(S_j)$, must not exceed a predefined threshold (the budget). The objective is the same as the weighted version, to maximize the total weight of the covered elements, while not exceeding the budget for picking sets.

Generalized version [5]: In the generalized maximum coverage problem, unlike the aforementioned variations, the weight of an element is not predetermined, but rather depends on which set covers it. The objective remains the same as the budgeted version: to maximize the total weight of the covered elements, while keeping the cost within budget. An additional constraint in this version is that each element can be covered by at most one set. This avoids any ambiguity regarding determining the weight of a covered element, in case it is covered by multiple sets.

In all of the variations discussed above, and in all other variations (see Section 2), there is no restriction on which set we can pick next depending on what has already been picked. However, in many applications such a constraint is necessary, as when we pick a set, which other sets are available to be picked next becomes restricted.

In this project, we are interested in studying the maximum coverage problem with a constraint on the *feasibility* or *compatibility* of the picked sets. Informally, we would like to have a constraint of the form "S_i can/cannot be picked after S_j is picked". If we assume that an iterative algorithm picks sets sequentially, then the possible choices for the next set to be picked depend on the set that was picked in the last iteration. We motivate our problem with a real-world application of code coverage.

1.1 Code Coverage: A Real-World Application

One goal of program testing is to make sure all parts of the source code are executed at least once. *Code coverage* [13], or *test coverage*, is a measure describing to what degree the source code of a program is executed when a test is performed. In software engineering, code is tested using several test input combinations. Depending on the input combination, different branches are taken and different portions of the source code are invoked. However, once the input is fixed, a deterministic program always follow a particular sequential control flow path, i.e., which statements can be executed next depends on the previous statement. A program can follow different control flow paths depending on its input. Of course, if the developer could exhaustively test the system with all possible combinations of the input data, she would get 100% code coverage. However, often there is a budget on the amount of tests developers are allowed to perform, due to the potentially infinite space of possible inputs.

We can build a *control flow graph (CFG)* for a program where each vertex denotes a function call. Note that since a program control flow cannot traverse back in time, a CFG must be a *directed acyclic graph (DAG)* which encodes all possible control flow paths a program can take during execution. We

```
def Main(A, B, C):
    if A > 0:
        FuncX(B, C)
    else:
        FuncY(B)
    FuncZ()
```

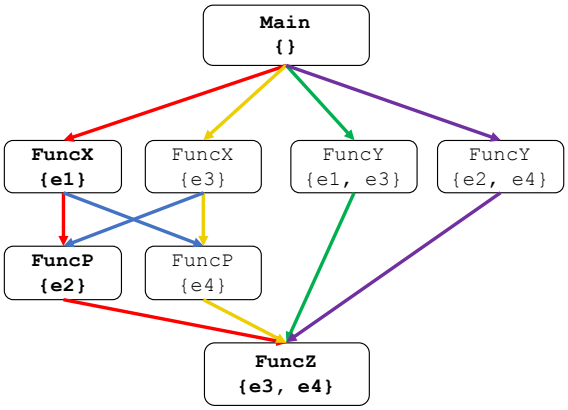
```
def FuncX(B, C):
    if B > 0:
        do_stuff_1()  // e1
    else:
        do_stuff_3()  // e3
    FuncP(C)
```

```
def FuncP(C):
    if C <= 0:
        do_stuff_2()  // e2
    else:
        do_stuff_4()  // e4
```

```
def FuncZ():
    do_stuff_3()      // e3
    do_stuff_4()      // e4
```

```
def FuncY(B):
    if B > 0:
        do_stuff_1()  // e1
        do_stuff_3()  // e3
    else:
        do_stuff_2()  // e2
        do_stuff_4()  // e4
```

(a)



(b)

Path	Variable Assignment	Atomic Segment Coverage	Code Coverage (%)
→	A > 0 B > 0 C <= 0	{e1, e2, e3, e4}	100%
→	A > 0 B <= 0 C > 0	{e3, e4}	50%
→	A <= 0 B <= 0	{e2, e3, e4}	75%
→	A <= 0 B > 0	{e1, e3, e4}	75%
...	...	...	...

(c)

Figure 1: Example of a code coverage scenario: (a) source code, (b) control flow graph (c) a few possible paths, the corresponding input combinations, and their code coverages. `do_staff_n` denotes the n-th atomic segment. The variable assignment $A > 0, B > 0$ and $C \leq 0$ yields 100% code coverage, and the corresponding control flow path is shown in red.

consider the *atomic segments* of a program as the elements which need to be covered. An atomic segment could be a single statement or a collection of statements, but they must always run together; statements within an atomic segment are either all executed, or none.

Figure 1 shows a sample code coverage scenario. The program takes different control flow paths based on the values of the three input parameters. The path that has highest code coverage is the red one, which yield 100% code coverage. In this figure, the variable assignments $A > 0, B > 0$, and $C \leq 0$ yields the maximum code coverage, which covers all four atomic segments — `e1, e2, e3,` and `e4`. There are 8 possible combinations of values for these three variables, even when we limit the domain for each variable to only two possible outcomes (positive or non-positive). In practice, the number of possible combinations of input data can easily explode. Finding the best combination of input parameters, i.e. the one that maximizes code coverage, is an NP-hard problem and the time required grows exponentially as the domain size of the parameters or the number of parameters increases.

In the code coverage problem, if we can determine a *path* in the CFG (Figure 1 (b)) that maximizes the code coverage, we can easily figure out the corresponding input combination which guarantees maximum code coverage. This motivates us to formulate our problem, *maximum coverage with path constraint*. We now describe how we modify the maximum coverage problem to model the code coverage maximization problem. Contrasting with the classical maximum coverage problem, instead of a set of sets $\mathcal{S}$, we start with a CFG DAG. We associate a function call with each vertex of the DAG. Each function call can be thought of as a set that contains (covers) all the atomic segments it executes. A function call may have multiple instances depending on its coverage. Then our goal is to find a path P in this DAG that maximizes the number of atomic segments executed if the program follows the control flow path P.

Unlike the classical maximum coverage problem, which puts a constraint on the number of sets that can be picked, we put a constraint that *the picked sets must comprise a path in the given DAG*. Thus, the ability to choose a set is dependent on which other sets it is connected to in the DAG. We contrast

Problem	Algorithm	Approximation
Unweighted	Greedy	$1 - \frac{1}{e}$
Weighted	Greedy	$1 - \frac{1}{e}$
Budgeted	Greedy + some Extension	$1 - \frac{1}{e}$
Generalized	Greedy + some Extension	$1 - \frac{1}{e} + o(1)$
Group Budget	Greedy in Oracle Model	$\frac{1}{2}$

Table 1: Approximation results of related works.

the semantics of our problem with the classical maximum coverage problem below:

- In the classical maximum coverage problem, sets can be picked in any arbitrary order. But in our problem, the order matters.
- In the classical maximum coverage problem, picking one set does not affect the freedom to pick another set. But in our case, the last picked vertex controls the freedom of choice for the next vertex.
- In the classical maximum coverage problem, the constraint is applied on the number of sets that are ultimately picked. In our problem there is no limit on the number of sets that can be picked. However, our constraint comes from the path requirement, i.e., we require that the selected sets must form a valid path in the DAG.

Beyond the aforementioned code coverage application, there are numerous other applications of the maximum coverage with path constraint problem. Whenever there are constraints on what sets can be picked next given some set is picked, we can encode such constraints using directed edges of a graph. A few other applications are: (1) maximizing influence propagation in social networks, (2) wireless sensor networks where each sensor requires communication with other sensors and the communication is only possible between certain sensors, etc.

In this project, we define a new problem: *maximum coverage with path constraint*, which has application to many practical areas. We provide a greedy algorithm for the problem and show that it is a $1/f$ approximation where the maximum number of times an element can occur in a path is at most f. We have also explored some other techniques to solve the problem and we explain them in this report.

The rest of the report is organized as follows: Section 2 describes the related work. In Section 3, we formalize the problem statement and show that the problem is NP-hard by showing a polynomial time reduction from the classical maximum coverage problem. In Section 4, we describe the pre-processing steps that is required for the greedy $1/f$ approximation which we provide in Section 5. Section 6 describes how we can apply linear program based methods to solve our problem and the challenges in doing so. In Section 7, we discuss some alternative approaches to attack the problem. Finally, we conclude in Section 8.

2 Related Work

There exists some work on set cover with path constraint for informative path planning [1] already. However, to the best of our knowledge, the maximum coverage problem with *path constraint* is studied for the first time here. The classical maximum coverage problem [10] is heavily studied in the field of computer science, complexity theory, approximation algorithms and operations research. For the past few decades, a handful of variants of the problem have been studied in the sense of computational hardness, approximation factors, and developing algorithms. The maximum coverage problem is NP-hard and a greedy approach gives an approximation factor of $1 - \frac{1}{e}$, which is the same approximation ratio that can be achieved by the generic greedy algorithm used for *maximization of the sub-modular function with cardinality constraint* [14].

The other variants — *weighted* [16], *budgeted* [11], and *generalized* [5] versions — are briefly described in Section 1. All of the variants achieve the same approximation factor of $1 - \frac{1}{e}$ using greedy approaches or some extended version of the greedy algorithm. In the group budget variation [4], the collection of sets is partitioned into different groups and the maximum coverage problem is solved under the constraint that at most one set can be picked form each group. A (1/2)-approximation is devised in their work. In Table 2, we summarize the results known for different variants of the problem.

The hardness of the maximum coverage problem and the budgeted version has also been of interest in the approximation algorithms community. A lower bound (assuming P $\neq$ NP) for the approximation factor of the maximum coverage problem is $(1-\frac{1}{e})$, shown in the work of Uriel Feige [6], and this is tight, as this approximation factor is achieved by using a greedy algorithm. The maximum coverage with path constraint problem is NP-Hard as we will show in Section 3, and the reduction also yields a lower bound of $(1-\frac{1}{e})$ for our problem. Achieving a better bound than this, if possible, would be of utmost interest to us.

Although somewhat different, some related work on approximating longest directed paths and cycles has been studied in [2]. The generalized budgeted submodular set function maximization has been studied in [3]. Other variants of the maximum coverage problem include maximum coverage over a matroid [7], partial sublinear time approximation and inapproximation results for maximum coverage [8], and maximum coverage in the streaming model [15].

3 Problem Statement

In this section, we first provide the problem definition. Then we prove that the problem is NP-hard by showing a reduction from the classical unweighted maximum coverage problem.

Definition 1 (Maximum coverage with path constraint)**.** *Given a directed acyclic graph $G = (V, E)$ and a set $\mathcal{U} = \{e_1, \ldots, e_n\}$, where each $v \in V$ covers a subset $S_v \subseteq \mathcal{U}$, the objective is to find the best path $P = \langle v_i, ..., v_n \rangle$ in the DAG such that $|S_P|$ is maximized, where $S_P = \bigcup_{v \in P} S_v$.*

Theorem 1. *The maximum coverage with path constraint problem is NP-hard.*

Proof. It is known that the unweighted maximum coverage problem is NP-hard. We provide a polynomial time reduction from an instance of the unweighted maximum coverage problem to the maximum coverage with path constraint problem. Given the sets $\{S_1, \ldots, S_m\}$ and the integer k, we create a DAG with k topological levels. At each level we put all sets S_i, $1 \leq i \leq m$, as vertices. We put a directed edge from every vertex at level $1 \leq j < k$ to all vertices at level $j + 1$. This completes the construction, and next we argue the correctness of the reduction.

Now in the maximum coverage with path constraint problem, the goal is to select a path from this DAG that maximizes coverage. Our reduction ensures that exactly one vertex is picked at each level and exactly k total vertices are picked. However, since it is possible to pick the same vertex multiple times, the total number of distinct vertices picked can be at most k. Therefore, a feasible solution in this problem is also a feasible solution of the unweighted maximum coverage problem. Despite the path constraint, this reduction ensures that all of the $\binom{m}{k}$ possible subsets can be picked from the DAG in such a way that they form a path. Hence, if we can solve the maximum coverage with path constraint problem on this DAG, the set of vertices on the optimal path must be the optimal solution of the unweighted maximum coverage problem. To see why, consider the optimal solution of the unweighted maximum coverage problem. Since the optimal solution does not contain more than k sets, and any subset of the m sets having cardinality at most k forms a valid path in the DAG, it must be picked as an optimal solution for the maximum coverage with path constraint problem. Therefore, we conclude that the path constrained version is at least as hard as the unweighted maximum coverage problem and hence it is also NP Hard. □

Note also that the above reduction is approximation-preserving, so in addition to showing that the path constrained version is NP-hard, by combining with the result of [6] which shows a lower bound of $(1 - \frac{1}{e})$ for unweighted maximum coverage, we have shown that this lower bound holds for the path-constrained problem as well.

4 Pre-processing

To more easily describe our approximation algorithms and linear programming formulations, we transform the DAG in the following way:

- We introduce a super-source s and a super-sink t in the DAG. We add directed edges from s to all $v \in V$ and from all $v \in V$ to t. s and t themselves cover no elements. The transformation adds 2 extra vertices and $2|V|$ extra edges in the DAG.
- Instead of the scenario where each vertex covers some elements, we push the coverage information to the edges. Specifically, we push coverage of each vertex to all the outgoing edges from that vertex. As we pick any vertex and move from it through an outgoing edge to the next vertex in

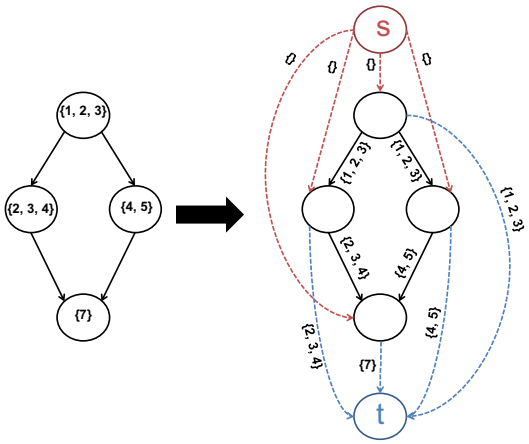


Figure 2: Demonstration of pre-processing of the DAG by introducing super-source and super-sink vertices and pushing the sets from vertices to their outgoing edges.

a path, we are still covering all the elements that the vertex used to cover, by taking one of its outgoing edges.

When introducing a super-source and super-sink, we also impose a constraint on the initial and terminal vertex of the path. Instead of any path, we now require that the path must have s as its initial vertex and t as its terminal vertex. This still allows for any path in the original graph to be chosen, since one can start from s and jump to any vertex in the DAG, and end the path by jumping from any vertex to t. Since the source and sink do not cover any elements, the constraints of the problem remain unchanged.

Moreover, by adding an extra $2|V|$ edges we do not blow up the problem in a meaningful way, since the number of additional edges is polynomial in $|V|$. To write our problem as an LP, we are interested in using a flow constraint (Section 6), and we aim to use the choice of edges as variables and cover the elements using edges of any chosen path. We take care of that in the second step by pushing the set of covered elements from the vertex to its outgoing edges. Since jumping out from any vertex requires picking one of its outgoing edges, and we impose that every path must finish at the super-sink t, we still cover all the elements that were covered by that vertex originally. We show an illustration of this transformation in Figure 2.

5 Greedy (1/f)-Approximation

In this section, we will make use of the pre-processing described in the previous section to obtain an approximation with quality that depends on the maximum "frequency" with which any one element appears in multiple sets. The algorithm and analysis are very similar to the well-known $(1/f)$-approximation for unweighted set cover; in that context f represents the maximum number of times any element of the universe appears in a set, but we will see that in our context we can use a slightly different f which is better in some cases. Without further ado, let's describe the algorithm in question given a pre-processed DAG G:

1. Construct a weighted graph G' with the same vertices and edges as G. For each edge e, let $w(e) = |S_e|$, where S_e is the set on edge e in G.
2. Find the longest weighted path in G'. Return that path.

It is clear that the returned solution is feasible, as it is a path in G. The time complexity of the algorithm is $O(|V|+|E|)$, as we can find the longest path in a weighted DAG using dynamic programming in this time. The following simple argument bounds the quality of the approximation.

Theorem 2. *Let f be the maximum number of times any element of $\mathcal{U}$ appears on a path in G. Then the algorithm described above is a $(1/f)$-approximation for maximum coverage with path constraint.*

Proof. Suppose the path P returned by the above algorithm covers C elements and has weight W in G', and the optimal path P^* covers C^* elements and has weight W' in G'. W is the number of elements P would cover if every set in the path were disjoint, so certainly we have $C \leq W$. The same is true of the optimal path, so $C^* \leq W'$, and also $W' \leq W$ as P is the longest path in G'.

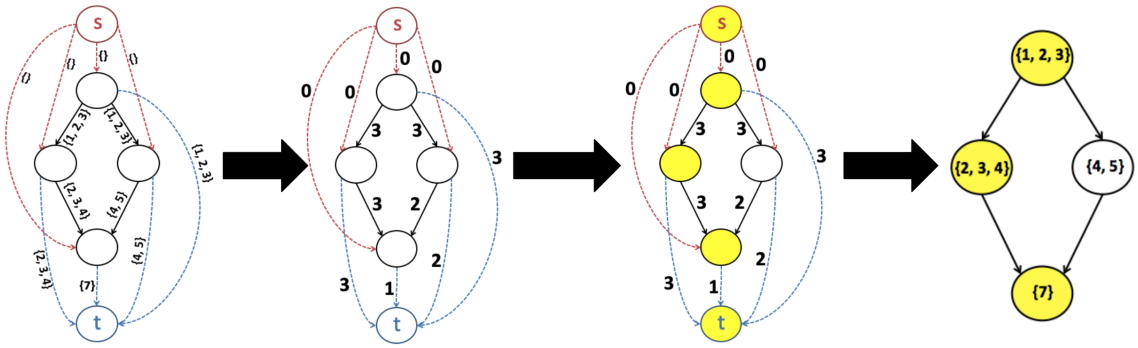


Figure 3: An example run of the $(1/f)$-approximation. Vertices colored yellow indicate those in the longest path. The approximation returns a path covering 5 elements, while the optimal path covers 6 elements.

Now consider how small C can be – as each element of $\mathcal{U}$ appears at most f times in any path, it appears at most f times in P, so at a minimum we have $f \cdot C \geq W$. Combining this with the above, we have:

$$f \cdot C \geq W \geq W' \geq C^*,$$

which shows that our algorithm gives a $(1/f)$-approximation.

□

6 Linear Program-Based Methods

Another direction we have considered is to model our problem as an integer linear program (ILP), then relax it to an LP, solve it, and try to somehow round the fractional solution to an integral one in a way that approximately preserves the solution quality. In writing the ILP, it will be convenient to make use of the graph pre-processing described in Section 4, and assume that sets correspond to edges, and that the graph has a super-source s and super-sink t. We will have a variable y_j for each element $u_j \in \mathcal{U}$ that is equal to 1 iff element u_j is being covered, and a variable x_e for each edge e that is equal to 1 iff the set S_e corresponding to that edge is being used in the path. The full ILP is written below:

$$\text{max.} \quad \sum_{\substack{j \\ u_j \in \mathcal{U}}} y_j \tag{1}$$

$$\text{s.t.} \quad \sum_{\substack{e \\ e=(s,v_i)}} x_e = 1 \tag{2}$$

$$\sum_{\substack{e \\ e=(v_i,t)}} x_e = 1 \tag{3}$$

$$\sum_{\substack{e \\ e=(v_i,v_k)}} x_e - \sum_{\substack{e \\ e=(v_k,v_i)}} x_e = 0 \qquad \forall v_k \in V \setminus \{s,t\} \tag{4}$$

$$\sum_{\substack{e \\ u_j \in S_e}} x_e \geq y_j \qquad \forall u_j \in \mathcal{U} \tag{5}$$

$$x_e \in \{0,1\} \qquad \forall e \in E \tag{6}$$

$$y_j \in \{0,1\} \qquad \forall u_j \in \mathcal{U}. \tag{7}$$

The objective is simply to maximize the coverage. The constraints in (2), (3), (4) are similar to flow constraints; they enforce that exactly 1 edge is chosen leaving the source, 1 edge is chosen going in to the sink, and at every other vertex, if an incoming edge is chosen, then an outgoing edge must be chosen as well. Constraint (5) enforces that if u_j is being covered, we must use some edge e in the chosen path with $u_j \in S_e$.

This ILP represents our problem exactly, but of course is not tractable to solve. We can, however, relax constraints (6), (7) to their linear equivalents, and then solve the resulting LP. The question then becomes what to do with the fractional values that may arise in the LP solution, as we need our output to represent a single path in G. This is where we need to bring in techniques for rounding fractional LP solutions to integral ones.

6.1 Rounding Techniques

Before diving into the specifics of rounding techniques, a useful observation to make is that for our problem, we only need to worry about rounding the x_e variables. Our objective is to maximize a sum of y_j variables, but these variables are only bounded above by the constraints in (5) and (7), so once we have rounded all x_e variables to integral, we can just set

$$y_j = \min(1, \sum_{\substack{e \\ u_j \in S_e}} x_e), \tag{8}$$

and then all y_j variables will also be integral. This makes sense intuitively, as once the x_e variables are integral, then they specify a particular path in G, and of course we ought to set the y_j variables such that we are "getting credit" for all u_j actually covered by the path in question.

The first technique to consider is whether we can deterministically round a fractional solution to an integral one. Typically the way this type of rounding is employed is to proceed one fractional variable at a time, and round it to an integer in such a way that no constraints are violated and the loss in objective value is somehow bounded. Unfortunately, this type of approach does not seem to work for our problem; the issue is the "flow constraints" in (2), (3), (4). Because these are equality constraints, we cannot adjust the value of fractional variables one at a time without violating constraints. In order to avoid violating these constraints, we need to find a way of adjusting many fractional variables simultaneously.

The next technique to try, which is useful in obtaining approximation algorithms for many NP-hard problems, is known as "independent rounding." Instead of deterministically rounding the variables, we round all fractional variables up to 1 with probability proportional to their value in the LP solution. That is, if $x_e = \frac{3}{4}$, then we will round it to 1 with probability $\frac{3}{4}$, and to 0 with probability $1 - \frac{3}{4} = \frac{1}{4}$. This technique is often useful because when the objective consists of a sum of many LP variables, we can apply Chernoff-type bounds to show that the objective value cannot change much in expectation from doing this rounding. However, this rounding technique again runs afoul of the flow constraints (2), (3), and (4); almost certainly rounding the variables independently will cause some of these constraints to be violated, and there is no easy fix for this even if we are willing to give away something in the objective. These two examples of rounding techniques that do not work both seem to indicate that we need to consider ways of rounding which will adjust the values of multiple fractional variables in tandem in order to avoid violating constraints.

One type of dependent rounding that we investigated briefly was introduced by Gandhi et al. [9]. They show that if a problem can be formulated such that the LP variables correspond to edges of a bipartite graph, then it is possible to round the fractional values to integral ones in such a way that the expected values on the edges are preserved, the degrees of the vertices are preserved if they were integral to begin with, and a certain "negative correlation" property holds which allows for the application of Chernoff-type bounds to sums of LP variables. The difficulty with using this approach on our problem is finding a way to represent our problem such that the x_e variables are edges on a bipartite graph. Clearly our graph G will not be bipartite in general, so we cannot just use this graph. We also need to ensure that there is only 1 edge in the bipartite graph for each LP variable, which does not hold if we take our bipartite graph to consist of something like a copy of the vertices of G in each bipartition. Because of this difficulty, we moved on from this method to other dependent rounding techniques.

The last technique which we have explored is to build up a set **Sol** of possible paths by repeating the following procedure until no fractional values remain:

1. Find the edge $e = (u, v)$ with minimum fractional value $f_e > 0$.
2. Find a path P of edges corresponding to nonzero x_e variables from s to u and from v to t (such a path must exist by the flow constraints).
3. Add (P, f_e) to **Sol**. Reduce the value of every edge e' in P by f_e.

We can carry out the above procedure at most $|E|$ times before running out of fractional variables, as every iteration sets some fractional edge to 0. Once we have built up the set **Sol** of potential paths, we then need to select a single path to return as our solution. There are two possible approaches here: either pick a path P with probability f_e (these f_e will sum to 1 because of the flow constraint), or simply check every path in **Sol** and return the best. The latter method strictly outperforms the former, but the former may be easier to analyze. The next step would be to bound the quality of this approximation in terms of the original LP objective value, however we have found evidence that obtaining a good bound using the above LP is probably not possible, which will be detailed in Section 6.3. Regardless, we still think this general rounding procedure may prove useful if combined with a different LP relaxation of this problem; this will be explained in Section 6.4.

6.2 Primal-Dual Approach

Another method which we pursued briefly was to employ some kind of Primal-Dual approach, using the dual LP to guide the rounding of the fractional values. For reference, the dual LP is written below, where the dual variables α_v correspond to the constraints in (2), (3), and (4), variables β_j correspond to the constraints in (5), and dual variables δ_j correspond to the linear versions of the constraints in (7).

$$\text{min.} \quad \alpha_s + \alpha_t + \sum_{\substack{j \\ u_j \in \mathcal{U}}} \delta_j \tag{9}$$

$$\text{s.t.} \quad \beta_j + \delta_j \geq 1 \qquad \forall u_j \in \mathcal{U} \tag{10}$$

$$\alpha_t \geq \sum_{\substack{j \\ u_j \in S_e}} \beta_j \qquad \forall e = (v, t) \tag{11}$$

$$\alpha_{v_j} \geq \alpha_{v_i} + \sum_{\substack{j \\ u_j \in S_e}} \beta_j \qquad \forall e = (v_i, v_j) \quad \text{s.t.} \quad v_i, v_j \in V \setminus \{s, t\} \tag{12}$$

$$\alpha_s \geq 0 \tag{13}$$

$$\beta_j, \delta_j \geq 0 \qquad \forall u_j \in \mathcal{U} \tag{14}$$

The main problem here is that in order to use the Primal-Dual method, we need a simple feasible starting solution from which we can begin increasing variables until constraints become tight. Typically if the dual is a packing LP, we use the all zeros solution, whereas if the dual is a covering LP, we use the all ones solution. However, neither of these are feasible solutions for our LP; in order for a solution to be feasible, it needs to specify a (fractional) path in G. But once we have initialized some particular path, there are no "loose constraints" which we can increase the variables for. In order to go from one path to another, we first need to decrease certain variables and move through an intermediate infeasible solution, before we can get to a different feasible solution. This seemed to us to be a fundamental issue that dooms most kinds of Primal-Dual technique to failure, so we have not pursued these methods further.

6.3 Integrality Gap

While working to find a rounding method for the primal LP, we came across an example demonstrating that the integrality gap can be quite large, which likely means no rounding technique can achieve good performance. The example is shown below.

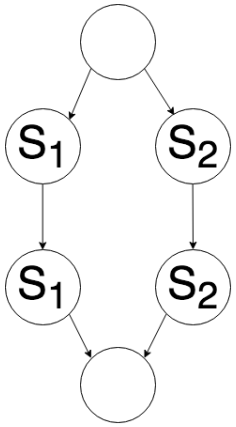


Figure 4: Graph demonstrating the integrality gap of the above LP. Let $S_1 = \{1\}, S_2 = \{2\}$. Then the LP can achieve coverage 2 by setting every edge variable to $\frac{1}{2}$.

While this example only shows the integrality gap is at least 2, it is simple to extend it and increase the gap: instead of the source branching into 2 disjoint paths of length 2, let the source branch into k disjoint paths of length k, where every vertex/edge in each path is assigned the same set $S_i = \{i\}$. Then the LP can assign value $\frac{1}{k}$ to each edge and obtain coverage k, while the best integral solution has coverage 1. Since this requires k^2 vertices in total (not counting the source/sink), this shows the integrality gap is at least $\sqrt{n}$.

It is worth considering that while the instance described above has a large integrality gap, it is actually quite easy to solve optimally; since the k paths are disjoint, there are only polynomially many paths in the entire graph, so we could simply brute force every possible path. Also, if we were to start adding edges between the paths, the integrality gap quickly collapses back down. So it may be possible to show that these instances with very large integrality gap must actually contain only a relatively small number of paths, in which case we could handle them separately.

Another direction we can go in is to try and strengthen the LP. The issue in the above instance is that the flow constraints are in some sense very weak, as they only enforce that the amount of flow going in is equal to the amount coming out, rather than that the amount coming in must be either 0 or 1. This allows the LP to get away with a lot by having only very tiny fractional amounts of flow entering many vertices. In the next subsection, we discuss another LP formulation which we hope might help us get around this issue.

6.4 Strengthening the LP

Instead of writing the LP with a variable for every edge, as above, we could instead write it with a variable for every path in G. The obvious issue with this is that there may be exponentially many paths, in which case this LP will have exponentially many variables, and we will not be able to solve it practically. We first define this new LP, then discuss a potential way of working around this difficulty. We will have a variable y_j for every element $u_j \in \mathcal{U}$, as before, but now will have a variable x_P for every path P in G. The LP is as follows, where we write $\mathcal{P}$ for the set of all paths in G:

$$\text{max.} \quad \sum_{\substack{j \\ u_j \in \mathcal{U}}} y_j \tag{15}$$

$$\text{s.t.} \quad y_j \leq \sum_{\substack{P \in \mathcal{P} \\ u_j \in P}} x_P \qquad \forall u_j \in \mathcal{U} \tag{16}$$

$$\sum_{P \in \mathcal{P}} x_P \leq 1 \tag{17}$$

$$0 \leq x_P \qquad \forall P \in \mathcal{P} \tag{18}$$

$$0 \leq y_j \qquad \forall u_j \in \mathcal{U}. \tag{19}$$

This LP is very similar to the first, but we have replaced the flow constraints by the constraint in (17), which enforces that we choose only one path. To try and work around the fact that this LP has exponentially many variables, we can instead work with the dual LP, which has polynomially many variables, but exponentially many constraints. The dual LP is written below, where the dual variables α_j correspond to the constraints in (16) and dual variable β corresponds to the constraint in (17):

$$\text{min.} \quad \beta \tag{20}$$

$$\text{s.t.} \quad \alpha_j \geq 1 \qquad \forall u_j \in \mathcal{U} \tag{21}$$

$$\beta \geq \sum_{\substack{j \\ u_j \in P}} \alpha_j \qquad \forall P \in \mathcal{P} \tag{22}$$

$$0 \leq \beta. \tag{23}$$

If we could solve the dual in polynomial time, we could use the solution to construct an optimal primal solution, which we could then try to round using the final technique described in Section 6.1. There are two barriers which need to be overcome here: first, we need a polynomial time separation oracle which can return a violated dual constraint for an infeasible solution in polynomial time. This would allow us to get an optimal solution to the primal LP in polynomial time. Then, assuming we have such an oracle, we need to bound the value of the objective after applying the rounding. We are still in the process of working on both pieces at this time.

7 Other Approaches

Aside from the $(1/f)$-approximation greedy approach and linear programming based approaches, we also explored two other approaches: dynamic programming and simple greedy heuristics. In this section, we describe our findings regarding these two approaches.

7.1 Dynamic Programming

For applying a dynamic programming approach, we need to keep a table of two states, one denoting the vertex, and the other denoting the *coverage state*. Coverage state represents the set of covered elements so far. At each vertex, we need to know what are the possible coverage states when that vertex is reached along some path. In the worst case, the number of distinct coverage states can be exponential in the number of different elements of $\mathcal{U}$. Hence, such a dynamic programming approach is prohibitive in general. However, if we have a guarantee that the number of different possible coverage states is polynomial in the graph size, then such a dynamic programming approach can give us exact solution in polynomial time and space.

7.2 Heuristic

We thought of few heuristic approaches inspired by the greedy algorithm for the maximum coverage problem. At each iteration, the greedy algorithm for standard maximum coverage chooses the set that covers the greatest number of uncovered elements. However, we have the additional path constraint, and hence cannot choose just any set. If we try to still apply the greedy approach and pick the vertex that covers the highest number of uncovered elements, we are not guaranteed feasibility, since in many cases the picked vertices might not form a valid path.

Another heuristic could be to pick a vertex that covers both a high number of uncovered elements and also has high out-degree. The intuition behind this heuristic is that a vertex with high out-degree has the potential of allowing more choices of vertex in the next iteration, and this increases the likelihood of covering more elements. However, this approach can also be made arbitrarily bad by designing an adversarial DAG where the optimal path is a long chain of vertices each covering one unique element (together the path covers the whole universe), but each vertex has out-degree one. Although this is an optimal path, the heuristic will not pick this path, since the first vertex of the long chain covers only one element and has out-degree one.

In summary, a good heuristic depends on the instance of the problem. We plan to study this further in order to develop ideas regarding what guarantees we can give under certain assumptions about the problem instance.

8 Conclusions and Future Work

Moving forward, we think that the most promising direction will be to continue investigating methods of rounding linear programming relaxations of the problem. As discussed in Section 6, there are several methods that still have not been fully explored. One method would be to continue investigating the integrality gap of the first LP, which we can solve efficiently, and hope to demonstrate a relationship between the integrality gap and the number of paths in the graph. Even if we cannot establish such a relationship, it may at least be possible to find a way of rounding the LP which attains an approximation factor of $\sqrt{n}$. While not as good as a constant factor approximation, this may still be an improvement over the $(1/f)$-approximation for graphs containing elements with very high frequencies. If instead we can find a polynomial time separation oracle for the second path-based LP, then this may lead to a rounding method achieving a much better approximation factor than $\sqrt{n}$, as this LP is much stronger.

Another direction that merits study which we have not focused much on is developing lower bounds for the problem. Given the reduction from maximum coverage to our problem and the lower bound for maximum coverage, we know that obtaining any approximation factor better than $(1 - \frac{1}{e})$ is impossible unless P = NP. But it seems possible that this bound is not tight for our problem; if continued efforts to establish a constant factor approximation for our problem do not yield results, it would be worth investigating whether a stronger lower bound could be proven.

References

[1] G. Best and R. Fitch. Probabilistic maximum set cover with path constraints for informative path planning. In *Australasian Conference on Robotics and Automation*. ARAA, 2016.

[2] A. Björklund, T. Husfeldt, and S. Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages, and Programming*, pages 222–233. Springer, 2004.

[3] F. Cellinese, G. D'Angelo, G. Monaco, and Y. Velaj. Generalized budgeted submodular set function maximization. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, pages 31:1–31:14, 2018.

[4] C. Chekuri and A. Kumar. Maximum coverage problem with group budget constraints and applications. In *Approximation, Randomization, and Combinatorial Optimization, Algorithms and Techniques, 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2004, and 8th International Workshop on Randomization and Computation, RANDOM 2004, Cambridge, MA, USA, August 22-24, 2004, Proceedings*, pages 72–83, 2004.

[5] R. Cohen and L. Katzir. The generalized maximum coverage problem. *Inf. Process. Lett.*, 108(1):15–22, 2008.

[6] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[7] Y. Filmus and J. Ward. The power of local search: Maximum coverage over a matroid. In *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, pages 601–612, 2012.

[8] B. Fu. Partial sublinear time approximation and inapproximation for maximum coverage. In *Computing and Combinatorics - 24th International Conference, COCOON 2018, Qing Dao, China, July 2-4, 2018, Proceedings*, pages 492–503, 2018.

[9] R. Gandhi, S. Khuller, S. Parthasarathy, and A. Srinivasan. Dependent rounding and its applications to approximation algorithms. *J. ACM*, 53(3):324–360, 2006.

[10] Y. N. Hoogendoorn. The Maximum Coverage Problem. `https://thesis.eur.nl/pub/38550/Hoogendoorn.pdf`, 2017.

[11] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.

[12] S. Khuller, A. Moss, and J. S. Naor. The budgeted maximum coverage problem. *Information processing letters*, 70(1):39–45, 1999.

[13] N. Kicillof, W. Grieskamp, N. Tillmann, and V. A. Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, July 9-12*, pages 1–11, 2007.

[14] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions. *Mathematical programming*, 14(1):265–294, 1978.

[15] B. Saha and L. Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA*, pages 697–708, 2009.

[16] V. V. Vazirani. *Approximation algorithms*. Springer, 2001.