Online Steiner Cover Problems in Hypergraphs

Joey Byrne

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

April 2025

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Joey Byrne**

Entitled: **Online Steiner Cover Problems in Hypergraphs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Harutyunyan

_____ Examiner
Dr. Harutyunyan

_____ Examiner
Dr. Narayanan

_____ Supervisor
Dr. Pankratov

Approved by _____
Dr. Charalambos Poullis, Chair
Department of Computer Science and Software Engineering

_____ 2025

_____
Dr. Mourab Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Online Steiner Cover Problems in Hypergraphs

Joey Byrne

The online Steiner cover problem in hypergraphs (OSCH) is a generalization of the online Steiner tree problem in graphs. An edge-weighted hypergraph $H = (V, E, w)$ is given offline and a set of terminal vertices $R \subseteq V$ is requested sequentially online. Upon receiving each request $r_i \in R$ an algorithm for the problem must buy some edges $P_i \subseteq E$ which connect r_i to the previous solution. The solution after satisfying the i^{th} request is then the union over all edges bought up to that point, $F_i = \bigcup_{j=1}^{i} P_j$. The goal is to minimize the total cost of the solution to connect the requests, i.e., for $|R| = n$ let $F = F_n$, then we want to minimize $\sum_{e \in F} w(e)$. The generalized OSCH (GOSCH) is a generalization of both the OSCH and the Steiner forest problem in graphs. Again, we are given an edge-weighted hypergraph $H = (V, E, w)$ offline, but instead of a set of terminals as the online portion of the input we are given a set of terminal pairs $R \subseteq V \times V$. Upon receiving the i^{th} request pair $p_i \in R$ an algorithm for this second problem must buy some set of edges $P_i \subset E$ which connect the terminals from p_i. We define the instantaneous solution after connecting request p_i as before, so $F_i = \bigcup_{j=1}^{i} P_i$ and the final solution is again denoted $F = F_n$ for a request sequence of size $|R| = n$. The worst-case performance of an online algorithm is measured by the competitive ratio, which is the ratio between the cost of a solution obtained by the online algorithm to that of an optimal offline solution. Besides some simpler preliminary results, we obtain a lower bound on the competitive ratio for OSCH (which also applies to GOSCH) of $\Omega(k \log n)$, where k is the rank of the hypergraph, and a matching upper bound for the simple $Greedy$ algorithm. For GOSCH we show that the simple $Greedy$ algorithm is $O(k \log^2 n)$-competitive and provide another algorithm, called $GGSC$, which achieves a competitive ratio of $O(k \log n)$.

Dedication

To my son, August

Acknowledgments

I would like to first acknowledge my wife Monica for her endless support throughout my time at university. Were it not for you it is doubtful that I would have made it as far as I did.

My supervisor, Dr. Pankratov, whose guidance and insight was most helpful in my research and in the classroom. It was thanks to him and to the members of the examining committee, Dr. Harutyunyan and Dr. Narayanan, that I knew I wanted to pursue further studies in Computer Science, so thank you all for helping me find and foster a new passion.

My colleagues and friends from the theory reading group: Ali, Akash, Aram, Arnav, Kamran, Mahtab, Mohammadhossein, Narek, Tristan, Tung, Yaqiao. Thank you for the engaging and informative discussions on the variety of subjects we managed to touch on.

To my close friends and family for always being there when I need them, Mom and Dad, Leigh and Estelle, Connor, Dante, Jay, Kevin, Tyler and Claudia, Tylor and Tyra. You are all among the most important parts of my life and I hope you know just how much you mean to me.

Contents

List of Figures

Chapter 1

Introduction

Graph theory provides us with a rich and powerful mathematical model to analyze a large variety of real world problems. Nearly all problems which deal with a finite set of objects and pairwise relations between them can be conceptualized into a graph problem in a quick and simple way. Formally, a graph G is a pair (V, E), where V is a set of vertices representing objects, and E is a set of edges representing pairwise relations between the objects. Each edge $e \in E$ is a subset of V of size 2, i.e., $e = \{u, v\}$ represents that there is a relation between u and v. From social networks and the internet to building roads and planning trips, after studying graphs one might find that they appear almost everywhere.

What if we want to study the connection of a group of points in some network, only to find that the relations are more complex than what can be seen between any two objects? For example, in a social network there are many ways that a user profile can be influenced by another: if they are friends then they might share a pairwise relation, but if they share a common interest but have never met they may still notice each others posts and find interest in what the other has to say. This common interest is possible to model by including topics pages in a social network, where people may follow each other but also follow a topic or group. These two relations are not the same, and nor should the model used to study them be the same. A group of users in a social network who follow a given topic is often called an online community, and this represents a subset of users who

may or may not know each other yet who interact in ways which aren't explained simply enough by pairwise relations. Trying to model these scenarios, one is naturally led to consider the concept of hypergraphs. A hypergraph H is also a pair (V, E), where V is a set of vertices representing objects of interest, similarly to ordinary graphs. Unlike ordinary graphs, in a hypergraph H the set E is a set of *hyperedges*, where each hyperedge $e \in E$ is a subset of V (not necessarily of size 2), i.e., $e \subseteq V$. Thus, for the scenario just discussed, each edge e would correspond to the community of users of a given topic. The rank of a hypergraph is the size of the largest edge.

This thesis is concerned with Steiner-style problems in hypergraphs in the online setting. Before we discuss our contributions, we need to explain these terms. In ordinary graphs, the Steiner tree and Steiner forest problems are classical combinatorial optimization problems with wide-ranging applications in social networks [50] [30], systems modeling [50] [29], biological networks [30], urban transit networks [29], just to name a few. Given a graph with edge weights and a subset of vertices called terminals, the Steiner tree problem seeks the minimum-weight connected subgraph that spans all terminals, potentially including non-terminal vertices (Steiner vertices) to reduce cost. The Steiner forest problem generalizes this by allowing multiple terminal pairs or groups, aiming to connect each specified pair or group within the smallest total edge weight. Both problems are $\mathcal{NP}$-hard and have inspired a rich body of research in approximation algorithms and heuristics.

Online algorithms are a class of algorithms designed to make decisions sequentially, without knowledge of future inputs. Unlike offline algorithms, which have access to the entire input up-front, online algorithms must process information as it arrives, often under strict time or resource constraints. This setting naturally arises in many real-world scenarios such as caching, scheduling, and network routing. The performance of an online algorithm is typically analyzed using competitive analysis, which compares its output to an optimal offline solution. Designing effective online algorithms requires balancing adaptability with foresight, making it a rich area of theoretical and practical research.

To the best of our knowledge, the analogues of the Steiner tree and Steiner forest problems on hypergraphs have not been considered in the online setting, despite the fact that for ordinary

graphs both online Steiner tree and online Steiner forest problems have received a fair amount of attention [39, 2, 31, 6, 13]. This thesis aims to narrow this gap. More specifically, we introduce and study a variant of the online Steiner tree problem but for the setting of hypergraphs, and we call it online Steiner cover problem in hypergraphs (OSCH). We note that the term Steiner *tree* is no longer appropriate in the hypergraph setting, as it is possible that no hypertree solution exists for a given instance. Similarly, we introduce and study the extension of online the Steiner forest problem, also called the generalized Steiner problem in graphs, to the setting of hypergraphs. We refer to this problem as the generalized online Steiner cover problem (GOSCH). A simple reduction is described in Chapter 5 showing that the GOSCH is a generalization of the OSCH. The main difference between the two problems is that in OSCH the requests come one by one, where the first request is considered the root node, whereas in GOSCH the requests come in pairs. The reduction simply takes each request in the OSCH and pairs it with the root node, giving an input compatible with the GOSCH setting.

Besides being natural online problems, the study of OSCH and GOSCH can be motivated by the following application in recommendation systems for social networks. Let's say you are trying to decide what content to recommend to the users of a social network. Obviously recommending community content to members of each community would be effective, but if that is the entirety of a users recommendations they might quickly become bored of seeing the same type of content over and over again. If we want to recommend content outside of a user's regular consumption, we can include them as a terminal request in a Steiner cover of the network alongside the profiles whose content they have consumed in the past outside of their regular community sources. In addition, the Steiner points (nodes covered by the solution which were not terminals) can be seen as more users to recommend content to. The edges in the solution are the online communities from which the recommendations can be sourced for these users. The connectivity requirement ensures that these recommendations are close to the users interests in some way, so that the recommendations are sufficiently appropriate for them.

In addition, the concatenation phase of obstacle-avoiding Euclidean Steiner tree problem (OAST) in ordinary graphs can be formulated as a Steiner tree problem in hypergraphs [68, 62], indicating

that the algorithms we develop in the hypergraph setting can be used as sub-routines in variants of Steiner problems for ordinary graphs. The OAST has applications in VLSI design, where some objectives are the minimization of wire lengths within the chip's circuits and minimizing the distances between chip components which communicate frequently.

Our contributions. We show a lower bound of $\Omega(k \log n)$ on the competitive ratio of deterministic algorithms for OSCH on hypergraphs of rank k. We also prove that the natural greedy algorithm achieves a matching upper bound. For GOSCH on hypergraphs of rank k, we show that the natural greedy achieves the bound of $O(k^2 \log n)$ and we demonstrate how to modify the algorithm to achieve the tight bound of $O(k \log n)$. Note that results for ordinary graphs follow from our results by setting $k = 2$. Our lower bound result introduces a new construction and, thus, it is an original contribution. Our proofs of upper bounds heavily rely on the framework and tools from the online Steiner tree and forest literature and are less original. Nonetheless, we can say that our contribution with regards to the upper bounds is two-fold: first, we fill in some details missing in original proofs, and second, we precisely identify and perform the required modifications for values of k larger than 2. The identification of necessary modifications from $k = 2$ to larger values of k is non-trivial due to the complexity of the primal-dual framework used to establish the upper bound for GOSCH. Whenever a theorem or a lemma appeared before in the literature we give a citation next to it, e.g., **Theorem ([citation])**. Theorems and lemmas without citations are original contributions.

Organization. The rest of this thesis is organized as follows. Chapter 2 goes over the requisite definitions more formally and introduces some useful algorithms and the analytic tools implemented here. Chapter 3 goes over the previous work on Steiner problems in graphs in the offline and online setting, as well as some related problems on hypergraphs. Our contributions for the OSCH problem appear in Chapter 4. Chapter 5 contains our contributions for the GOSCH problem. Finally, Chapter 6 ends with a recap of the results and some discussion on future work.

Chapter 2

Preliminaries

Here we provide the necessary definitions, algorithms, and results that will be needed in the rest of the thesis. We begin with formally defining online algorithms in Section 2.1, graphs in Section 2.2, and hypergraphs in Section 2.3. Then we discuss the Depth-First Search algorithm in Section 2.4, the Breadth-First Search and Dijkstra's algorithms in Section 2.5. In Section 2.6, we discuss Steiner tree and forest problems in graphs, and give an example to illustrate why it is no longer appropriate to call the problem Steiner *tree* or Steiner *forest*, respectively, when extending them to hypergraphs. We present greedy algorithms for the Steiner tree and Steiner forest problem in graphs in Section 2.7. Following this we provide the Linear Programming framework and Primal-Dual analysis method in Section 2.8.

2.1 Online Algorithms

The study of online algorithms concerns itself with problems whose input is not given in its entirety from the beginning, but which arrives sequentially; piece by piece. Any algorithm designed to solve such problems must make an irrevocable decision upon receiving each portion of the input before the next one is revealed to it. The worst-case analysis of these problems is likened to a game between an algorithm, designed to solve any instance of a problem, and an adversary, who may

design an input sequence with the knowledge of which decisions the algorithm will make.

For an optimization problem, the performance of an algorithm is measured by taking the ratio of the value it achieves on the adversarial input with the optimum value achievable in an offline setting (when the entire input is available in advance). More specifically, let $R = (r_1, r_2, \ldots, r_n)$ be a sequence of requests for an online *minimization* problem. Let ALG be an online algorithm[1]; the value of the objective achieved by ALG on R is denoted by $ALG(R)$. We use OPT to denote an optimal offline algorithm with $OPT(R)$ having similar meaning to $ALG(R)$. We say that ALG achieves *competitive ratio* ρ if there exists a constant c such that for all inputs R we have:

$$ALG(R) \leq \rho \cdot OPT(R) + c.$$

If $c = 0$ then we say that ALG achieves *strict* competitive ratio ρ. If ALG achieves competitive ratio ρ (respectively, strict competitive ratio ρ), we refer to ALG as being ρ-competitive (respectively, strictly ρ-competitive). Then the *competitive ratio* of ALG, denoted by $\rho(ALG)$, is defined to be the infimum over ρ such that ALG is ρ-competitive. An algorithm who achieves a ratio of ρ in this way is said to be *ρ-competitive*.

This online framework can in some cases prove more practical than traditional offline algorithms, where the input is given from the start and an algorithm can perform any sort of preprocessing before attempting to compute a solution. For example, what if the amount of input is exceedingly large and the available storage is quite small? In this case an algorithm could not preprocess all the input at once, so it would have to decide what to do with the input it has before moving on to the next batch. Another example would be if the problem is intractable, then even small input instances would require an unrealistic amount of time to provide a solution. In this case, it makes sense to avoid an exact solution in favor of an approximation. To study the online version of the problem and determine the competitive ratio would then be equivalent to determining the approximation ratio of an algorithm which doesn't sort or search through the input beforehand. Although online algorithms

[1] In this thesis we deal exclusively with deterministic algorithms.

almost always fail to provide optimal solutions, they surprisingly often provide some of the best approximation guarantees. For more background on online algorithms, we refer an interested reader to the monograph of Borodin and El-Yaniv [16].

2.2 Graphs

A *graph*, $G = (V, E)$, is a set of *points*, or *vertices*, *nodes*, denoted V along with a set of *lines*, or *edges*, *arcs* denoted E, containing pairs of vertices. If two vertices $u, v \in V$ are joined by an edge $e = \{u, v\}$, then we say u and v are *adjacent* vertices, or that u is adjacent to v, or vice-versa. We also say that the edge e is *incident* on u and v. If in a given graph there is a sequence of vertices $\{u_i\}_{i=0}^{n}$ and edges $\{e_j\}_{j=0}^{n-1}$ alternating in this way: $u_0, e_0, u_1, e_1, ..., e_{n-1}, u_n$, for some $n \geq 1$ with $e_i = \{u_i, u_{i+1}\}$, then we say the vertices u_0 and u_n are *connected*. The set of edges in the sequence is a called a *walk* from u_0 to u_n in G. The *length* of the walk is the number of edges in the sequence. If every edge in the walk is unique, then it is called a *path*, if every vertex is unique except for $u_0 = u_n$ then it is called a *cycle*. Some graphs can include edge-weights, $G = (V, E, w)$ where $w : E \to \mathbb{R}$, node-weights where $w : V \to \mathbb{R}$. Then we can describe the weight of a path P as $w(P) = \sum_{e \in P} w(e)$ for edge-weighted graphs, or $w(P) = \sum_{v \in P} w(v)$ for node-weighted graphs. For more information about graph theory, we recommend consulting the text by Harary [37].

2.3 Hypergraphs

A *hypergraph* is a pair $H = (V, E)$, where V is the set of vertices and $E \subseteq 2^V$ is a set of hyperedges. Sometimes we refer to vertices and edges of hypergraph H by $V(H)$ and $E(H)$, respectively. It is a generalization of a graph in which the edges become subsets of the vertices of arbitrary size, instead of containing strictly 2 nodes. In a simple hypergraph, no hyperedge may be a proper subset of another. When it is clear that we are talking about hypergraphs and not ordinary graphs, we will refer to the hyperedges as simply edges of H.

The *rank* of a hypergraph H, denoted by $\mathrm{rank}(H)$, is the size of its largest edge, i.e., $\mathrm{rank}(H) = \max_{e \in E(H)} |e|$. Similarly, if the rank of a hypergraph is k we may say that it is *k-restricted*. Additionally, if every edge has size k, we say that the hypergraph is *k-uniform*. Note that every simple ordinary graph is then a 2-uniform hypergraph.

A *walk* in a hypergraph $H = (V, E)$ is a sequence of alternating vertices and edges from $V(H)$ and $E(H)$, $(v_1, e_1, v_2, e_2, \ldots, e_{m-1}, v_m)$ where each $v_i, v_{i+1} \in e_i$ for every $i \in [m]$. The vertices and edges may repeat any number of times, and we say that the *length* of the walk is the number of edges in the sequence. A *path* in a hypergraph is defined as a walk which does not repeat any vertex, and a *cycle* is a walk which does not repeat any vertex except that $v_1 = v_m$. A hypergraph is called *connected* if for every pair of vertices $s, t \in V(H)$ there is a path P starting at s and ending at t. The *distance* between nodes $u, v \in V(H)$ is the length of a shortest path connecting u and v.

An *edge-weighted* hypergraph is a triple $H = (V, E, w)$ where V and E are defined as before, and $w : E \to \mathbb{R}$ is a function from the edge set of H to the real numbers. We will consider only non-negative edge-weighted hypergraphs, so $w : E \to \mathbb{R}_{\geq 0}$. We call $w(e)$ for some $e \in E$ the *weight* of edge e in H, but sometimes it is referred to as the *cost* of the edge, especially in the context of an algorithm selecting edges to include in a solution. Also, the weight of a hypergraph can be defined as the sum of its edge weights, $w(H) = \sum_{e \in E(H)} w(e)$, and the same can be done for any set of weighted edges, such as a subset of $E(H)$ for some weighted hypergraph H. All hypergraphs can be interpreted as edge-weighted hypergraphs, by simply taking the weight function to be $w : E \to \{1\}$ when no weight function is specified. The notions of walks, paths, and cycles are the same for weighted hypergraphs, but in addition to the length being the number of edges we can define the *weight* of these objects as the sum of their edge-weights, i.e., for a path $P = (v_1, e_1, ..., e_m, v_{m+1})$ the weight of the path would be $w(P) = \sum_{i=1}^{m} w(e_i)$. The *distance* between nodes in a weighted hypergraph is then the minimum weight among the paths between them,

$$\mathrm{dist}_H(u, v) = \min_{\{P|\ P \text{ is a path between } u, v \in V(H)\ \}} w(P)$$

Adjacent vertices in a graph, those connected by a single edge, are sometimes referred to as

neighbors. Expanding on this, we can define the *neighborhood* of a vertex, sometimes denoted $N(v)$ for a vertex $v \in V$, as the subset of vertices in the graph which are adjacent to it,

$$N(v) = \{u \in V(H) : \exists e \in E \text{ such that } u, v \in e\}.$$

For a comprehensive introduction to hypergraphs, we refer an interested reader to the book by Berge [12].

2.4 Depth-First Search

The depth-first search algorithm (see, for example, this classical textbook [21]) in graphs is shown below, it traverses a graph by following a path of unvisited vertices as far as possible, then once it has no more options it reverts back to the last seen node which had unvisited neighbors and continues. To achieve this, we can mark nodes with labels to determine whether they have been discovered but not explored, explored fully, or neither. It proves useful in the analysis of the greedy algorithm for Steiner trees. The DFS algorithm is often used to explore the connected components of a graph.

Algorithm 1 Depth-First Search (DFS)

```
1: procedure DFS(G, v)
2:     Mark v as discovered
3:     for every edge {v, u} do
4:         if u is unmarked then
5:             DFS(G, u)
6:     Mark v as explored
```

Here is the same algorithm (the same method of traversal) in hypergraphs. The main difference is that each edge now contains multiple vertices to choose where to proceed from.

Algorithm 2 Hypergraph Depth-First Search (HDFS)

```
1: procedure DFS(H, v)
2:     Mark v as discovered
3:     for every edge e such that v ∈ e do
4:         for every u ∈ e such that u ≠ v do
5:             if u is unmarked then
6:                 DFS(H, u)
7:     Mark v as explored
```

A property that will prove helpful in the coming analysis is the number of times DFS and $HDFS$ traverse a given edge. DFS will traverse each edge in a connected component exactly twice; the first time exploring deeper into the component, and the second time while retracing its steps to explore the next path. This is generalized in the $HDFS$, as each edge e is traversed at most $2(|e| - 1)$ times; going between each of the vertices contained in the edge in search of all paths outwards from that edge before retracing its steps.

2.5 Breadth-First Search and Dijkstra's Algorithm

The breadth-first search algorithm visits all neighbors of a given vertex before moving on to the next node. This splits up the traversal in "waves" where in each wave we explore nodes further from the source than the last (in terms of number of edges between the current node and the source, not necessarily the weight of the path). This is useful for computing the single-source shortest paths of an unweighted graph. Let Q be a *queue*, which is a data structure similar to a set but which maintains FIFO (First In - First Out) order. Some important function calls of the queue data structure are the pop function, which deletes and returns the first element of the queue, and the $enqueue$ function, which inserts an element to the end of the queue. [21]

Algorithm 3 Breadth-First Search Algorithm

```
1:  procedure BFS(G = (V, E), s ∈ V)
2:      Q ← {s}
3:      for u ∈ V do
4:          d(u) ← ∞
5:      d(s) ← 0
6:      while Q ≠ ∅ do
7:          u ← Q.pop()
8:          for e = {u, v} ∈ E do
9:              if d(v) = ∞ then
10:                 Q.enqueue(v)
11:                 d(v) ← d(u) + 1
12:     return d
```

The BFS algorithm can be implemented with some modifications in order to compute the single-source shortest paths in a graph with non-negative edge weights. This implementation is referred to as Dijkstra's algorithm, after its inventor Edsger Dijkstra. The implementation of BFS by Dijsktra's algorithm must keep a priority queue of all nodes in the graph. This implementation will make use of a *priority queue*, which is a data structure built up from the queue seen in the BFS algorithm, but where the elements are added with a given priority value which determines their relative ordering. The priority of each node in the queue is initialized as ∞ (besides the source node whose priority is initialized to 0) which will ultimately correspond to its least distance from the source. This is updated when the node is encountered in the traversal the first time, then each subsequent time it is encountered if the update would result in a lesser distance. It can also prove useful to keep track of the predecessor within this shortest path separately, in order to reconstruct the actual sequence of edges of the shortest path. Once a vertex becomes is next in the priority, it can be removed from the queue before its neighbors are explored. The algorithm terminates once the queue is empty, and the values returned are the list of shortest path distances and the list of predecessors along the corresponding shortest paths.

Algorithm 4 Dijkstra's Shortest Path Algorithm

1: **procedure** DIJKSTRA($G = (V, E, w), s \in V$)
2: $Q \leftarrow \emptyset$
3: **for** $u \in V$ **do**
4: $d(u) \leftarrow \infty$
5: $Q \leftarrow Q \cup \{(u, d(u))\}$
6: $\pi(u) \leftarrow \varnothing$
7: $d(s) \leftarrow 0$
8: update $d(s)$ for $s \in Q$
9: **while** $Q \neq \emptyset$ **do**
10: $u, d(u) \leftarrow \min_{d(v)}\{(v, d(v)) \in Q\}$
11: $Q \leftarrow Q \setminus \{(u, d(u))\}$
12: **for** $e = \{u, v\} \in E$ **do**
13: **if** $d(v) > d(u) + w(e)$ **then**
14: $d(v) \leftarrow d(u) + w(e)$
15: $\pi(v) \leftarrow u$
16: update $d(v)$ for $v \in Q$
17: **return** (d, π)

The BFS and $Dijkstra$ [21] algorithm are often used to test the connectivity of a graph, since if the graph is connected each vertex would have been marked as explored or have a given shortest path in the distance array at the end of the procedure. In undirected graphs this can be checked no matter which vertex is chosen to be the source node. In order to get a hypergraph variant of $Dijkstra$, we simply replace the **for** loop starting on line 12 with a nested loop in which the outer loop iterates over every edge $e \in E$ that contains u (as set in line 10), and the inner loop iterates over every neighbor of u in edge e.

Algorithm 5 Dijkstra's Shortest Hyperpath Algorithm

procedure HDIJKSTRA($H = (V, E, w), s \in V$)
  $Q \leftarrow \emptyset$
  for $u \in V$ **do**
    $d(u) \leftarrow \infty$
    $Q \leftarrow Q \cup \{(d(u) : u)\}$
    $\pi(u) \leftarrow \varnothing$
  $d(s) \leftarrow 0$
  update $d(s)$ for $s \in Q$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow \min_{d(v)}\{v \in Q\}$
    $Q \leftarrow Q \setminus \{d(u) : u\}$
    for $e \in E : u \in e$ **do**
      for $v \in e : v \neq u$ **do**
        if $d(v) > d(u) + w(e)$ **then**
          $d(v) \leftarrow d(u) + w(e)$
          $\pi(v) \leftarrow u$
          update $d(v)$ for $v \in Q$
  return (d, π)

2.6 Steiner Problems from Graphs to Hypergraphs

In the online Steiner Tree problem on graphs, the input is split between an online and offline portion. Offline, we are given the graph itself, $G = (V, E, w)$, and online we are then presented with the terminal request sequence, $R = (r_1, r_2, \ldots, r_n)$, where $r_i \in V$ for each $i \in [n]$. Unless otherwise stated, we will denote $|V|$ by N. Upon receiving each request an online algorithm for this problem must make a decision, it must choose which edges to include in the solution in order to connect the current request to the previous ones.

Problem Definition 6 Online Steiner Tree

Input

$G = (V, E, w)$ an edge-weighted graph $w : E \to \mathbb{R}^+$, offline.
$R = (r_1, ..., r_n)$ – request sequence where $r_i \in V \; \forall i \in [n]$; given online.

Output

$F_i \subseteq E$ on request r_i, s.th. $\bigcup_i F_i$ is a connected component with
$\forall r \in R \; \exists e \in \bigcup_{i=1}^{n} F_n$ such that $r \in e$ and $\bigcap_{i=1}^{n} F_i = \emptyset$

Objective

$\min w(F) = \sum_{e \in F} w(e)$, i.e., minimize the total edge weight of the solution.

For the online Steiner forest problem on graphs, the input is split similarly to the Steiner tree variant. In fact, the offline portion of the input is exactly the same. However, the online request sequence is now pairs of terminals, $R = (p_1, p_2, \ldots, p_n)$ where $p_i = (s_i, t_i)$ for $s_i, t_i \in V \; \forall i \in [n]$.

Problem Definition 7 Online Steiner Forest

Input

$G = (V, E, w)$ an edge-weighted graph $w : E \to \mathbb{R}^+$, offline.
$P = \{p_1, ..., p_n\}$ terminal pair sequence $p_i \in V \times V$ for $i \in [n]$ online.

Output

$F_i \subseteq E$ on input p_i, set of hyperedges connecting the terminal pairs $p_j, 1 \leq j \leq i$.

Objective

$\min w(F) = \sum_{e \in F} w(e)$, i.e., minimize the total edge weight of the solution, where $F = \bigcup_{i=1}^{n} F_i$ a sub-hypergraph of H containing all requested terminals.

In Chapter 4 we define a generalization of the Steiner tree problem where the input graph is instead a hypergraph, and in Chapter 5 we similarly consider the Steiner forest generalization in the hypergraph setting.

When dealing with ordinary graphs, if we consider only positive edge-weighted graphs, the optimal solution is necessarily a tree. It is noteworthy that in the hypergraph setting we cannot in general restrict the solution to be a hypertree while maintaining connectivity of the requests. Figure 2.1 demonstrates how it is possible that an input instance has no hypertree solutions.

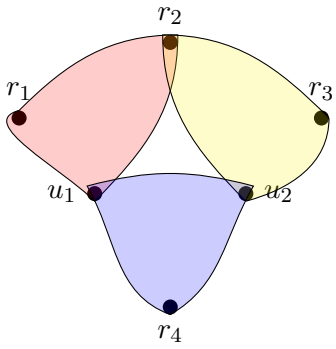

Figure 2.1: An example input where the solution cannot be a hypertree.

The vertices of the hypergraph $H = (V, E, w)$ in Figure 2.1 are shown as dots labeled r_1, r_2, r_3, u_1, u_2 and r_4. The hyperedges are represented by the colored regions enclosed by curves: $\{r_1, r_2, u_1\}, \{r_2, r_3, u_2\}$

and $\{u_1, u_2, r_4\}$. For the online request sequence $R = (r_1, r_2, r_3, r_4)$, the only solution is to buy all edges which introduces a cycle $(r_1, r_2, r_3, u_2, r_4, u_1, r_1)$. The fact that OPT is always a tree does not generalize to the hypergraph case, where even the offline problem of deciding if a given set of terminals has a hypertree solution is *NP-complete* [38]. For this reason we do not include the constraint that the solution must be a hypertree, but view it as a result of the minimization constraint in ordinary graphs.

2.7 The Greedy Family of Algorithms

The greedy algorithm is not in fact a single algorithm, but it is a family of algorithms which evade strict formal definition. A simple way to think of them is as algorithms which make decisions based on some optimal choice in the moment. So, for a minimization problems like the Steiner problems in graphs, the greedy algorithm can be one which selects the shortest path (the path of minimal edge weight) to satisfy the connection requirements and includes those edges in the solution. The shortest paths are computed between the current request and all previous requests for the current step. Since the solution is a tree this satisfies the connection requirements perfectly well.

Algorithm 8 Online Greedy Steiner Tree Algorithm

```
1: procedure GreedyST(G = (V, E, w), R)
2:     i ← 2
3:     while i ≤ n do
4:         Compute the shortest path P in G connecting r_i to r_1        ▷ P ⊆ E
5:         S ← S ∪ P
6:         for e ∈ P do
7:             w(e) ← 0
8:         i ← i + 1
9:     return S
```

The greedy algorithm for the online Steiner forest problem now needs to connect pairs of vertices in each request, so it only needs to compute the shortest path between each pair of nodes as they arrive. The connection requirements are more strict in the sense that the set of paths to choose from is much smaller, but less strict in that the final solution can be a tree, where all requests are

connected to each other, or a set of paths connecting only the pairs within each request, or anything in between.

Algorithm 9 Online Greedy Steiner Forest Algorithm

```
1: procedure GreedySF(G = (V, E, w), R)
2:     i ← 1
3:     while i ≤ n do
4:         Compute the shortest path P in G connecting s_i to t_i        ▷ P ⊆ E
5:         S ← S ∪ P
6:         for e ∈ P do
7:             w(e) ← 0
8:         i ← i + 1
9:     return S
```

When adapting these greedy algorithms to accept hypergraphs as input instead of just ordinary graphs, the main difference is in the implementation to compute the shortest paths. For example, using $HDijkstra$ (Algorithm 5) instead of $Dijkstra$ (Algorithm 4). The term $Greedy$ is used in Chapter 4 to refer to the adaptation of Algorithm 8 to hypergraphs, and in Chapter 5 to refer to the adaptation of Algorithm 9 to hypergraphs.

2.8 Linear Programming and Primal-Dual Analysis

Linear programming (we refer the interested reader to the classical text [21] for a gentle introduction) not only has practical applications in many industries, but is also a useful analytical tool. A *linear program* (LP) comprises a linear equation, called the objective function, and a set of linear inequalities, called the constraints. The goal is to optimize the objective function (maximize of minimize) while satisfying all the constraints. For variables x_i and coefficients c_{ij} a minimization

LP would look like this,

$$\text{minimize: } \sum_{j=1}^{n} c_j x_j$$
$$\text{subject to: } \sum_{j=1}^{n} a_{ij} x_i \geq b_i \text{ for } i \in [m]$$
$$x_j \geq 0 \text{ for } j \in [n]$$

where $[n] = 1, 2, ..., n$. A solution which satisfies all the constraints is called *feasible*, and one which is feasible and obtains the optimal value is called an optimal solution. Related to this LP is another, called the dual LP (the original can thus be called the primal LP). When the primal LP is a minimization problem, then the dual LP is a maximization problem and vice-versa. The dual for the above example looks like this,

$$\text{maximize: } \sum_{i=1}^{m} b_i y_i$$
$$\text{subject to: } \sum_{i=1}^{m} a_{ij} y_i \leq c_j \text{ for } j \in [n]$$
$$y_i \geq 0 \text{ for } i \in [m]$$

We will sometimes employ the vector notation for the above. Observe that the objective function and each constraint have the form of a vector dot product. If we define a vector $\vec{c}^T = (c_1, c_2, \ldots, c_n)$ to represent the coefficients in the objective function and $\vec{x}^T = (x_1, x_2, \ldots, x_n)$ to represent the variables then we can obtain the objective value by computing the vector product $\vec{c}^T \cdot \vec{x}$. Similarly, we define for the vector for the RHS of the constraints $\vec{b}^T = (b_1, \ldots, b_m)$ and the dual variables $\vec{y}^T = (y_1, \ldots, y_m)$. The objective value of the dual is the same as the product $\vec{b}^T \cdot \vec{y}$. The constraint coefficients can be obtained by defining a matrix $A_{m \times n} = [a_{ij}]$, then the dual constraints matrix is simply the transpose $A^T = [a_{ji}]$ and the primal and dual constraints can be expressed, respectively, as follows

- primal constraints: $A\vec{x} \geq \vec{b}$

- dual constraints: $A^T\vec{y} \leq \vec{c}$

An important result in the study of linear programming is the strong duality theorem, which establishes that the optimal solutions to the primal and dual (when they exist) are equal.

Theorem 2.8.1 (Strong duality theorem)**.** *Let $\vec{x}$ be an optimal variable assignment to a LP, then there exists an optimal variable assignment $\vec{y}$ for the corresponding dual and*

$$\vec{c}^T \cdot \vec{x} = \vec{b}^T \cdot \vec{y}$$

i.e., the optimal objective values of the primal and dual LP are equal.

One direction of the above theorem is known as the weak duality theorem, namely, the inequality $\vec{c}^T \cdot \vec{x} \geq \vec{b}^T \cdot \vec{y}$, which holds more generally whenever $\vec{x}$ and $\vec{y}$ are feasible solutions to primal and dual (regardless of whether optimal solutions exist or not).

When the variables are further constrained to take on only integer values, the corresponding program is called an *integer* linear program, or *integer program* (IP) for short. An integer program can be relaxed, that is to say the integrality constraints removed so that the variables can take on any real value within the same bounds, and the result is called the *LP-relaxation* of the IP. If the optimal value to a minimization IP is z_{IP} and to the corresponding LP-relaxation is z_{LP}, then the value $\frac{z_{LP}}{z_{IP}}$ is called the *integrality gap* between the IP and its LP-relaxation. Let z^*_{IP} and z^*_{LP} be optimal solutions to the dual IP and dual LP-relaxation, respectively. Since every IP is also an LP, we can apply the weak-duality theorem to determine that the inequality $z^*_{IP} \leq z^*_{LP}$ holds. Also, let ALG be the approximate (feasible) solution of some online algorithm to the primal IP.

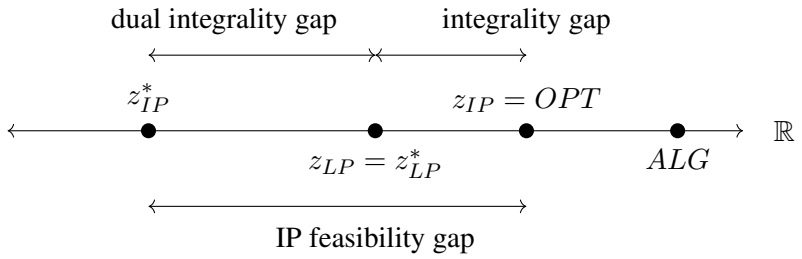

Figure 2.2: Relations between the optimal objective values for an IP, its LP-relaxation, their respective dual solutions, and an arbitrary approximation for the objective value of the primal IP.

So given a feasible solution to the primal minimization IP, it is possible to bound the competitive ratio of an online algorithm by the ratio of the value obtained by that algorithm, ALG, over the value of some offline feasible dual solution for the LP-relaxation, z^*_{LP}, i.e.,

$$1 \leq \frac{ALG}{z_{IP}} \leq \frac{ALG}{z^*_{LP}}.$$

Interestingly, this gives us a way to possibly determine the competitive ratio of an online algorithm without any knowledge or assumptions about an offline optimal solution.

The following is a very useful fact about primal-dual analysis,

Theorem 2.8.2. *If a feasible primal solution (for a minimization LP) can be partitioned into α classes such that for each class we can construct a feasible dual solution that achieves objective value within factor β of the corresponding primal objective value (for that class), then the ratio between the primal solution and the total sum of feasible dual solution objective values is at most $\alpha \cdot \beta$.*

This theorem gives us a tool to analyze LP problems without worrying about optimal objective values, so it will consequently prove quite useful. In particular, it will allow us to bound the competitive ratio of an online algorithm for an LP primal by simply providing an algorithm for the dual LP.

Proof. Let $\vec{x}$ be a feasible variable assignment for the primal LP, and partition this assignment into

α classes $\vec{x}_i$ for $i \in [\alpha]$. Clearly each $\vec{x}_i$ is feasible. Then

$$\vec{c}^T \cdot \vec{x} = \sum_{i=1}^{\alpha} \vec{c}^T \cdot \vec{x}_i.$$

Now let $\vec{y}_i$ for $i \in [\alpha]$ be a collection of feasible dual variable assignments corresponding to the variables assigned in each class $\vec{y}_i$. Since each such dual solution is feasible and each class of the primal solution is feasible, each dual solution is bounded above by the corresponding primal solution. Let β be the maximum such bound, then

$$\vec{b}^T \cdot \vec{y}_i \leq \beta(\vec{c}^T \cdot \vec{x}_i)$$

Now if we sum over each dual solution we get that

$$\sum_{i=1}^{\alpha} \vec{b}^T \vec{y}_i \leq \sum_{i=1}^{\alpha} \beta(\vec{c}^T \cdot \vec{x}_i) \leq \sum_{i=1}^{\alpha} \beta(\vec{c}^T \cdot \vec{x}) = \alpha\beta(\vec{c}^T \cdot \vec{x})$$

□

Chapter 3

Related Work

In this chapter we briefly visit some previous work relevant to this thesis. We begin in Section 3.1 with the first occurrence of the Steiner problem in graphs, where the input is given offline. Following this, in Section 3.2 we look at the main results from the online setting of the Steiner problem in graphs. In Section 3.3 we look at some work done in the generalized online Steiner problem (still in graphs). Finally, in Section 3.4 we see some interesting results for some related offline problems in the hypergraph setting.

3.1 Offline Steiner Problems

The earliest formulation of the Steiner problem in graphs is attributed independently to Hakimi [36] and Levin [47] in 1971. It was later revealed by Karp [42] to be NP-complete in 1972. Many exact and approximation algorithms for the Steiner problem in graphs are known. Dreyfus and Wagner [26] provided a first exact dynamic programming algorithm for the problem with time complexity $O(3^{|R|}|V|)$, where R is the set of terminals and V the set of vertices, which has since been improved (Björklund et al. [14], Nederlof [52], and Vygen [64]). The best combinatorial approximation algorithm so far is by Robins and Zelikovsky [57], with an approximation ratio of $1.55 + \epsilon$. The best approximation ratio to date is $1.39 + \epsilon$ and was first discovered by Byrka et al. [17], then

the same bound was achieved with a more efficient algorithm by Goemans et al. [33]. The simpler approaches of greedy and MST heuristic give approximation ratios of 2. The LP based algorithm by Goemans and Williams give a $2 - \frac{2}{|R|}$ approximation. A recent survey was prepared by Ljubić [48] going into further detail on the latest advances in the offline variants of the problem. For a survey of earlier work on the problem, there are surveys by Maculan [49], Winter [66], or Voß [27].

Many variants on the Steiner tree problem exist, including the priority [59], prize-collecting [54, 41], and node-weighted [60, 45, 51, 22] and group Steiner tree problems [22, 32]. In the priority Steiner tree problem, terminals have an associated bandwidth requirement which must be met via the edges connecting them to the root node. Each edge has a bandwidth capacity as well as a cost, and the goal is to minimize the total edge cost of a solution where there is a path of edges of at least the required minimum bandwidth connecting each terminal to the root, or between the request pairs, for the Steiner tree and Steiner forest problems, respectively. The node-weighted Steiner tree problem is one where the cost function assigns a value to the vertices of the graph instead of the edges. the goal is to minimize the total cost for the Steiner points of the solution, since the cost of the terminals is the same for any solution of a given instance. In the prize-collecting Steiner tree problem, we have two cost functions. The first is on the vertices, and the second is on the edges. The goal here is to optimize the difference between the total node weight (the prizes) and the total edge cost.

Some related problems are facility location [20, 61] and survivable network design problems [43]. The facility location problem we have two types of terminals, the first are the facilities and the second are the customers. Each node has a given cost to open a facility there and the facilities should be interconnected by a Steiner tree. The cost to connect customers to the appropriate facility also has a certain cost. The survivable network design problem is concerned with the robust network connection. Given an edge-weighted graph $G = (V, E, w)$, a set of terminals R, and an integer assigned to each pair of terminals $d_{uv} \forall u, v \in R$, we want to find a minimum cost subgraph containing at least d_{uv} edge-disjoint paths between terminals $u, v \in R$.

3.2 Online Steiner Tree

The Online Steiner Tree Problem in graphs was first posed by Imase and Waxman [39]. They proved the upper and lower bounds to establish a competitive ratio of $O(\log n)$. They also considered a dynamic variant of the problem where terminal requests could be to add or remove terminals from the subgraph. Alon and Azar [2] considered the online Steiner problem in the Euclidean plane in 1993. This is an online formulation of the classical Steiner problem in Euclidean geometry. They found that there is a lower bound of $\Omega(\frac{\log n}{\log \log n})$ and give an alternate proof for the same upper bound from Imase and Waxman (there remains a gap of $\log \log n$ between the two bounds). Garg et al. [31] analyzed the problem in i.i.d. stochastic settings. In the i.i.d. model, the adversary specifies a distribution on the input requests, and the requests given to the algorithm are sampled i.i.d. from the distribution. There are two kinds of the i.i.d. model – known and unknown, indicating whether the distribution chosen by the adversary is known to the algorithm in advance or not. Garg et al. [31] showed that the competitive ratio is $O(1)$ in the known i.i.d. model and almost logarithmic lower bounds for the setting of unknown i.i.d. model, as well as a stochastic model without the independence assumption. In the random order model, the adversary selects a set of requests and they are presented to the algorithm in (uniformly sampled) random order. Note that upper bounds on the competitive ratio in the random order model carry over to the i.i.d. models. Thus, the unknown i.i.d. lower bound of Garg et al. [31] carries over to the random order setting, as well.

The online version of many variants of the Steiner problem in graphs have also been considered, such as the priority [4], node-weighted [4] and prize-collecting [35] Steiner problems. In the online versions of these problems, the only difference is that the terminal set arrives as a request sequence, all other input is given offline.

3.3 Online Steiner Forest

The generalized OST problem, or online Steiner forest (OSF), was studied by Awerbuch et al. [6] and Berman and Coulston [13]. In [6] it was established that the simple greedy algorithm, which

they call *min-cost*, is $O(\log^2 n)$-competitive. They conjectured that *min-cost* is actually $O(\log n)$-competitive. Berman and Coulston [13] introduced a non-greedy primal-dual based algorithm which is $O(\log n)$-competitive. Some advancement on the analysis of the greedy algorithm for the Steiner forest problem was made by Bamas et al. [9], showing that when the algorithm does not benefit from re-using edges from the current solution, the competitive ratio becomes $O(\log n \log \log n)$. It is not shown that these instances (where the contraction is 1) are worst-case, but it is mentioned that all known worst-case instances are so.

3.4 Similar Problems on Hypergraphs

The offline variant of the Steiner problem in hypergraphs was recently introduced by Hörsch and Szigeti [38] in 2023. They provide some proofs for the NP-Completeness of the problem and that of some related problems. The problem they consider differs from that of this thesis not only in that it is in the offline setting, but also in that they constrain the solution to be a hypertree. To be specific, they show (among other things) that it is an NP-complete problem to decide, given a hypergraph $H = (V, E)$ a set of terminals $T \subseteq V$ and an integer k whether there is a subhypertree H_T of weight at most k of H where $T \subseteq V(H_T)$.

In the set cover problem [8, 18, 28], we are given a ground set of points and a family of subsets. The goal is to select a minimum number of the subsets such that the set of points is contained in the union. In the weighted version of the problem, the subsets have an assigned cost and the goal is to minimize sum of costs of the subsets which cover the set of points. This problem is very easily modeled in hypergraphs, where the ground set represents vertices, the family of subsets are hyperedges, and the subset of points to be covered are the terminal nodes. Greedy and LP-based algorithms provide an $O(\ln n)$ approximation for Set Cover [19, 21, 63], and multiple seminal works established that no significantly better approximation is possible unless fundamental complexity assumptions fail. Johnson's 1974 [40] algorithmic guarantee and Feige's 1998 hardness threshold [28] together pinpointed $\Theta(\ln n)$ as the tight approximation ratio for Set Cover unless NP has slightly subexponential time algorithms. The subsequent results by Raz-Safra [56], Alon et al. [3], and

Dinur-Steurer [24] showed that Set Cover's approximability is settled at roughly the logarithmic scale – improving beyond a $\ln n$ factor is infeasible in general – under the standard assumption of P not equal to NP. Various restricted versions (bounded frequencies or set sizes) also have matching logarithmic thresholds, further confirming that the known $\ln$-factor algorithms are essentially best possible [23, 44]. The difference between this problem and the Steiner cover problem is that the solution does not have any connectivity requirements, a single edge can be selected to cover any given terminal, whereas in the Steiner cover problems a path of hyperedges must be selected connecting each terminal to either the root node or the associated vertex from the request pair (for the original Steiner cover and generalized Steiner cover problems, respectively). The online set cover problem has also been studied in [1, 25, 46]. In the online version we do not necessarily need to cover all the points, and so the terminal points to be covered arrive online. Alon et al. [1] provided an algorithm which is $O(\log m \log n)$ competitive, where m is the number of subsets and n the number of terminals, and proved a lower bound of $\Omega\left(\frac{\log m \log n}{\log\log m + \log\log n}\right)$.

The minimum spanning tree problem in hypergraphs (MSTH) [55, 65] represents an extension of the minimum spanning tree (MST) in ordinary graphs. The goal is to find a subhypergraph H' of H that is a hypertree such that $V(H') = V(H)$, i.e., every vertex of H appears in H' and H' is connected.

In [11] the minimum spanning subhypergraph problem (MSSH) is considered, where the goal is to find a minimum weight subhypergraph. This is similar to the MSTH, where the result need not be a hypertree. This is a special case of the offline Steiner cover problem in hypergraphs (the offline variant of the problem considered in this thesis), where the set of terminals is the entire vertex set. The difference between the Steiner tree problem in hypergraphs being that the result is no longer constrained to be a hypertree. Baudis [11] showed that the problem admits a $O(\ln k)$-approximation for k-restricted hypergraphs.

An LP approach to the survivable network design problem in hypergraphs by Zhao et al. [69] showed a $O(k \cdot \ln r_{\max})$ approximation, where k is the maximum edge degree (rank of the hypergraph) and $r_{\max}$ is the maximum connectivity requirement. In this problem we are given an

edge-weighted hypergraph $H = (V, E, w)$ and each pair of vertices $u, v \in V$ has an assigned integer value r_{uv} representing the number of hyperedge-disjoint paths one must buy in order to connect them. This problem can be adapted to a further generalization of the (offline) Steiner problem in hypergraphs, where we would have $r_{max} = 1$, if the connectivity requirement parameter r were allowed to take on the value 0 for pairs containing a non-terminal vertex.

A directed hypergraph is a generalization of a hypergraph in which edges go from a subset of source nodes to a single target node. A good introduction on this topic can be found in [29, 5]. An interesting change from the undirected setting is that shortest path problems become NP-hard in directed hypergraphs.

Chapter 4

Online Steiner Cover on Hypergraphs

In this chapter, we formally define the online Steiner cover on hypergraphs problem (OSCH) and present an online greedy algorithm (Section 4.1), which we denote by $Greedy$. Then we prove a simple upper bound of n on the competitive ratio for $Greedy$ on all hypergraphs, followed by $O(k \log n)$ upper bound on the competitive ratio of deterministic algorithms when the input hypergraph is of rank k (Section 4.2). The same bound is then given in another way, using LP duality to bound $Greedy$ with an algorithm for the dual of the relaxed LP for the problem (Section 4.3). We end the chapter with a presentation of two lower bounds on the competitive ratio for any deterministic algorithm; a simple lower bound of k for hypergraphs of rank k, and an improved lower bound of $\Omega(k \log n)$ matching the upper bound of $O(k \log n)$ (Section 4.4).

4.1 The Online Steiner Cover and $Greedy$ Algorithm on Hypergraphs

In OSCH, we have as input an edge-weighted hypergraph $H = (V, E, w)$ where $|V| = N$, $E \subseteq 2^V$ (offline) and a subset of vertices arriving sequentially (online), $R = (r_i)_{i=1}^{n}$, where $r_i \in V$ for all $i \in [n]$. We can consider the first input r_1 to be the root, upon its arrival there are no edges which need to be added to the solution. The output of our algorithm after the i^{th} request should be a set of edges $F_i \subseteq E$ describing a connected sub-hypergraph of H which contains all previously

requested terminals. Recall that a hypergraph is connected when there is a path between any pair of its vertices, so there should be a path between any two requested terminals along some subset of F_i. After each request arrives, the algorithm must immediately select edges which connect it to the current solution, so $F_{i-1} \subseteq F_i$ for $2 \leq i \leq n$ and $F_1 = \emptyset$. In other words, upon arrival of request i, the algorithm must connect r_i to the solution obtained after the previous request, F_{i-1}, thus maintaining connectivity of the solution in each step. After the last request is processed by the algorithm, we have the final solution $F = F_n$.

Problem Definition 10 Online Steiner Cover on Hypergraphs

Input

$H = (V, E, w)$ – hypergraph with edge weights $w : E \to \mathbb{R}^+$; given offline.
$R = (r_1, ..., r_n)$ – request sequence where $r_i \in V \ \forall i \in [n]$; given online.

Output

$F_i \subseteq E$ on request r_i, s.th. $\bigcup_i F_i$ is a connected component with
$\forall r \in R \ \exists e \in \bigcup_{i=1}^{n} F_n$ such that $r \in e$ and $\bigcap_{i=1}^{n} F_i = \emptyset$

Objective

$\min w(F) = \sum_{e \in F} w(e)$, i.e., minimize the total edge weight of the solution.

Since an edge does not need to be bought multiple times, i.e., once it is included in the solution by an algorithm it can be reused to connect future requests at no extra cost, we can define the updated hypergraphs $H_i = (V, E, w_i)$ after the i^{th} request is satisfied, where we have

$$w_i(e) = \begin{cases} w(e) & \text{if } e \notin F_i \\ 0 & e \in F_i. \end{cases}$$

4.2 Upper Bounds on $Greedy$

We begin by showing a simple upper bound of n on the competitive ratio of $Greedy$ for OSCH. Following this, we improve the bound by adapting the analysis of Imase and Waxman [39] from ordinary graphs to k-restricted hypergraphs, or hypergraphs of rank k. The following theorem is proved by observing that the cost of an optimal solution is at least the minimum distance between any two requests. Since there are n requests in total and $Greedy$ connects requests by minimum

distances between them, we can bound the total cost of $Greedy$ to satisfy the request sequence of length n by n times the cost of an optimal solution.

Theorem 4.2.1. *For any instance of OSCH with input hypergraph H and request sequence $R, |R| = n$, we have*

$$Greedy(R) \leq n \cdot OPT(R),$$

that is, $\rho(Greedy) \leq n$.

Proof. $Greedy$ adds the edges which minimize the cost to connect each request to the current solution. Let request r_{max} be the request furthest from the root node for some instance of the Steiner problem in hypergraphs; then the cost for any optimal solution is at least $\mathrm{dist}_H(r_{max}, r_1)$, where r_1 is the root. Since $Greedy$ connects each request to the root, the cost for each request connected is bounded above by the cost connecting r_{max} to the root.

$$Greedy(R) \leq \sum_{i=1}^{n} \mathrm{dist}_{H_{i-1}}(r_i, r_1) \leq \sum_{i=1}^{n} \mathrm{dist}(r_{max}, r_1) \leq n \cdot OPT(R)$$

So the competitive ratio for the greedy algorithm is at most n, the number of requests. □

In the following theorem we establish $O(\mathrm{rank}(H) \log n)$ upper bound by using a more intricate argument. Notice that in most cases the second bound is stronger. The first bound of n outperforms the second bound for hypergraphs of very large rank, i.e., when $\mathrm{rank}(H) \geq n/\log n$. We want to bound the cost of $Greedy$ by comparing the minimum distances between a request r_i and those that appeared before it in H, the input hypergraph, to the cost of H_{OPT}, the subgraph induced by including the edges in an optimal solution and excluding the rest. Two things are immediately clear: *(i)* every request appears in $V(H_{OPT})$, and *(ii)* the minimal distances in H are at most the minimal distances in H_{OPT}, i.e., $\mathrm{dist}_H(r_i, r_j) \leq \mathrm{dist}_{H_{OPT}}(r_i, r_j)\ \forall i, j \in [n]$. Then if we can somehow relate the minimal distances $\min_{j<i} \mathrm{dist}_{H_{OPT}}(r_i, r_j)$ for $i, j \in [n]$ to the total cost of H_{OPT}, then we would have a link between the cost of $Greedy$ and the cost of OPT. To obtain such a relation, we construct a path graph L_n by $HDFS$ traversal of H_{OPT} satisfying a recursive structure which allows it to be shown in Lemma 4.2.3 [39] that the sum of minimal distances $\min_{j<i} \mathrm{dist}_{L_n}(r_i, r_j)$

for $i, j \in [n]$ is bounded above by a factor $\log n$ of the weight of L_n. Combined with the fact that the $HDFS$ traversal step in the construction of L_n guarantees that its weight is at most k times that of H_{OPT}, with $rank(H) = k$, we get the desired bound on $Greedy$.

Theorem 4.2.2. *Let $H = (V, E)$ be a hypergraph of rank k and R a request sequence of length n for OSCH, then:*

$$Greedy(R) \leq (k-1) \log n \cdot OPT(R)$$

i.e., $\rho(Greedy) \leq (k-1) \log n$.

Proof. Consider an arbitrary instance of $OSCH$ where H is the input hypergraph and R is the request sequence. Then if OPT is some offline optimal algorithm, let $OPT(R)$ be the value of the solution found by OPT, and let F_{OPT} be the set of edges selected by OPT on this instance. Now, let H_{OPT} be the sub-hypergraph induced on H by taking as edges those from F_{OPT} and the vertices to be those incident on some edge in F_{OPT}. Finally: let $\text{dist}_H(u, v)$ be the minimum distance between nodes $u, v \in V(H)$ in H and denote the distance between nodes $u, v \in H_{OPT}$ by $\text{dist}_{OPT}(u, v)$.

In order to prove the upper bound, we will adjust the argument of [39] for Steiner trees, generalizing for hypergraphs of rank k. To begin, define an ordering (a permutation) of the requested vertices on the nodes of OPT using the *HDFS* algorithm to traverse H_{OPT}. This traversal reveals a path structure which preserves the distances between requests, so it can be used to analyze the cost of $Greedy$ relative to the cost of OPT. To construct this path, we add a node each time a requested terminal is traversed by the *HDFS* algorithm *for the first time*. Each request is connected to the last traversed request by a single edge whose weight is the sum of the edge weights along the traversal between them. Any Steiner nodes (non-request vertices) visited between requests in the traversal are not added to the path. Note that the traversal may cross the same edge multiple times before reaching the next unvisited request, in which case its weight is added to the sum each time it is traversed. See Figure 4.1 for an illustration of the line graph construction on ordinary graphs and Figure 4.2 for the generalized image in hypergraphs. If the rank of the hypergraph is k, then it will traverse any given edge at most $2(k-1)$ times, once per visit for the (at most $k-1$) vertices

other than the source of the traversal incident to that edge and again to backtrack. Then if we denote the resulting path on n vertices by L_n, the weight of this path can be found by summing the edge weights as follows

$$w(L_n) = \sum_{e \in E(L_n)} w(e) = \sum_{e \in F_{OPT}} (k-1) \cdot w(e) = (k-1) \cdot OPT(R)$$

Note that in this case the distance between some pair of requests r_1, r_2, where r_1 was requested before r_2 might not be the minimum distance between the two in H_{OPT}, but it is an upper bound in the path graph, so the argument still follows. Even when applied to hypergraphs, the traversal yields an ordinary path structure. The remainder of the proof follows exactly that of [39], which is repeated here for completeness.

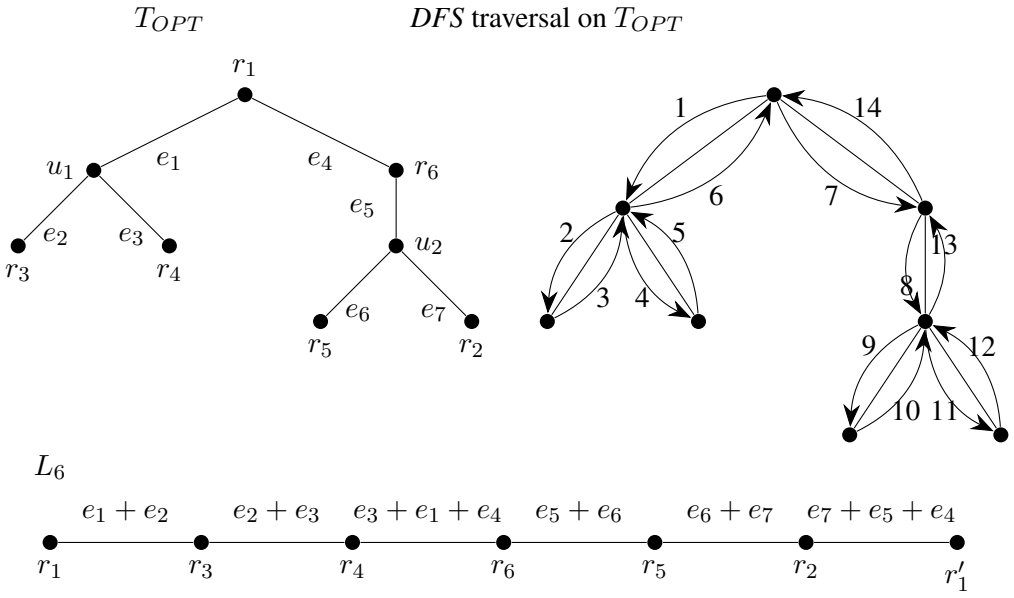

Figure 4.1: Illustration of how the *DFS* algorithm is used to construct the line graph L_6 from T_{OPT}.

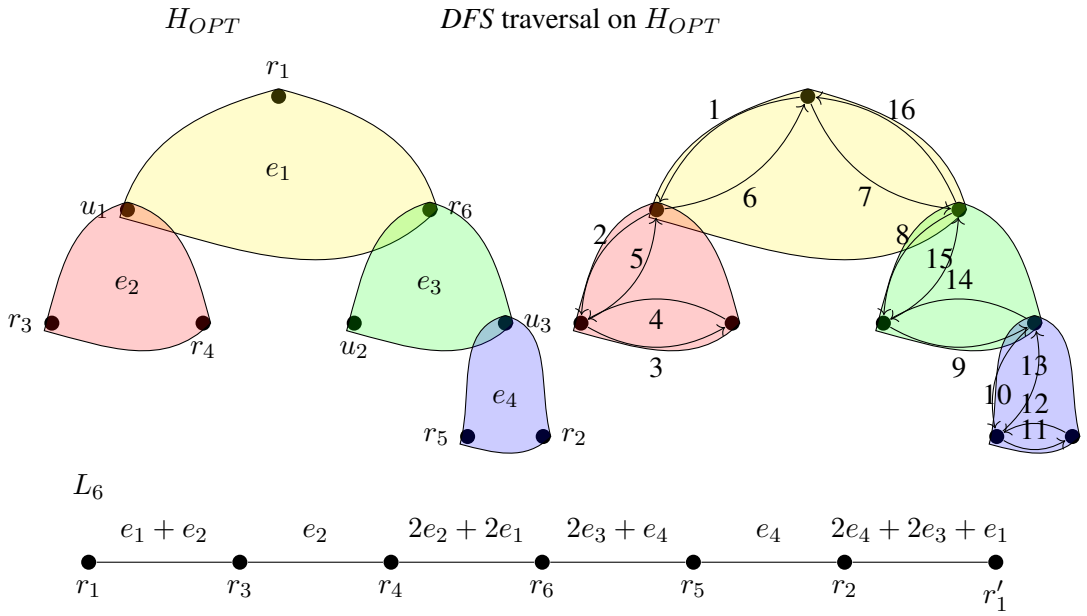

Figure 4.2: Illustration of how the *HDFS* algorithm is used to construct the line graph L_6 from H_{OPT}.

Let π denote the permutation of the request sequence in L_n so that $\pi(r_i) = r_{\pi_i}$, i.e., the i^{th} request is the π_i^{th} which is encountered in the traversal. The weight of the edges connecting adjacent vertices in L_n is equal to the sum of edge weights between those requests in the traversal of OPT. Once the final request r_{π_n} is visited and therefore added to L_n, we add a copy of the root r'_{π_1} (where the traversal was initiated) to the end of the path, with the edge weight connecting it being the cost of the path from r_{π_n} to the root in OPT, $\mathrm{dist}_{OPT}(r_{\pi_n}, r_1)$.

In L_n, what we are interested in is the minimal distances between each request r_i $1 < i \leq n$ and the previous requests, $\min_{1 \leq j < i} \mathrm{dist}_{L_n}(r_i, r_j)$. This is because $Greedy$, for each request, adds the shortest path to connect it to the current solution. Therefore, if we can bound the sum of these distances in L_n it will give us an upper bound for $Greedy$. The reason that a copy of the root node is added to the opposite end of the path L_n is that this imposes a recursive structure, so that each intermediate node can connect to a previous request either along the path towards r_{π_1} or towards r'_{π_1}. In other words, each request after the first is enclosed by earlier requests in either direction along L_n. If we initiate the traversal at the first request, then $\pi_1 = 1$, then no intermediate node of

L_n was requested before the end nodes. Similarly, if we split the path in two at the second request r_2 (wherever it lies along the path) then we get two paths where still no intermediate nodes were requested before the end nodes (r_1 and r_2). So, adding up the distances

$$\sum_{i=2}^{n} \min_{1\le j<i} \operatorname{dist}_{L_n}(r_i, r_j) = \min_{1\le j<2} \operatorname{dist}_{L_n}(r_2, r_j) + \sum_{i=3}^{n} \min_{1\le j<i} \operatorname{dist}_{L_n}(r_i, r_j).$$

Where the summation on the right consists of the remaining requests on either side of r_2 along L_n. If we let L_{ij} denote the segment of L_n between request i and j, and $\sigma(L_{ij})$ as the sums over $\min_{1\le j<i} \operatorname{dist}_{L_{ij}}(r_l, r_j)$ for r_l earliest requested intermediate node between r_i and r_j (r_i and r_j must be still earlier requests than r_l), then we can describe the summation recursively as follows,

$$\sigma(L_{ij}) = \begin{cases} \min\{\operatorname{dist}_{L_{ij}}(r_l, r_i), \operatorname{dist}_{L_{ij}}(r_l, r_j)\} + \sigma(L_{il}) + \sigma(L_{lj}) & \text{If } \exists \text{ intermediate node } r_l \in L_{ij} \\ 0 \qquad \text{otherwise} \end{cases}$$

Then by induction on the number of intermediate nodes along the path segment, the following is shown,

Lemma 4.2.3 ([39]). *For a path graph with end nodes u, v, and with m intermediate nodes,*

$$\sigma(L_{uv}) \le \frac{\log m + 1}{2} \operatorname{dist}_{L_{uv}}(u, v)$$

Proof. The base case of the induction is when $m = 0$, in which case both sides are 0 and the inequality holds. Now we assume that for any path with $0 \le m' < m$ intermediate nodes the inequality holds, and let r_k be the earliest requested intermediate node, and consider some path L_{uv} with m intermediate nodes. We can further assume that $\operatorname{dist}_{L_{uv}}(r_l, u) \le \operatorname{dist}_{L_{uv}}(r_l, v)$, so we get that

$$\sigma(L_{uv}) = \operatorname{dist}_{L_{uv}}(r_l, r_u) + \sigma(L_{ul}) + \sigma(L_{lv})$$

Now let m_1 and m_2 be the number of intermediate nodes in $\sigma(L_{ul})$ and $\sigma(L_{lv})$ respectively. Since r_l is not intermediate in either of these path segments, the number of intermediate nodes between

the two adds up to $m_1 + m_2 = m - 1$. This implies that

$$\begin{aligned}\sigma(L_{uv}) &\leq \operatorname{dist}_{L_{uv}}(r_l, u) + \frac{\log m_1 + 1}{2} \operatorname{dist}_{L_{uv}}(r_l, u) + \frac{\log m_2 + 1}{2} \operatorname{dist}_{L_{uv}}(r_l, v) \\ &= \operatorname{dist}_{L_{uv}}(r_l, u) + \frac{\log m_1 + 1}{2} \operatorname{dist}_{L_{uv}}(r_l, u) + \\ &\quad + \frac{\log m - m_1}{2} (\operatorname{dist}_{L_{uv}}(u, v) - \operatorname{dist}_{L_{uv}}(r_l, u))\end{aligned}$$

Since this expression is a linear function of $\operatorname{dist}_{L_{uv}}(r_k, v)$, its maximum will be achieved at one of the boundary values *(i)* $\operatorname{dist}_{L_{uv}}(r_k, v) = 0$ or *(ii)* $\operatorname{dist}_{L_{uv}}(r_k, v) = \frac{\operatorname{dist}_{L_{uv}}(u,v)}{2}$.

$$\begin{aligned}(i) \quad & \operatorname{dist}_{L_{uv}}(r_l, v) = 0 \\ & \Rightarrow \sigma(L_{uv}) \leq \frac{\log m - m_1}{2} \operatorname{dist}_{L_{uv}}(u, v) \\ & \leq \frac{\log m + 1}{2} \operatorname{dist}_{L_{uv}}(u, v) \\ (ii) \quad & \operatorname{dist}_{L_{uv}}(r_l, v) = \frac{\operatorname{dist}_{L_{uv}}(u, v)}{2} \\ & \Rightarrow \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{2} + \frac{\log m_1 + 1}{2} \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{2} + \frac{\log m - m_1}{2} \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{2} \\ & = \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{4} (2 + \log m_1 + 1 + \log m - m_1) \\ & = \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{4} (\log 4(m_1 + 1)(m - m_1)) \\ & = \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{2} \left(\log 2\sqrt{(m_1 + 1)(m - m_1)}\right) \\ & \leq \operatorname{dist}_{L_{uv}}(u, v)\frac{1}{2} \log m_1 + 1 + m - m_1 = \frac{\log m + 1}{2} \operatorname{dist}_{L_{uv}}(u, v)\end{aligned}$$

□

Since the path L_n rooted at r_1 satisfies the above conditions with $n - 1$ intermediate nodes, we can conclude that the sums of minimal distances for each request to the previous requests in L_n is less than $\frac{\log n}{2} w(L_n)$. And since the weight of the path is $w(L_n) = 2(k - 1) \cdot OPT(R)$, we have that

$$Greedy(R) \leq \frac{\log n}{2} \cdot w(L_n) \leq (2(k-1))\frac{\log n}{2} \cdot OPT(R) = (k-1) \log n \cdot OPT(R)$$

For an instance with n requests on k-restricted hypergraphs. □

4.3 LP Analysis of $Greedy$

Here we analyze the $Greedy$ algorithm from the perspective of linear program duality. The goal of this section is to provide an alternative proof to Theorem 4.2.2. The OSCH can be stated first as the following IP,

$$\begin{aligned} \text{min.:} & \sum_{e \in E} x_e w(e) \\ \text{s.t.:} & \sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{T} \\ & x_e \in \{0, 1\} \end{aligned}$$

Where x_e is the value of the characteristic function of the solution for edge $e \in E$, $\mathcal{T}$ is the family of cuts which separate at least one pair of terminals, i.e.,

$$\mathcal{T} = \{S \subset V : 1 \leq |S \cap R| \leq n - 1\}$$

and $\delta(S)$ represents the set of edges crossing some cut $S \in \mathcal{T}$, i.e.,

$$\delta(S) = \{e \in E : e \cap S \neq \emptyset, e \cap (V \setminus S) \neq \emptyset\}.$$

Then the objective is to minimize the total edge weight of the solution, satisfying the constraint that there is at least one edge in the solution which connects a request separating cut S to its complement

$V \setminus S$ in G. If we drop the integrality constraint, we get the following LP-relaxation,

$$\begin{aligned} \text{min.:} & \sum_{e \in E} x_e w(e) \\ \text{s.t.:} & \sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{T} \\ & 0 \leq x_e \leq 1 \quad \forall e \in E \end{aligned}$$

and the dual of this LP is the following,

$$\begin{aligned} \text{max.:} & \sum_{S \in \mathcal{T}} y_S \\ \text{s.t.:} & \sum_{S : e \in \delta(S)} y_S \leq w(e) \quad \forall e \in E \\ & y_S \geq 0 \quad \forall S \in \mathcal{T} \end{aligned}$$

In the dual, y_S is a dual variable corresponding to cut $S \in \mathcal{T}$. The dual is a packing problem, in that a solution should increase each y_S appropriately to maximize their sum, while satisfying the constraint that for each edge $e \in \delta(S)$ which crosses some family of cuts, the sum of their corresponding variables should never increase beyond the cost $w(e)$. In other words, the sum of values assigned to each cut variable crossed by a given edge should never exceed the cost of that edge. We know that the result of $Greedy$ is feasible, where x_e are initialized to 0 and changed to 1 when edge $e \in E$ appears in the solution of the algorithm. Therefore, by the weak duality theorem, we can bound the competitive ratio by the performance of some algorithm to the dual of this relaxed LP. Since the dual algorithm does not need to run any computations online, we can consider the request terminals as a subset of vertices, denoted by $\hat{R} = \cup_{i=1}^{n} \{r_i\}$, instead of the usual subsequence $R = (r_i)_{i=1}^{n}$.

Given the problem setting described above, we continue with a high-level overview of the argument to follow. In order to bound the performance of $Greedy$, we want to compare it to a solution of the dual for the relaxed LP. This is because the dual of the relaxed LP is a lower bound on the dual IP, which is in turn a lower bound on any solution for the primal (and therefore a lower bound on any

optimal solution for the primal). Let C be the solution of some instance with request sequence R of the problem obtained by $Greedy$, and let OPT be some offline optimal algorithm. We begin by partitioning C into cost classes, C_j, where the variables assigned in C_j correspond to the requests satisfied by $Greedy$ incurring a cost of at least 2^{j-1} and less than 2^j. Let the total cost to $Greedy$ of each class be denoted by $Greedy(C_j)$. Note that the total cost of solution C incurred by $Greedy$ is equal to the sum of costs incurred by $Greedy$ for the requests in each C_i, i.e.,

$$Greedy(R) = O\left(\sum_{j=1}^{\log OPT(R)} Greedy(C_j)\right).$$

Recall from the proof of Theorem 4.2.1 that the cost for any single request satisfied by $Greedy$ is bounded above by the cost of OPT, so this partitioning defines $\log OPT(R)$ cost classes. We would like to have that $OPT(R) = n$, as this would bring the factor $\log n$ which we know can be achieved from Section 4.2. This can be achieved by normalization, where we normalize every edge by a factor $\frac{n}{OPT(R)}$. Normalization does not affect the competitive ratio, since we are dividing the sum of edge weights of the solution obtained by $Greedy$ with that obtained by OPT, so the normalizing factor is canceled out. To show this, let F be the edge set for the solution obtained by $Greedy$ and F_{OPT} that obtained by OPT, then

$$\frac{Greedy(R)}{OPT(R)} = \frac{\sum_{e\in F} w(e)}{\sum_{e\in F_{OPT}} w(e)} = \frac{\sum_{e\in F} w(e)\cdot \frac{n}{OPT(R)}}{\sum_{e\in F_{OPT}} w(e)\cdot \frac{n}{OPT(R)}}.$$

Back to the cost classes, the big-O notation comes from the fact that the class C_0 is not included in the sum. These requests cost $Greedy$ less than 1 to connect, and there are at most n of them. Since the cost of OPT was normalized to n, this can only add a constant factor to the competitive ratio. Now the dual will be solved one cost class at a time, so each primal class C_j will have a corresponding dual assignment D_j. This is because it simplifies proving feasibility, as done in Lemma 4.3.3, when considering only requests in a given cost class. The cost of a request in class C_j guarantees that the corresponding dual class can be packed by at least $\frac{2^{j-1}}{k}$ and remain feasible. If we assume two requests of a given class yield infeasible dual assignments, then we can derive a contradiction on the cost $Greedy$ needed to pay to connect them. Once the process for solving the

dual classes is known, it can be used to bound the primal cost classes. Therefore, summing over each class gives a bound on the solution obtained by $Greedy$.

$$Greedy(R) = \sum_{j=1}^{\log OPT(R)} Greedy(C_j) \leq \sum_{j=1}^{\log OPT(R)} \frac{2^{j-1}}{k}(\vec{1}^T \cdot \vec{y}_j)$$

where $\vec{y}_i$ is the dual assignment for class i and $\vec{1}$ is the vector of ones, where every entry is 1. Note that since each dual solution is bounded by $OPT(R)$,

$$\sum_{j=1}^{\log OPT(R)} \frac{2^{j-1}}{k}(\vec{1}^T \cdot \vec{y}) \leq \sum_{j=1}^{\log OPT(R)} \frac{2^{j-1}}{k} OPT(R).$$

Now that the plan has been elucidated, and before we proceed by describing the algorithm for setting our dual variables, we give some useful definitions. We define an *active component* to be any cut S represented by the dual variable y_S, satisfying the following conditions,

$$(i) \qquad 1 \leq |S \cap \hat{R}| \leq n-1$$

$$(ii) \qquad \sum_{S':e \in \delta(S')} y_{S'} < w(e) \quad \forall e \in \delta(S)$$

We further define the *reduced weight* $w'(e)$ of an edge $e \in E$ to be the result of subtracting the sum of dual variables at a given point in the computation from the hypergraph weight $w(e)$ of that edge. This is also known as the *slack* for the corresponding constraint in the dual LP.

$$w(e) - \sum_{S:e \in \delta(S)} y_S$$

It is important to note that a given edge can cross up to k components at any given time in the computation since it is the rank of H, and therefore contains at most k vertices, each of which can be part of a distinct component.

The way that our algorithm works is as follows; let r_i be the first request in class C_j, then the singleton set $S = \{r_i\}$ is initialized, and the corresponding variable y_S is considered active, since

$1 \leq |S \cap \hat{R}| \leq n-1$, and initialized to 0. The algorithm will implement a subroutine in order to increment the variables associated with a given active component one-by-one. This process is conceptualized as placing a *ball* centered at each request, of a given radius. The radius will be determined by the cost of the request incurred by $Greedy$. The subroutine is given the hypergraph $H = (V, E, w)$, a terminal $r_i \in V$, the current dual solution vector $\vec{y}$, and the desired radius γ. A dual-LP-ball is then placed of a radius γ by incrementing a chain of dual variables centered at r_i by a cumulative amount of at most γ. The first such variable in the chain is the one associated with the singleton $S = \{r_i\}$, the initial active component containing request r_i. We examine the edges containing r_i, which are the edges crossing the cut S, and increase dual variable $y_{\{r_i\}}$ by the minimal reduced weight edge, e, from among them. Then, the component S *absorbs* the vertices contained in e, by which we mean that $S \leftarrow S \cup e$, and the component $\{r_i\}$ is considered *deactivated* while the new component S is considered *activated*. The same process is repeated for the newly considered variable y_S; the edges in $\delta(S)$ are examined and the minimal reduced weight edge among them is selected, the value of y_S is incremented by that amount, S is updated, and so on until the cumulative increase in all y_S attains γ. If the final incremented variable requires an increase which would lead the cumulative change in dual variables beyond γ, then that dual variable is only incremented enough for the total change to reach γ and the active component S is not updated to include the vertices of the minimal reduced weight edge which was found in $\delta(S)$. Once the total change in dual variables centered at r_i reaches γ by this process, the next request can be considered.

Algorithm 11 Algorithm that produces a dual LP solution for a given class i of requests.

1: **procedure** $CreateDualSolution(H, R, i)$
2: $\quad \vec{y} \leftarrow \vec{0}$
3: $\quad$ **for** $r \in \hat{R}$ **do**
4: $\quad\quad$ **if** $Greedy(r) \in [2^{i-1}, 2^i)$ **then**
5: $\quad\quad\quad (t, \vec{\Delta y}) \leftarrow ComputeDualLPBall(H, r, \vec{y}, \frac{2^{i-1}}{k})$
6: $\quad\quad\quad$ **if** $t = \frac{2^{i-1}}{k}$ **then**
7: $\quad\quad\quad\quad \vec{y} \leftarrow \vec{y} + \vec{\Delta y}$ ▷ place the dual-LP-ball Δy
8: $\quad$ **return** $\vec{y}$
9: **procedure** $ComputeDualLPBall(H, v, \vec{y}, \gamma)$
10: $\quad S \leftarrow \{v\}$
11: $\quad \vec{\Delta y} \leftarrow \vec{0}$
12: $\quad t \leftarrow 0$
13: $\quad$ **while** $t < \gamma$ **do**
14: $\quad\quad \Delta w \leftarrow \min_{e \in \delta(S)}\{w(e) - \sum_{S': e \in \delta(S)'}(y_{S'} + \Delta y_{S'})\}$
15: $\quad\quad e \leftarrow$ edge in $\delta(S)$ minimizing $w(e) - \sum_{S': e \in \delta(S)'}(y_{S'} + \Delta y_{S'})$
16: $\quad\quad$ **if** $\Delta w = 0$ and $\sum_{S': e \in \delta(S)'} y_{S'} \neq 0$ **then**
17: $\quad\quad\quad$ **return** $(t, \vec{\Delta y})$
18: $\quad\quad \Delta \leftarrow \min\{\Delta w, \gamma - t\}$
19: $\quad\quad \vec{\Delta y} \leftarrow \Delta y_S + \Delta$
20: $\quad\quad S \leftarrow S \cup e$
21: $\quad\quad t \leftarrow t + \Delta$
22: $\quad$ **return** $(t, \vec{\Delta y})$

Lemma 4.3.1. *Let $r_1, r_2, \ldots, r_m$ be a subsequence of requests in a given cost class C_j. If r_m is the first request such that $t < \frac{2^{j-1}}{k}$ is the first output of $ComputeDualLPBall$ for line 5 of Algorithm 11, then there is a subgraph of saturated edges connecting r_m with up to $k-1$ of the previous requests in the subsequence, with a total weight less than 2^{j-1}.*

Proof. Let r_i be the earliest request to co-saturate the edge in question alongside r_m. Then let $\vec{y}$ be the solution obtained by Algorithm 11 before r_m was requested. Then

$$\vec{y} = \sum_{j=1}^{m-1} \vec{\Delta y}_j$$

and this assignment is obtained by placing non-overlapping dual LP balls centered at each $r_{i'}$, $1 \leq i' \leq m-1$, each of radius $\frac{2^{j-1}}{k}$. So $\exists S$ such that $\Delta w = 0$ and $\sum_{e \in \delta(S)} y_S > 0$. There can be at most k balls in $\vec{y}$ for which any edge e is in the boundary, so let $\Delta y_j = \Delta y_{j_1}, \ldots, \Delta y_{j_{k'}} = \Delta y_m$

for $k' \leq k$ be the corresponding dual variables. The weight of edges connecting r_m to r_{i_1} and/or r_{i_2}, and so on up to $r_{i_{k'}}$ is at most

$$\vec{1}^T \cdot (\Delta\vec{y}_m + \sum_{i=1}^{k'-1} \Delta\vec{y}_{j_i}) = \vec{1}^T \cdot \Delta\vec{y}_m + \sum_{i=1}^{k'-1} \vec{1}^T \cdot \Delta\vec{y}_{j_i}$$

$$< \frac{2^{j-1}}{k} + (k-1)\frac{2^{j-1}}{k} < 2^{j-1}.$$

and the inequality is strict since the radius of the dual LP ball about r_m is strictly less than $\frac{2^{j-1}}{k}$. □

The following lemma tells us that the subroutine $ComputeDualLPBall$ of Algorithm 11 always terminates successfully, which is to say it manages to pack the dual variables associated with each request in a given class to the full amount γ while maintaining feasibility of the dual solution for that class.

Lemma 4.3.2. *The if statement on line 6 of Algorithm 11 on an instance of OSCH is always true.*

Proof. Consider for the sake of contradiction that the value of $t < \frac{2^{j-1}}{k}$ for some request r. This means that the while loop on line 9 was terminated early by the if condition on line 16. This condition implies that the edge with minimal reduced weight was previously reassigned to 0, and therefore that it was saturated by at most $k-1$ previously requested terminals of the same class. Then by Lemma 4.3.1, there is a subgraph of saturated edges connecting each of these requests, whose combined weight is less than $k \cdot \frac{2^{j-1}}{k} = 2^{j-1}$. This contradicts the $Greedy$ choice of the algorithm, since r was supposed to be in class j, and therefore that the cost of $Greedy$ to satisfy request r was at least 2^{j-1}, $c(r) \in [2^{j-1}, 2^j)$. □

Lemma 4.3.2 is now applied in Lemma 4.3.3 to bound the total cost of each class by its corresponding dual solution. This brings us one step closer to bounding the total cost incurred by $Greedy$.

Lemma 4.3.3. *If C_j denotes requests in class j, then dual assignment $\vec{y}_j$ returned by Algorithm 11 is feasible and achieves objective value at least $|C_j| \cdot \frac{2^{j-1}}{k}$*

Proof. As we saw in Lemma 4.3.2, the assignment to each dual LP ball centered at requests of class C_j remains feasible until at least $\frac{2^{j-1}}{k}$. The while loop condition on line 6 guarantees that this amount is not exceeded. Therefore we have that the value of the dual solution is

$$\sum_{i=1}^{|C_j|} \frac{2^{j-1}}{k} = |C_j| \cdot \frac{2^{j-1}}{k}$$

□

Now we relate the bound on the cost classes by the dual solutions to the cost of $Greedy$.

Lemma 4.3.4. *The cost incurred by $Greedy$ to satisfy the requests in class C_j is at most $2k(\vec{1}^T \cdot \vec{y}_j)$, where $\vec{y}$ is the dual assignment returned by Algorithm 11 for the corresponding dual class D_j.*

Proof. Each request in class C_j cost $Greedy$ strictly less than 2^j, by construction of the cost classes. Therefore, we have that

$$Greedy(C_j) < |C_j| \cdot 2^j.$$

Now by simple algebraic manipulation we get

$$Greedy(C_j) < |C_j| \cdot 2^j \cdot \frac{2}{k} \cdot \frac{k}{2} = 2k \cdot |C_j| \frac{2^{j-1}}{k}$$

where by Lemma 4.3.3 we have that $|C_j| \frac{2^{j-1}}{k}$ is the amount assigned to the dual class D_j, therefore

$$Greedy(C) < 2k(\vec{1}^T \cdot \vec{y}_j).$$

□

The lemma gives us the last piece required to provide the alternative proof for the desired result. Here we sum over all cost classes to obtain the bound on the total cost of $Greedy$.

Theorem (4.2.2, restated)**.** *Let $H = (V, E)$ be a hypergraph of rank k and R a request sequence of*

length n for OSTH, then:

$$Greedy(R) \leq 2k \log n \cdot OPT(R)$$

i.e., $\rho(Greedy) \leq k \log n$.

Proof. Let $Greedy(R)$ denote the value of the solution obtained by $Greedy$, and let T be the edge set of the corresponding solution. If r_m was the most costly request for $Greedy$ to connect, then consider the normalized solution obtained by multiplying each edge in $E(H)$ by $\frac{n}{c(r_m)}$. The normalized cost of r_m becomes $c_n(r_m) = n$. Now we proceed by partitioning the terminals of R into cost classes $C_j,\ 0 \leq j \leq \log n$, where request r_i is in C_j if $c_n(r_i) \in [2^{j-1}, 2^j)$. Then

$$Greedy(R) = \sum_{j=1}^{\log n} Greedy(C_j).$$

For each cost class C_j, we construct a corresponding dual solution D_j by implementing Algorithm 11. Then by Lemma 4.3.4 we get that

$$\sum_{j=1}^{\log n} Greedy(C_j) \leq \sum_{i=j}^{\log n} 2k(\vec{1}^T \cdot \vec{y}_j)$$

where $\vec{y}_j$ is the dual variable assignment vector for dual solution D_j. Since we know from Theorem 2.8.1 that the relaxed dual LP solution $\vec{1}^T \cdot \vec{y}_j$ is bounded above by the primal IP (and therefore by a given offline optimal solution), we can bound the objective value for each dual solution by that of $OPT(R)$ to get

$$\sum_{j=1}^{\log n} 2k(\vec{1}^T \cdot \vec{y}_j) \leq 2k \sum_{j=1}^{\log n} OPT(R) = 2k \log n \cdot OPT(R).$$

□

4.4 Lower Bounds for OSCH

As with the upper bounds, we begin with a quick and easy first result for the lower bound before moving on to a tight lower bound for the OSCH. To begin, consider the complete k-uniform hypergraph on N nodes, denoted $K_{N,k}$, with $N \geq k + (k-1)^2$. The edges are all subsets of size k among the N nodes of the graph, so there are $\binom{N}{k}$ edges in total. The online adversarial input is then a sequence of nodes such that every time a given algorithm selects an edge to connect the current request (and therefore selects some subset of k vertices), the next request is a vertex from some other subset of k vertices. The online input is of k vertices, and the competitive ratio when all hyperedge weights are 1 is therefore at least k. Following this, another adaptation from [39], where the *diamond graph* used in that paper is multiplied and organized in layers over itself, so that the hyperedges extend to connect "similar" vertices across the layers.

We begin by considering an online deterministic algorithm, as defined in Section 2.1.

Theorem 4.4.1. *Let ALG be an online deterministic algorithm for OSCH, then*

$$\rho(ALG) \geq k$$

for hypergraphs of rank k. In other words, there is an input instance $\mathcal{I} = \{H = (V, E, w), R\}$ of OSCH such that $\mathrm{rank}(H) = k$ *and $ALG(R) = k \cdot OPT(R)$.*

Proof. Let ALG be an online deterministic algorithm for OSCH. Consider the complete k-uniform hypergraph $K_{N,k}$ on $N \geq k + (k-1)^2$ nodes , where the edge set is all vertex subsets of size k, and $\forall e \in E,\ w(e) = 1$. After the first connection request is made, a deterministic algorithm must choose at least 1 of the edges to connect it to the root. No matter which edge is selected, there will be $k-2$ Steiner vertices covered by this choice, which leaves $N - k \geq (k-1)^2$ vertices uncovered, from which the next terminal is requested. Since each edge can cover at most $k-1$ Steiner vertices, the request sequence can last $k-2$ more rounds without requesting a node which was previously covered, for a total of k requests. The number of edges selected by the deterministic algorithm is then $ALG(R) = k$, one for each request, while all k requests lie in just one of the

edges, $OPT(R) = 1$. Thus, the competitive ratio is

$$\rho(ALG) \geq \frac{ALG(R)}{OPT(R)} = k.$$

□

Now to achieve a tighter bound we use a layered diamond graphs construction. The basic structure of the modified input is to take the lower bound input of [39] and build it up into a k-uniform hypergraph. We begin by providing a definition for these original diamond graphs, G_ℓ, where the ℓ is called the *level* of the graph. G_0 is simply two nodes (called the level 0 nodes), $v_{0,0}, v_{0,1}$ connected by an edge of cost 1. Then G_ℓ is obtained by taking each edge in $G_{\ell-1}$, doubling it, and adding a new level ℓ vertex between the two vertices along each new edge. In other words, take each edge (u, v) in $E(G_{\ell-1})$, connecting nodes $u, v \in V(G_{\ell-1})$, and replace it with new edges $(u, w_1), (w_1, v)$ and $(u, w_2), (w_2, v)$ for w_1, w_2 two new level ℓ nodes. The unit cost for any path from $v_{0,0}$ to $v_{0,1}$ is maintained, so the cost for each edge in G_ℓ is half that of $G_{\ell-1}$. Therefore the cost of each edge in G_ℓ is $2^{-\ell}$.

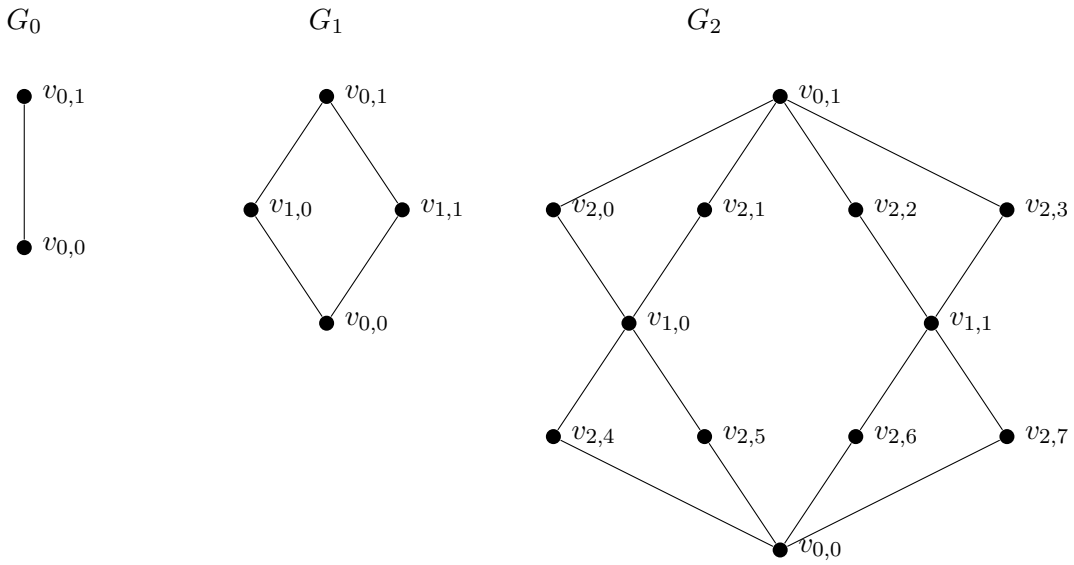

Figure 4.3: Diamond graphs, first described in [39].

Note the following observations about this family of graphs: there is 1 edge at level 0, 4 at level

1, then 16 at level 2. Each edge in $G_{\ell-1}$ is split into 4 as the graph gains a level to G_ℓ, so the number of edges $|E(G_\ell)| = 4^\ell$. Now for the number of nodes, there are 2 level 0 nodes, 2 level 1 nodes, and 8 level 2 nodes. For each edge in $G_{\ell-1}$, there will be 2 nodes of level ℓ. This means there are $2 \cdot 4^{\ell-1}$ level ℓ nodes for $\ell > 0$. Then the total number of nodes in G_ℓ is

$$2 + \sum_{i=1}^{\ell} 2 \cdot 4^{i-1} = 2\left(1 + \sum_{i=1}^{\ell} 4^{i-1}\right)$$

Applying the formula for the geometric series we get that G_ℓ has a total of $2\left(1 + \frac{4^\ell - 1}{3}\right)$ nodes. Each path from $v_{0,0}$ to $v_{0,1}$ in G_ℓ must pass through 2^{i-1} level i nodes for $i \geq 1$, since it must pass through one of the two level 1 nodes, passing through two of the four level 2 nodes to do so, and so on. The following lemma was among the first major results in the study of the online Steiner tree problem in graphs, and it will prove quite handy for the current analysis in hypergraphs.

Lemma 4.4.2. *[39] For every ℓ, and any deterministic algorithm ALG there is an adversarial request sequence R, with $|R| = n$, such that $ALG(R) = O(\log n)$ and $OPT(R) = 1$. Moreover, OPT is a single path from $v_{0,0}$ to $v_{0,1}$.*

Proof sketch. The idea is for the adversary to request nodes level by level adapting to decisions of ALG, while ensuring that at the end all the requested nodes would lie on the same path from $v_{0,0}$ to $v_{0,1}$. This means that OPT would be 1. At each level the adversary selects nodes of that level not yet covered by ALG (assuming that ALG does not buy redundant edges[1]) and consistent with previous requests (meaning lying on the same $v_{0,0}$ to $v_{0,1}$ path), forcing the algorithm to buy more paths. Another way to think of this argument is as follows: initially there are many $v_{0,0}$ to $v_{0,1}$ paths, and the adversary wants to delay ALG from guessing the final path for as long as possible.

Initially the adversary requests $v_{0,0}$ and $v_{0,1}$, after which any deterministic algorithm must

[1] We say that a set of edges bought by ALG is non-redundant if there is no proper subset of the edges bought by ALG that would still result in valid processing of the input request. Note that any algorithm that buys redundant edges can be converted into one that does not buy redundant edges by delaying the purchase of redundant edges until they become non-redundant. Thus, we may assume without loss of generality that a deterministic algorithm only buys non-redundant edges.

choose a path through one of the level 1 nodes to connect them, say, $v_{1,0}$. In this case, the adversary specifies the next request to be the other level 1 node, specifically, $v_{1,1}$. Observe that there are 8 level 2 vertices, but only 4 of them are consistent with $v_{1,1}$ (that is they lie on $v_{0,0}$ to $v_{0,1}$ paths passing through $v_{1,1}$). Once the algorithm chooses a set of edges passing through one of the four level 2 nodes between $v_{0,0}$ or $v_{0,1}$ and the last request, the next request is revealed to be two of the three level 2 nodes which were not selected by the algorithm in the previous round. This pattern continues, requesting twice the number of level i nodes than what was requested previously for those of level $i-1$, all while keeping the entire request sequence along a single path. The cost incurred by any deterministic algorithm is then 1 to connect the level 0 nodes initially, then 2^{-i} to connect each level i node requested afterwards. Since there are 2^{i-1} level i nodes requested for each level $i > 0$, the cost incurred by the algorithm for the nodes in level $i > 0$ is $\frac{1}{2}$. Since there is a total of ℓ levels, the total cost is $1 + \frac{\ell}{2}$. Since there are 2^{i-1} requests for each level $i > 0$, we have that the total number of requests is $2 + \sum_{i=1}^{\ell} 2^{i-1} = 2^{\ell}$, so we can write the total cost in terms of the number of requests as $1 + \frac{\log(n-1)}{2}$. Since each request lies along a single path between the level 0 nodes, the cost of the offline optimum is 1, giving us the desired competitive ratio.

For a more formal description and proof, we refer the interested reader to [39]. □

Recall that we have been talking about ordinary graphs (or, 2-uniform hypergraphs) so far. Next we show how to achieve a similar bound for hypergraphs, amplifying the competitive ratio by a multiplicative factor of the rank of the hypergraph. The diamond graphs are used as a gadget to build the hypergraphic input and the adversarial sequence from Lemma 4.4.2 is used as a subsequence to complete the input instance.

Theorem 4.4.3. *No deterministic algorithm for OSCH can achieve a competitive ratio better than $O(k \log n)$ in hypergraphs of rank k, where n is the size of the request sequence.*

Proof. Let ALG be a deterministic algorithm for the OSCH. We construct our adversarial input instance on a k-uniform hypergraph for $k = 2d$, $d \geq 2$. By implementing the diamond graph construction of [39], we start with the graph G_ℓ and from this we want to construct a hypergraph

H_ℓ by adding layered copies of G_ℓ above itself, where each edge in the original will correspond to a family of hyperedges in the resulting hypergraph. We can imagine the original graph is embedded in the plane, then d layers are filled with copies vertically above it, embedded in parallel planes to that of the original. The first layer above the original graph will contain $4^\ell+1$ copies of the diamond graph G_ℓ. The next layer will contain $(4^\ell+1)^2$ copies of the diamond graph. The number of copies in the j^{th} layer will be $(4^\ell+1)^j$. The reason the number of copies should increase as described in each layer is that we want there to be a copy of the input graph for which no edges have been selected to satisfy the requests from lower layers of H_ℓ (unless an algorithm is redundant). This would allow us to derive a lower bound on the cost of ALG to satisfy the request sequence in each layer, and subsequently a lower bound on the total cost of ALG.

To describe the edges $E(H_\ell)$, we begin by defining a *region* in the diamond graph. These regions are sets of edges, where region 0 is those edges incident on $v_{0,0}$, and similarly in the rest of the graph the region of a given edge is defined by the path length (non-weighted distance) of the incident vertex nearest to $v_{0,0}$. Then the hyperedges will contain the vertices connected in the original diamond graph, as well as a pair connected by an edge of the same region in one of the copies from each layer of this construction. Each hyperedge is like a ladder from some region of the original diamond graph, along a particular edge of the same region of the diamond graphs of each layer all the way to the final layer. Each such ladder-edge is added to this construction, some of which are illustrated in Figure 4.4 and Figure 4.5 below. The cost of each edge is the same, so that a direct path from $v_{0,0}$ to $v_{0,1}$ remains 1 the weights are set to $2^{-\ell}$.

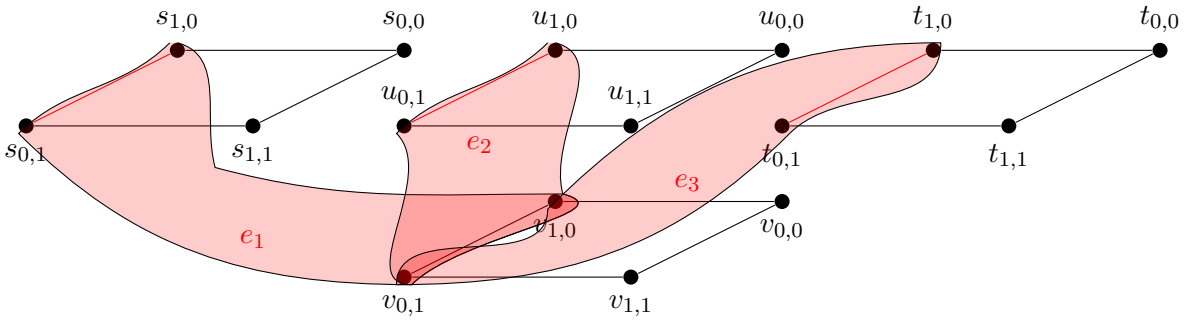

Figure 4.4: A part of H_1 with hyperedges e_1, e_2, e_3 highlighted.

The edges highlighted edges in Figure 4.4 and Figure 4.5 are all region 1 edges, since they contain at least one copy of either $v_{1,0}$ or $v_{1,1}$, which are at distance 1 from a $v_{0,0}$ node.

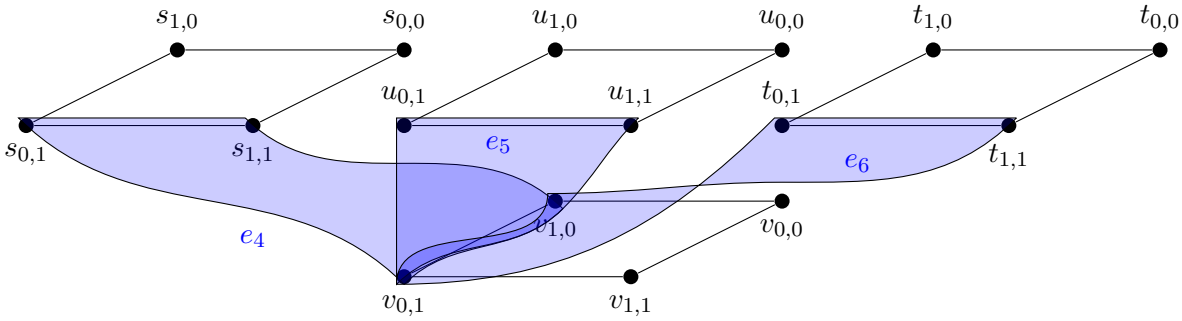

Figure 4.5: A part of H_1 with hyperedges e_4, e_5, e_6 highlighted.

The black edges in the Figure 4.4 and Figure 4.5 are edges of G_ℓ, they do not appear in H_ℓ but the vertices they connect are also connected by a hyperedge in H_ℓ. Some of these hyperedges are highlighted in red in the first figure and blue in the second, the two figures are separate to avoid clutter.

The adversary shall play the strategy of Lemma 4.4.2 d times, once for each layer, starting with layer 0 and progressing to higher layers, one-by-one. We refer to the number of requests in a single copy as n_0. Thus, there shall be dn_0 requests in total. Once the adversary moves to the next layer, it selects a copy of G_ℓ untouched by any previously selected hyperedges by ALG (assuming that ALG does not buy redundant edges) and restarts the strategy of Lemma 4.4.2 on that copy of G_ℓ. An untouched copy is guaranteed to exist in each layer by pigeonhole principle, since we introduce an excessive number of copies in each layer and we assume non-redundancy of ALG.

More specifically, whenever ALG buys a hyperedge to process a request in a copy of the original graph in a particular layer, that hyperedge contains an edge from the same region of some copy of the original graph in each layer. This way, once the n_0 requests are exhausted in the original graph in layer 0, at most 4^ℓ hyperedges must have been selected to process these requests (since this is the total number of ordinary edges in a copy of G_ℓ and non-redundant algorithm would never select more hyperedges than there are edges in a copy of G_ℓ to cover the requests in G_ℓ) and one of the

copies in the next layer remains untouched by ALG, so a similar request sequence can be continued on this untouched copy. Since the solution of ALG constructed so far does not have any hyperedges in common with the untouched copy, it must pay the cost of Lemma 4.4.2 again, while the cost of an optimal solution remains 1. The degree of each hyperedge is twice the number layers diamond graphs in the new construction $k = 2d$, since each hyperedge contains one ordinary edge per layer. Since the deterministic algorithm has cost $1 + \frac{\log n_0}{2}$ for each layer and the cost of OPT remains 1, the competitive ratio is

$$\frac{d}{2}\Big(1 + \frac{\log n_0}{2}\Big) = \frac{k}{4}\Big(1 + \frac{\log n_0}{2}\Big) = \frac{k}{4}\Big(1 + \frac{\log \frac{n}{k}}{2}\Big) = \frac{k}{4}\Big(1 + \frac{\log n - \log k}{2}\Big) = O(k \log n)$$

□

Note that the same lower bound with k odd is obtainable by simply adding a dummy vertex to the construction above and including it in every hyperedge.

Chapter 5

Generalized Online Steiner Cover on Hypergraphs

In this chapter, we formally define the generalized online Steiner cover problem on hypergraphs and present a generalized greedy algorithm (Section 5.1), which we continue to denote by *Greedy*. Then we prove an upper bound of $O(k \log^2 n)$ on the competitive ratio (Section 5.2). A tighter bound is determined for a new algorithm which we call the *Generalized Greedy Steiner Cover* algorithm (*GGSC* for short), using LP duality to bound the performance of *GGSC* (Section 5.3).

5.1 The Generalized Online Steiner Cover on Hypergraphs

In the generalized online Steiner cover problem on hypergraphs (GOSCH), we have as input an edge-weighted hypergraph $H = (V, E, w)$ where $|V| = N$, $E \subseteq 2^V$ offline, exactly as in the Steiner tree problem, and a subset of unordered pairs of vertices $R \subseteq V \times V$, $|R| = n \leq \binom{N}{2}$ presented online. In other words, the difference between this problem and the one previously considered is that the requests now arrive in pairs and no longer need to be connected to a common root node. A solution needs to select a minimum-weight subset of edges which maintains connectivity of the

terminal pairs. The output of our algorithm after the i^{th} request should be a set of edges $F_i \subseteq E$ describing a sub-hypergraph of H which contains all previously requested terminals and which connects the terminal pairs requested thus far. For any given pair $p_i = (s_i, t_i), 1 \leq i \leq n$ there should be a path between s_i and t_i along some subset of F_i. After each request arrives, the algorithm must immediately select edges which connect the vertices to each other and add them to the existing solution, so $F_{i-1} \subseteq F_i \quad \forall i \geq 0$ and $F_0 = \emptyset$. Once the last request is processed by the algorithm, we have the final solution denoted by $F = F_n$. This is a generalization of the online Steiner cover problem discussed in the previous chapter since the latter can be simulated by this problem by making each request share a common terminal. This common terminal can be seen as the root node, or the first request of the OSCH. More specifically, if the online request sequence for OSCH is $R = (r_i)_{i=1}^n$ (note that in the OSCH problem r_i was used to denote the i^{th} request, whereas in this chapter we denote the i^{th} request pair as p_i) then the equivalent request sequence for GOSCH would be $R = (p_i = (r_{i+1}, r_1))_{i=1}^{n-1}$.

A solution will generally be composed of some collection of connected components $\mathcal{C}$, where each component $C \in \mathcal{C}$ connects a subset of the request sequence.

Problem Definition 12 Generalized Online Steiner Cover on Hypergraphs

Input

$H = (V, E, w)$ a weighted hypergraph $w : E \to \mathbb{R}^+$ offline.
$P = \{p_1, ..., p_n\}$ terminal pair sequence $p_i \in V \times V$ for $i \in [n]$ online.

Output

$F_i \subseteq E$ on input p_i, an edge set connecting the terminal pairs $p_j, 1 \leq j \leq i$.

Objective

$\min w(F) = \sum_{e \in F} w(e)$, i.e., minimize the total edge weight of the solution, where $F = \bigcup_{i=1}^n F_i$ a sub-hypergraph of H containing all requested terminals.

Example: Consider the 3-uniform hypergraph $H = (V, E, w)$ with $N = 7$ vertices and 3 edges of unit weight defined as follows:

$$V = \{s_1, s_2, u_1, u_2, u_3, t_1, t_2\}, \text{ and}$$

$$E = \Big\{e_1 = \{s_1, u_1, t_1\}, e_2 = \{u_1, u_2, u_3\}, e_3 = \{s_2, u_3, t_2\}\Big\}.$$

Suppose two requests arrive online $p_1 = (s_1, t_1)$ and $p_2 = (s_2, t_2)$, then the solution $F = \{e_1, e_3\}$ is feasible and has weight 2. Observe that when this request sequence is reinterpreted in the OSCH setting as discussed above, this solution is no longer feasible, and the only feasible solution is to take the entire E. See Figure 5.1.

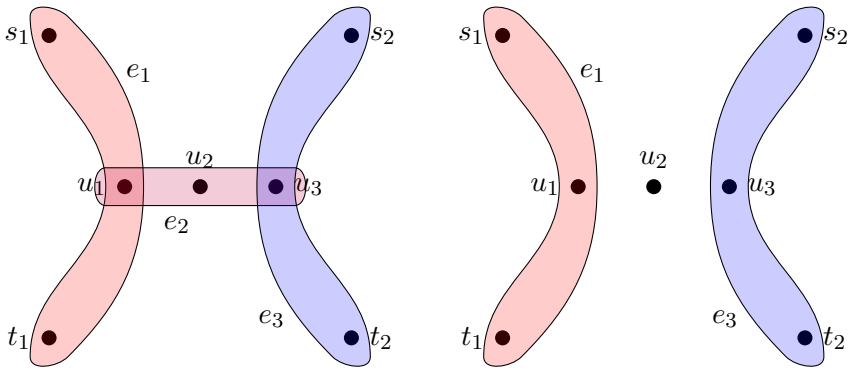

Figure 5.1: Sample input hypergraph with request sequence $(s_1, t_1), (s_2, t_2)$ (left), edges are of unit weight; and the solution with 2 connected components, connecting s_1 to t_1 and s_2 to t_2 at minimal edge weight (right).

5.2 Upper Bounds for GOSCH

Recall that in the setting of ordinary graphs, the analysis of *Greedy* for the online Steiner forest problem was first done by [6], who showed the upper bound on the competitive ratio of $O(\log^2 N)$. Here, we adapt this result to hypergraphs of rank k and show that *Greedy* has a competitive ratio of at most $O(k \log^2 N)$ for GOSCH. Our argument follows the structure of the argument in [6], generalizing some steps to the hypergraph setting. We begin by introducing some key definitions and then giving a high-level overview of the argument.

Let $H = (V, E, w)$ be a given edge-weighted hypergraph and fix $d \in \mathbb{R}_{\geq 0}$. Two nodes $u, v \in V$ are called *d-separated* if the distance between them in H is greater than d, i.e., $\text{dist}_H(u, v) > d$. A map $rep : V \to V$ is called a *d-net* if it satisfies the following two properties:

- for $u, v \in V$ if $rep(u) \neq rep(v)$ then $rep(u)$ and $rep(v)$ are d-separated, and

- for every $u \in V$, u and $rep(u)$ are not d-separated.

In other words, the pairwise distance between distinct representatives is at least d, and the pairwise distance between any vertex and its representative is less than d.

Intuitively, d-net partitions the set of vertices of H into blocks, such that each block has a representative vertex in V, all vertices in the block are within distance d from the representative, and different representatives are d-separated. We denote the set of representatives by $V_d = rep(V)$. For a representative $v \in V_d$, the block of nodes corresponding to this representative is $rep^{-1}(v)$. While H might have several different d-nets, for the purpose of the argument in this section we could fix an arbitrary one. For concreteness, we shall fix a d-net constructed by the following greedy process. Start by picking an arbitrary vertex v and consider the set S of all the nodes u within distance d of v, i.e., $\mathrm{dist}_H(u, v) \leq d$. Define $rep(u) = v$ for all $u \in S$. Then remove vertices S from H, and repeat.

Now, suppose that in addition to H we have a sequence of requests $R = (p_i)_{i=1}^n$. Define an *auxiliary graph* $G_{l,d} = (V_d, E_{l,d})$, where V_d is the set of representatives of the d-net rep constructed as described above, and

$$E_{l,d} = \Big\{ \{rep(s_i), rep(t_i)\} : \text{the cost of } \textit{Greedy} \text{ on } p_i = (s_i, t_i) \text{ is at least } l \Big\}.$$

Observe that the auxiliary graph is an ordinary graph, but may not be simple, i.e., for some choices of l and d, the auxiliary graph might have self-loops and parallel edges (in which case the sets in the definition of auxiliary graph are replaced by multisets, where elements may not be unique). The argument below will make sure that l and d are chosen so that the auxiliary graph is an ordinary simple graph, thus known graph theory results of simple graphs can be applied to it.

Next, we give a high-level overview of the proof of the main theorem of this section, Theorem 5.2.7, which states that the competitive ratio of *Greedy* is $O(k \log^2 N)$ with respect to hypergraphs of rank k. Fix an arbitrary offline optimal solution OPT and consider a particular connected

component C of this solution. Let $P(C)$ denote the request pairs whose endpoints are within component C. It is sufficient to show that the cost of $Greedy$ in processing $P(C)$ is within the factor $O(k \cdot \log N \cdot \log |P(C)|)$ of the total weight of that component, denoted by $w(C)$. The bound would then follow for the entire input instance by summing over all connected components of OPT. Suppose that $z = |P(C)|$ and let $l_1 \geq l_2 \geq \cdots \geq l_z$ be the costs of $Greedy$ in processing $P(C)$ for each request sorted from largest to smallest. The crux of the argument is in showing that the i^{th} largest cost of processing a request by $Greedy$ is not too large compared to the weight of the component, namely, $l_i = O\left(\frac{k \cdot w(C) \cdot \log N}{i}\right)$. Observe that if we proved this then the overall bound would follow using the bound on the Harmonic numbers [58]:

$$\sum_{i=1}^{|P(C)|} l_i = \sum_{i=1}^{|P(C)|} O\left(\frac{k \cdot w(C) \cdot \log N}{i}\right) = O(k \cdot w(C) \cdot \log N \cdot \log |P(C)|),$$

where the n^{th} Harmonic number has the following bound

$$\sum_{k=1}^{n} \frac{1}{k} = H_n \leq \ln n + 1.$$

By simple algebraic manipulation observe that showing $l_i = O\left(\frac{k \cdot w(C) \cdot \log N}{i}\right)$ is equivalent to showing $i = O\left(\frac{k \cdot w(C) \cdot \log N}{l_i}\right)$. In fact, an even stronger claim is shown: let $P_l(C)$ denote the subset of requests from $P(C)$ such that the cost of $Greedy$ on a request is at least l. We want to show $|P_l(C)| = O(\frac{k \cdot w(C) \cdot \log N}{l})$. In order to derive a bound on $|P_l(C)|$ we shall make use of the auxiliary graph $G_{l,d} = (V_d, E_{l,d})$. Notice that for every value of d we have $|P_l(C)| = |E_{l,d}|$ if we allow parallel edges and self-loops in the construction of $G_{l,d}$, as discussed above. If we view the parameter d as being set on a slider to adjust its value, then increasing or decreasing it has the following effect on the auxiliary graph: if we increase d then this decreases $|V_d|$ but increases the number of parallel edges and self-loops in $G_{l,d}$, and if we decrease d then this increases $|V_d|$ but decreases the number of parallel edges and self-loops. The trick is to find a value of d such that (1) $G_{l,d}$ is a simple graph, (2) $|E_{l,d}| = O(|V_d|)$, and (3) $|V_d| = O\left(\frac{k \cdot w(C) \cdot \log N}{l}\right)$. Setting $d = \frac{l}{8 \log N}$ achieves all these goals. More specifically, for this choice of d, the girth of $G_{l,d}$ is

at least $4 \log N \geq 4$ (assuming $N \geq 2$). This immediately establishes (1) that $G_{l,d}$ is simple, and it establishes (2) that $|E_{l,d}| = O(|V_d|)$ via a girth-based bound on the number of edges (see Lemma 5.2.2 and Lemma 5.2.3). To establish (3), we use the observation that for hypergraphs of rank k whenever $d \leq w(C)$ we have $|V_d| \leq \frac{k}{d} w(C)$. This is derived from our setting V_d given by the greedy ball packing procedure, where we place balls of radius d repeatedly until there are no more vertices left to cover, we can implement a similar procedure to prove the existence of a packing of disjoint balls to achieve the desired bound. This is done in Lemma 5.2.5 and Lemma 5.2.6. This happens to be where the extra factor of k appears as compared to the ordinary Steiner forest problem: when packing disjoint $\frac{d}{k}$-balls in a hypergraph, at most k balls can cover any single edge due to the fact that at most k vertices are contained within it.

In the following lemma, Lemma 5.2.1, we give a proof for the existence of a subgraph with a large minimum degree in ordinary graphs. This result, together with the girth bound which will follow, will allow us to count edges to achieve a lower bound on $|E_{l,d}|$. The idea here is to remove vertices of small degree until the minimum degree of the resulting subgraph becomes sufficiently large, and to prove that this process terminates before removing every vertex in the graph (so the subgraph obtained is non-empty).

Lemma 5.2.1. *[70] Every graph $G = (V, E)$ has a non-empty subgraph with minimum degree $\delta \geq \frac{|E|}{|V|}$.*

Proof. Given a graph $G = (V, E)$, we want to show that G has a non-empty subgraph $G' = (V', E')$ with minimum degree $\delta(G') \geq \frac{|E|}{|V|}$. To do this, we simply proceed by iteratively removing vertices of degree strictly less than $\frac{|E|}{|V|}$. Obviously, if the process terminates and the result is non-empty, the lemma is proved. We must therefore show that the result of this process is non-empty. Assume for the sake of contradiction that it is. Then after the last round we removed exactly $|V|$ vertices of degree strictly less than $\frac{|E|}{|V|}$, and therefore we removed strictly less than $\frac{|E|}{|V|} \cdot |V| = |E|$ edges overall. This is a contradiction, since if we removed every vertex then we must have removed every edge as well. This implies that the resulting subgraph is nonempty. Since the process is guaranteed to terminate in less than $|V|$ steps, we can conclude that the resulting subgraph has minimum degree

at least $\frac{|E|}{|V|}$. $\square$

Now we apply Lemma 5.2.1 to prove that we can bound the number of edges in a graph by some function of the number of vertices and the girth of the graph. This will be used to bound the number of edges in the auxiliary graph (recall that the edges of $E_{l,d}$ correspond to request pairs whose cost to $Greedy$ exceeds l). As mentioned previously, the proof is done by counting the edges. We consider an expanding neighbourhood centered at some vertex, up to a diameter just shy of the girth of the graph. This ensures that no edge is double counted, since this would imply a cycle smaller than the graph's girth. Then, by taking into account the existence of a large degree subgraph from Lemma 5.2.1, we come to a suitable bound on the number of edges.

Lemma 5.2.2. *[15] If g is the girth of an unweighted graph $G = (V, E)$ then the following bound holds*

$$|E| \leq |V|^{1+\frac{O(1)}{g}}$$

Proof. Let $v \in V$ and consider placing a ball $G(v, r)$ with $r = \lfloor \frac{g-1}{2} \rfloor$ in graph G. Then the covered subgraph $G(v, r)$ must be a tree, since a cycle in this ball would imply a cycle of length less than g in G and all vertices are reachable from v. Now consider the expanding component obtained by increasing the radius of the ball from 0 to $\lfloor \frac{g-1}{2} \rfloor$; in each step the last vertices added to the neighborhood reveal at least δ new neighbors in $G(v, r)$, where δ denotes the minimum degree of G. Once the ball reaches its maximum radius, there must be at least $\delta^{\lfloor \frac{g-1}{2} \rfloor}$ distinct vertices in $V(B)$. Therefore,

$$\delta^{\lfloor \frac{g-1}{2} \rfloor} \leq |V|$$

$$\delta \leq |V|^{\frac{1}{\lfloor \frac{g-1}{2} \rfloor}}$$

Using Lemma 5.2.1, we know that G contains a subgraph with minimum degree $\frac{|E|}{|V|}$, and since the girth of any subgraph of G is at least as large as the girth of G,

$$\frac{|E|}{|V|} \leq |V|^{\frac{1}{\lfloor \frac{g-1}{2} \rfloor}}$$

$$|E| \leq |V|^{1+\frac{1}{\lfloor \frac{g-1}{2} \rfloor}} = |V|^{1+\frac{O(1)}{g}}$$

□

Proceeding with the plan highlighted previously, we provide the proof of a lower bound on the girth of the auxiliary graph. The proof makes use of the behaviour of the $Greedy$ algorithm: by contradiction it is shown that a small cycle cannot exist in the auxiliary graph, since this would imply the existence of a shorter path which $Greedy$ could have selected to connect the last terminal pair that was requested along this cycle. This can be used to reveal the edge bound on the auxiliary graph, thanks to the result from Lemma 5.2.2.

Lemma 5.2.3. *[6] The girth of the auxiliary graph $g(G_{l,d})$ is at least $\frac{l}{2d}$.*

Proof. Assume for the sake of contradiction that there is in fact a cycle of length $s < \frac{l}{2d}$, and consider the request sequence on some instance of the GOSCH for those requests which correspond to edges in the auxiliary graph $G_{l,d}$. The cost of the last of these pairs to arrive (which completes the cycle) is greater than l, but since it was the last to arrive we can alternatively connect it by the existing path highlighted by $G_{l,d}$ (see Figure 5.2), by routing each of the requested pairs to its representative vertex in V_d and traveling along a path consisting of edges previously selected by *Greedy*, for a cost of at most $2ds < l$, contradicting that *Greedy* selects the minimum cost path available to connect each request pair. □

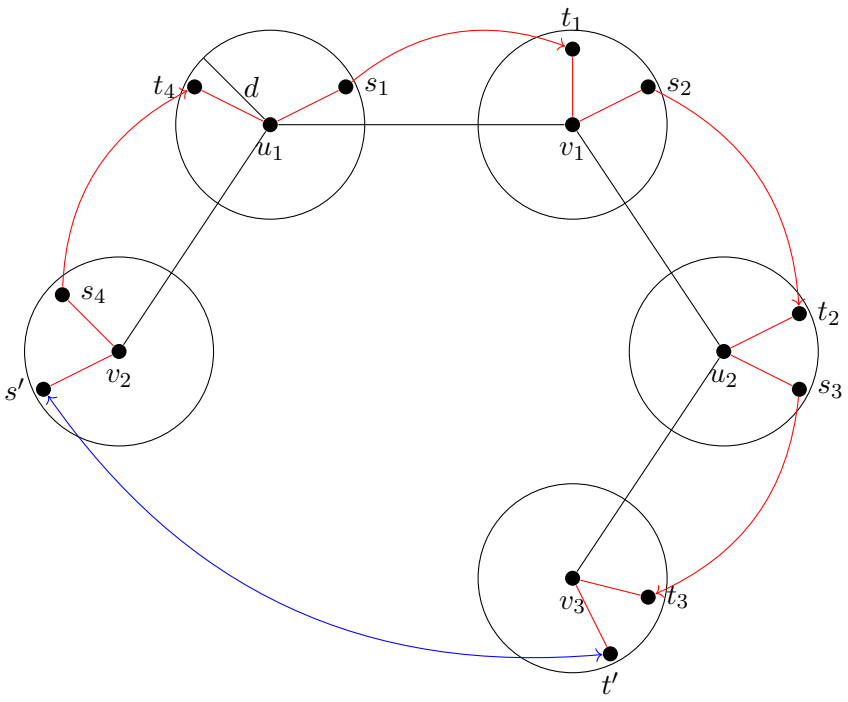

Figure 5.2: An example cycle of length $s = 5$ in the d-net, where the last pair to arrive was $p' = (s', t')$.

The blue path in Figure 5.2 corresponds to the path bought by *Greedy* to satisfy the last request pair p' to complete the cycle, of cost exceeding l, and the red path in the same figure shows a shortcut which was available to *Greedy* using previously selected edges. The total cost using the pre-selected edges amounts to at most $2ds$ where s is the length of the cycle, whereas *Greedy* is supposed to have paid more than $l > 2ds$. As previously stated, Lemmas 5.2.2 and 5.2.3 will allow us to bound the number of edges in the auxiliary graph.

Corollary 5.2.4. *[6] For $l \geq 8d \log N$ we have the following bound*

$$|E_{l,d}| = O(|V_d|)$$

Proof. From Lemma 5.2.2 we have that

$$|E_{l,d}| \leq |V_d|^{1+\frac{O(1)}{g(G_{l,d})}}$$

where $g(G_{l,d})$ is the girth of the auxiliary graph. Then, since in Lemma 5.2.3 $g(G_{l,d})$ was bounded below by $\frac{l}{2d}$ and it is given that $l \geq 8d \log N$, we have that $\frac{l}{2d} \geq \frac{8d \log N}{2d} = 4 \log N$

$$|V_d|^{1+\frac{O(1)}{g(G_{l,d})}} \leq |V_d|^{1+\frac{O(1)}{4 \log N}}$$

and since $|V_d| \leq N$ we have that $\log |V_d| \leq \log N$ and $\frac{1}{\log |V_d|} \geq \frac{1}{\log N}$, so

$$|V_d|^{1+\frac{O(1)}{4 \log N}} \leq |V_d| \cdot |V_d|^{\frac{O(1)}{4 \log |V_d|}} = |V_d| \cdot O(1) = O(|V_d|).$$

□

Now, since we have bounded the number of edges in the auxiliary graph by a constant factor of the number of vertices, we proceed with a bound on the number of vertices $|V_d|$. To do this, we will use the notion of a *ball* in a hypergraph. Fix a graph $H = (V, E, w)$, a vertex $v \in V$ and a real number $r > 0$. A *ball* centered at node v of radius r, denoted by $H(v, r) = (V(v, r), E(v, r))$, is a subgraph of H with the vertex set

$$V(v, r) = \{u \in V : \text{dist}_H(u, v) \leq r\}$$

and the edge set

$$E(v, r) = \{e \in E : \forall u \in e(\text{dist}_H(u, v) \leq r)\}.$$

In words, $H(v, r)$ is a subgraph of H induced on all the vertices within distance r of v. Observe that $H(v, r)$ is a connected subgraph. Then we say the nodes in $V(v, r)$ and edges in $E(v, r)$ are *covered* by the ball $H(v, r)$. A collection of balls $B = \{H(v_i, r_i)\}_{i=1}^m$ is called *disjoint* if $\forall j_1, j_2 \in [m]$ with $j_1 \neq j_2$ we have that

$$V(v_{j_1}, r_{j_1}) \cap V(v_{j_2}, r_{j_2}) = \emptyset$$

and otherwise they are called *overlapping*.

Example:

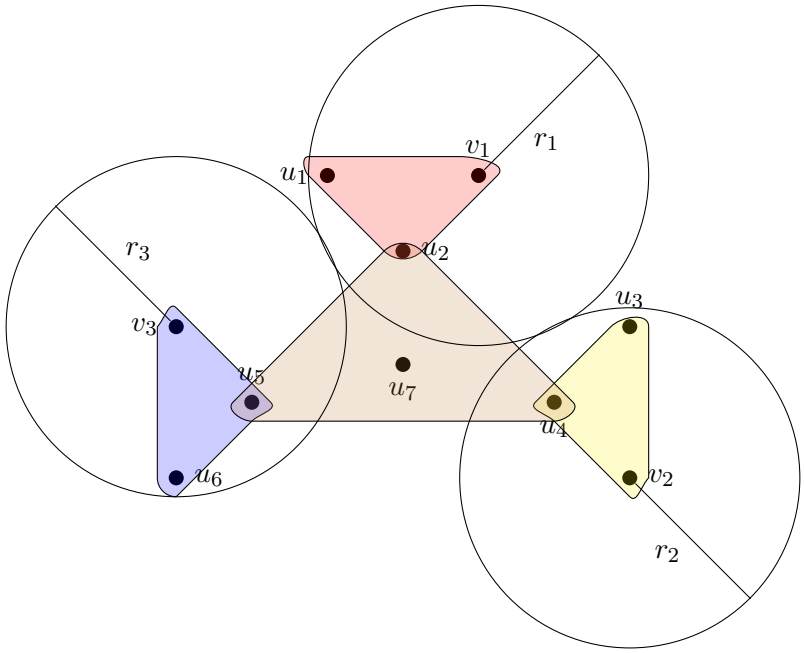

Figure 5.3: A hypergraph with balls centered at nodes v_1, v_2 and v_3.

In Figure 5.3 we have three disjoint balls, $B_1 = H(v_1, r_1), B_2 = H(v_2, r_2)$ and $B_3 = H(v_3, r_3)$ centered at nodes v_1, v_2 and v_3, respectively. B_1 containes nodes v_1, u_1, u_2, B_2 contains nodes v_2, u_3, u_4, and B_3 contains nodes v_3, u_6, u_8. The node u_7 is not contained by any of the three balls. The sum of radii of the balls is $r_1 + r_2 + r_3$ is at least as large as the sum of weights for the red, yellow, and blue hyperedges (we know this because the vertices contained in each of these edges are covered by one of the balls). However, the sum of radii is less than $e_{\text{red}} + e_{\text{yellow}} + e_{\text{blue}} + 3 \cdot e_{\text{brown}}$ since $r_1 < e_{\text{red}} + e_{\text{brown}}$, $r_2 < e_{\text{yellow}} + e_{\text{brown}}$, and $r_3 < e_{\text{blue}} + e_{\text{brown}}$. A more general statement of this fact is proven in the following lemma, Lemma 5.2.5.

Lemma 5.2.5. *If we can pack some connected weighted hypergraph $H = (V, E, w)$ of rank k with a collection $B = \{H(v_i, r_i)\}_{i=1}^{m}$, $m \geq 2$, of pairwise disjoint balls, then the following inequality holds,*

$$\sum_{i=1}^{m} r_i \leq k \cdot w(H)$$

Proof. We define the *boundary* of a ball $H(v_i, r_i)$, denoted $\delta(H(v_i, r_i))$, as the edges which contain

at least one vertex in $V(v_i, r_i)$ and at least one vertex not in $V(v_i, r_i)$, i.e.,

$$\delta(H(v_i, r_i)) = \{e \in E(H) : \exists u, v \in e, u \neq v \text{ such that } u \in V(v_i, r_i) \text{ and } v \in V(H) \setminus V(v_i, r_i)$$

Then consider the shortest path P_i from v_i to any node outside of $H(v_i, r_i)$. It is clear that P_i contains exactly one edge from $\delta(H(v_i, r_i))$, since otherwise we could delete one of them to get a shorter path which still leads to a node outside of $H(v_i, r_i)$. Note that this implies the following

$$w(P_i) \geq r_i.$$

Furthermore, we define the set of edges in P_i which intersect with $H(v_i, r_i)$ as follows,

$$\hat{P}_i = P_i \cap E(v_i, r_i)$$

and the edge of P_i which lies on the boundary of $H(v_i, r_i)$ is also defined,

$$\hat{e_i} = P_i \cap \delta(H(v_i, r_i)).$$

Note that this implies the following,

$$P_i = \hat{P}_i \cup \hat{e_i}.$$

Summing over the radii, we get that

$$\sum_{i=1}^{m} r_i \leq \sum_{i=1}^{m} w(P_i) = \sum_{i=1}^{m} w(\hat{P}_i) + \sum_{i=1}^{m} w(\hat{e_i})$$

Where the edges from each $\hat{P}_i$ are disjoint, but each edge among the $\hat{e_i}$ may appear a total of k times due to the rank of the hypergraph, and therefore

$$\sum_{i=1}^{m} w(\hat{P}_i) + \sum_{i=1}^{m} w(\hat{e_i}) \leq k \cdot w(H)$$

□

With this we can prove the following lemma regarding the number of representatives of a d-net. Notice that this number depends on the value of the parameter d relative to the weight of the hypergraph $H = (V, E, w)$ of rank k. This generalizes the case from [6] on 2-uniform graphs. The proof follows from Lemma 5.2.5, we need only specify the radii of the balls with which we want to pack the hypergraph. The idea of the proof is to bound the number of representatives from above by some factor of the weight of the hypergraph. This is achieved by placing a ball of maximal radius centered at each representative while avoiding any overlap. This covering by disjoint balls allows us to sum over the radii of the balls to achieve the desired bound.

Lemma 5.2.6. *Let $H = (V, E, w)$ be a hypergraph of rank k and fix $d > 0$. Let V_d be the set of representatives of an arbitrary d-net. Then:*

- *if $d \geq w(H)$ we have $|V_d| = 1$.*
- *if $d \leq w(H)$ we have $|V_d| \leq 2 \cdot \frac{w(H)}{d}$.*

Proof. When $d \geq w(H)$, the bound is quite trivial since the weight of any path between two nodes in H is at most the sum of edge weights of H, and therefore any node chosen for the first representative will remove all other nodes from consideration. Now consider $d < w(H)$, we want to show that the balls about nodes in V_d can be at most radius $\frac{d}{2}$ for any hypergraph. This is because we must avoid overlap along any hyperedge when d-separated. So, assume for the sake of contradiction that two balls of radius $\frac{d}{2}$ centered at nodes $u, v \in V_d$ do overlap. This implies that

$$\text{dist}_H(u, v) < 2\frac{d}{2} \leq d.$$

This contradicts that the nodes are both d-separated (being distinct representatives in the d-net). Therefore, by Lemma 5.2.5, the total weight of these balls $|V_d|\frac{d}{2}$ is bounded above by the weight of the component $w(H)$,

$$|V_d| \leq \frac{w(H)}{\frac{d}{2}} = 2 \cdot \frac{w(H)}{d}.$$

□

This change in the bound of a factor of k when extended to hypergraphs explains the change in the competitive ratio of $O(k \log^2 N)$ for N the number of vertices and k the rank of the hypergraph H. Lemma 5.2.1 and Lemma 5.2.2 alongside the previous one, Lemma 5.2.6, combine to prove the bound on the number of edges in the auxiliary graph from Lemma 5.2.4.

Now we can finally prove the main theorem of this section, by bounding the competitive ratio of *Greedy* for the GOSCH problem. This, recall, is done by bounding the performance of $Greedy$ on request sequence R for the terminal pairs within each connected component of the fixed offline optimal solution $OPT(R)$, then taking the sum over all connected components.

Theorem 5.2.7. *The* Greedy *algorithm for the GOSCH is $O(k \log^2 N)$-competitive, where N is the number of vertices in the input hypergraph.*

Proof. Let $c(p_i)$ be the cost incurred by *Greedy* to connect request p_i on some instance of GOSCH. Given some connected component of an optimal solution C for that instance, recall that $P(C)$ denotes the requests connected in that component (by OPT) and that $P_l(C)$ denotes the subset of $P(C)$ which cost *Greedy* at least l to connect,

$$P_l(C) = \{p_i = (s_i, t_i) \in R : s_i, t_i \in V(C) \text{ and } c(p_i) \geq l\}$$

Then we sort the request pairs by non-ascending cost, where l_i corresponds to the i^{th} cost in this sorting, as described previously and get the following:

$$\sum_{p_i \in P(C)} c(p_i) = \sum_{i=1}^{|P(C)|} l_i.$$

Now we consider the auxiliary graph $G_{l, \frac{l}{8 \log(N)}}$ for the component C. Then the number of edges $\left|E_{l, \frac{l}{8 \log N}}\right|$ is equivalent to $|P_l(C)|$ by construction. Consider also the sets $P_{l_i}(C)$, of pairs whose cost to $Greedy$ was at least as large as the i^{th} most expensive request in component C. Then applying Lemma 5.2.4 we have that

$$|P_{l_i}(C)| = \left|E_{l_i, \frac{l_i}{8 \log N}}\right| = O\left(V_{\frac{l_i}{8 \log N}}\right).$$

And following up with Lemma 5.2.6 we get

$$|P_{l_i}(C)| = O\left(\frac{k \cdot w(C)}{\frac{l_i}{8 \log N}}\right) = O\left(\frac{k \cdot w(C) \log N}{l_i}\right).$$

Allow a simple algebraic manipulation to isolate l_i,

$$l_i = O\left(\frac{k \cdot w(C) \log N}{|P_{l_i}(C)|}\right).$$

Note that by definition, $|P_{l_i}(C)| \geq i$ because it contains at least each request from the first to the i^{th} in the sorted ordering, and therefore

$$l_i = O\left(\frac{k \cdot w(C) \log N}{i}\right).$$

Now we can substitute for l_i in the summation,

$$\sum_{i=1}^{|P(C)|} l_i = \sum_{i=1}^{|P(C)|} O\left(\frac{k \cdot w(C) \log N}{i}\right)$$

Here we apply the bound on the Harmonic function [58]

$$= O\left(k \cdot w(C) \log N\right) \sum_{i=1}^{|P(C)|} \frac{1}{i} \leq O\left(k \cdot w(C) \log N\right) \log |P(C)|.$$

Here recall that the number of requests is bounded by $\binom{N}{2} \leq N^2$ to get

$$\begin{aligned} O\left(k \cdot w(C) \log N\right) \log |P(C)| &= O\left(k \cdot w(C) \log N \log N^2\right) \\ &= O\left(k \cdot w(C) \log^2 N\right). \end{aligned}$$

Finally, to get the desired bound on the overall performance of $Greedy$, we simply take the sum

over all connected components of OPT,

$$\begin{aligned} Greedy(R) = \sum_{C} \sum_{p_i \in P(C)} c(p_i) &= \sum_{C} \sum_{i=1}^{|P(C)|} l_i \\ &\leq \sum_{C} O\big(k \cdot w(C) \cdot \log^2 N\big) \\ &= O(k \cdot \log^2 N) \sum_{C \in \mathcal{C}} w(C) \\ &= O(k \cdot \log^2 N) \cdot OPT(R). \end{aligned}$$

□

5.3 LP Approach to GOSCH

When looking at the relaxed LP statement for GOSCH, the only difference with the OSCH LP is that the family of cuts $\mathcal{T}$ is no longer over cuts which separate *any* pair of requests, but specifically those which separate some request pair (s_i, t_i) for $1 \leq i \leq n$. [34]

$$\begin{aligned} \text{min.:} \quad & \sum_{e \in E} w(e) x_e \\ \text{s.t.:} \quad & \sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{T} \\ & 0 \leq x_e \leq 1 \quad \forall e \in E \end{aligned}$$

for $\mathcal{T} = \{S \subsetneq V : \exists (s_i, t_i) \in R \text{ where } |S \cap \{s_i, t_i\}| = 1\}$, and the corresponding dual,

$$\begin{aligned} \text{max.:} \quad & \sum_{S \in \mathcal{T}} y_S \\ \text{s.t.:} \quad & \sum_{S : e \in \delta(S)} y_S \leq w(e) \quad \forall e \in E \\ & y_S \geq 0 \quad \forall S \in \mathcal{T} \end{aligned}$$

So, as in the LP analysis from Section 4.3, we consider normalized edge weights to partition the primal solution C into $\log n$ cost classes C_j, $0 \leq j \leq \log n$. When setting the dual variables for the GOSCH, it is no longer possible to simply increment all dual variables y_S equally and maintain feasibility. This is because it no longer contradicts the behavior of the algorithm when two dual balls of a given cost class intersect, as not all requested vertices need be connected. The adjustment is the following: we continue making greedy decisions while keeping track of the dual variables, this time placing dual LP balls of radius $\frac{2^{j-2}}{k}$ for a request in class C_j, and when the dual LP balls for some set of dual variables corresponding to request $p_i = (s_i, t_i)$ "intersect" with another dual LP ball corresponding to $s_j \in p_j$ and/or $t_m \in p_m$, for some $1 \leq i, j, m \leq n$ with $j, m < i$, the algorithm will first buy the shortest paths to connect these non-request terminal pairs. The intersection of dual LP balls occurs when an edge is saturated by dual variables whose corresponding cuts contain terminals from distinct requests. Note that after buying the extra path between s_i and s_j and/or that between t_i and t_m, the shortest path connecting p_i may have changed to include more newly set zero-weight edges. We do not assign p_i a new cost class from this change. Since p_i, p_j and p_m belong to the same cost class, buying the paths from s_i to s_j and/or t_i to t_m increases the cost by a factor of at most 3. This is done by considering a *dual auxiliary graph*, similar to the auxiliary graph considered in the previous section, where the terminals at centers of successfully placed dual LP balls correspond to vertices of the dual auxiliary graph, and paths bought between these vertices (due to dual LP balls which could not be placed) correspond to the dual auxiliary graph edges. We show in Lemma 5.3.2 that the auxiliary graph is acyclic, which immediately provides an upper bound on the number of edges based on the number of vertices. This shows two things, *(i)* that the number of dual LP balls which could not be placed is sufficiently small, and *(ii)* that $|C_j|$ can be upper bounded in terms of the size of the corresponding dual solution $|D_j|$. Since $|C_j|$ is equal to the number of nodes, $|D_j|$, plus the number of edges, which is at most $|D_j| - 1$, in the dual auxiliary graph, we get that

$$|C_j| \leq |D_j| + |D_j| - 1.$$

From Lemma 5.3.1 we get that each edge in the dual auxiliary graph for class C_j corresponds to a path bought by the algorithm of cost at most 2^{j-2}, add to this the fact that each request in C_j cost

Algorithm 13 at most $3 \cdot 2^j$ to connect, we can bound the total cost of each class by the corresponding dual solutions as follows,

$$3 \cdot 2^j |C_j| \leq 3 \cdot 2^j (|D_j| + |D_j| - 1) = 12k \left(2\frac{2^{j-2}}{k} |D_j| - \frac{2^{j-2}}{k} \right)$$

where $\frac{2^{j-2}}{k}|D_j| = \vec{1}^T \cdot \vec{y}_i$ is the value of the dual solution D_j. This allows us to bound the total cost of Algorithm 13 by the desired amount, by summing over the cost classes.

Algorithm 13 Generalized Greedy Steiner Cover Algorithm

procedure $GGSC(H = (V, E, w), R)$
  $T \leftarrow \emptyset$
  $\vec{y} \leftarrow 0$
  while $i \leq n$ **do**
    $j \leftarrow$ cost class of p_i such that $Greedy(p_i) \in [2^{j-1}, 2^j)$
    $\tau_1, \vec{\Delta y}_1 \leftarrow ComputeDualLPBall(H, \vec{y}, s_i, \frac{2^{j-2}}{k})$
    $\tau_2, \vec{\Delta y}_2 \leftarrow ComputeDualLPBall(H, \vec{y}, t_i, \frac{2^{j-2}}{k})$
    if $\tau_1 = \frac{2^{j-2}}{k}$ and $\tau_2 = \frac{2^{j-2}}{k}$ **then**
      $\vec{y} \leftarrow \vec{y} + \vec{\Delta y}_1 + \vec{\Delta y}_2$
    if $\tau_1 < \frac{2^{j-2}}{k}$ **then**
      Let s' be the previous request whose dual LP ball intersected that of s_i.
      $T \leftarrow T \cup IncludePath(H, s_i, s')$
    else
      $\vec{y} \leftarrow \vec{y} + \vec{\Delta y}_1$
    if $\tau_2 < \frac{2^{j-2}}{k}$ **then**
      Let t' be the previous request whose dual LP ball intersected that of t_i.
      $T \leftarrow T \cup IncludePath(H, t_i, t')$
    else
      $\vec{y} \leftarrow \vec{y} + \vec{\Delta y}_2$
    $T \leftarrow T \cup IncludePath(H, s_i, t_i)$
  return T
procedure $IncludePath(H = (V, E, w), u \in V, v \in V)$
  $S \leftarrow \emptyset$
  Compute shortest path P in H connecting s_i to t_i
  $S \leftarrow S \cup P$
  for $e \in P$ **do**
    $w(e) \leftarrow 0$
  return S

Example:

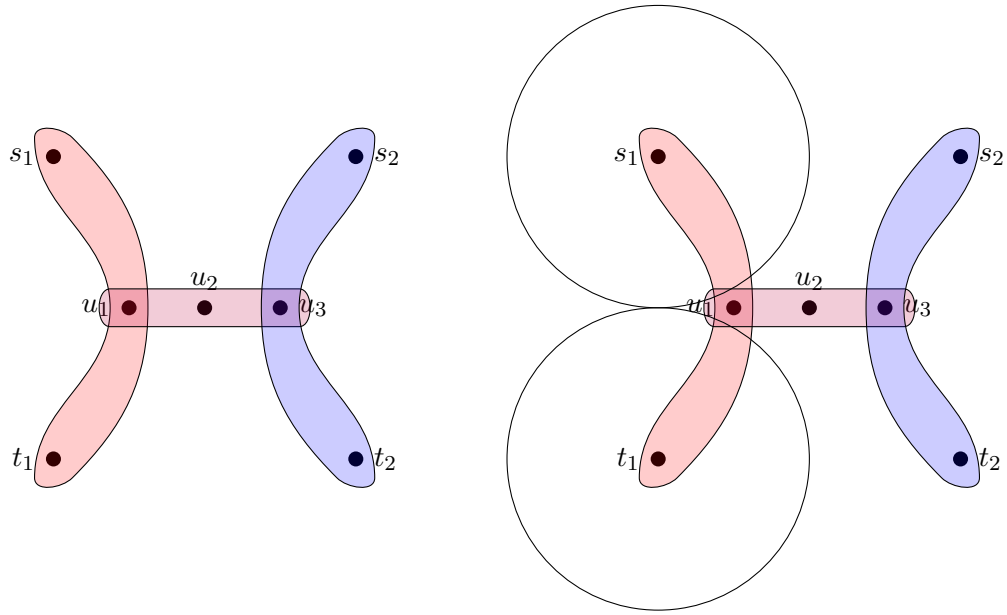

Figure 5.4: Example input for $GGSC$ Algorithm: first request.

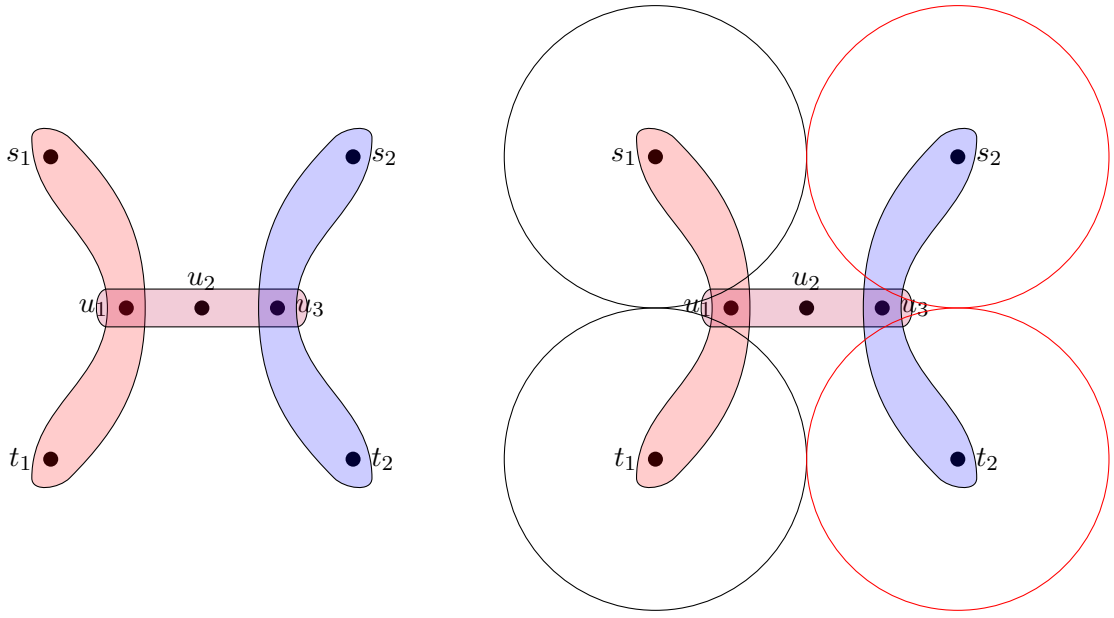

Figure 5.5: Example input for $GGSC$ Algorithm: second request.

After the first request in Figure 5.4 $p_1 = (s_1, t_1)$, *GGSC* places balls centered at s_1, t_1 and selects the edge $\{s_1, u_1, t_1\}$ to be in the solution. Since it is the only request, no balls were placed previously to intersect with and the algorithm simply buys the shortest path connecting the terminals

in p_1. The terminals s_1 and t_1 are added to the dual auxiliary graph, but they are not connected by an edge. After the second request in Figure 5.5, *GGSC* checks if the balls about s_2, t_2 can be placed without overlapping those previously placed about s_1, t_1. If they can, they are placed and the algorithm selects edge $\{s_2, u_3, t_2\}$ to add to the solution. The terminals s_2 and t_2 are added to the dual auxiliary graph, and no edges are added. If they cannot be placed, then both remaining edges are added to the solution. This achieves two things, first it connects the request pair p_2, and second it connects the terminals to the previous ones who prevented the balls from being placed. s_2 and t_2 are not added to the dual auxiliary graph, instead an edge is added to it between s_1 and t_1.

Since the placement of dual LP balls can now result in an infeasible solution, we must prove that in this case we can drop some dual assignments without hurting the asymptotic performance obtained in the analysis of OSCH. To do this, we first show in the following lemma that some path of small weight must exist between otherwise unrelated terminals for their dual LP balls to intersect. We want to buy this path, but need to be sure that the cost of doing so will not grow out of hand for our desired result.

Lemma 5.3.1. *When the if statement on line 10 (or 15) of Algorithm 13 is true, there is a path of saturated hyperedges from the corresponding terminal(s) in p_i to some previously requested terminal(s) s' (and t') corresponding to tight constraints in $\vec{y} + \vec{\Delta}y$.*

Proof. The if statement on line 10 (or 15) being true implies that the if statement on line 16 of $ComputeDualLPBall$ was true. In other words, there was an edge which was partially saturated by a previous request, s', and which one of the terminals of the current request, p_i, contributed to fully saturating, say s_i. Then by Lemma 4.3.1 there is a subgraph of saturated edges connecting up to k terminals, including s_i and at least one other s' in class C_j, of weight at most 2^{j-2}. □

Now we show that the number of extra paths bought by Algorithm 13 is small, which combined with the result from Lemma 5.3.1 allows us to guarantee that the total cost of these extra paths is not too large. The lemma and proof are adapted from a similar result from the online Steiner forest problem in graphs, a high level description of which can be found in [53].

Lemma 5.3.2. *The dual auxiliary graph for each cost class C_i, $1 \leq i \leq \log n$, implicitly constructed by Algorithm 13 is acyclic.*

Proof. This proof is similar to the proof of Lemma 5.2.3. By contradiction, assume there exists a cycle in the dual auxiliary graph for class C_j, and let (s_i, t_i) be the request which introduced this cycle via an auxiliary edge between terminals v_2 and v_3 (see Figure 5.2 for an illustration). Then the algorithm found the shortest path in H to connect them had cost greater than 2^{j-1}. Since the balls centered at s_i and v_2 intersect, $PlaceDualLPBall$ must have terminated for some $\gamma < \frac{2^{j-2}}{k}$. Then by Lemma 5.3.1 there is a subgraph connecting s_i and v_2 of weight less than 2^{j-2}. Following the same line of reasoning, we can say that the same is true for t_i and v_3. Now consider the other edges in the cycle of the auxiliary graph: they correspond to a zero-cost path for the algorithm between all previously requested terminals in the cycle. Then the distance in H_{i-1} between s_i and t_i satisfies

$$\text{dist}_{H_{i-1}}(s_i, v_2) < 2 \cdot 2^{j-2} = 2^{j-1}.$$

This now contradicts that the terminals were in cost class C_j, since the cost classes are assigned by the cost of $Greedy$ to satisfy them. We can therefore conclude that the auxiliary graphs imposed by each cost class is a tree. □

The following theorem establishes the main result of this section. Recall that for the lower bound, there were actually exponentially many more vertices than requests. This suggests that the average-case performance could actually be better than what the pessimistic view of the competitive ratio implies.

Theorem 5.3.3. *$GGSC$ (Algorithm 13) is $O(k \log n)$-competitive for the GOSCH problem w.r.t. k-restricted hypergraphs, where n is the length of the input sequence.*

Proof. Let R be the input sequence of length n for some instance of the GOSCH problem. Let C be the solution obtained by the $GGSC$ algorithm, and OPT denote some offline optimal algorithm. We begin by considering the input instance with normalized edge weights, where the weight of

each edge is multiplied by $\frac{n}{OPT(R)}$. Then, we consider partitioning C into cost classes C_j for $0 \leq j \leq \log n$. Since each terminal in class C_j cost at most 2^j to connect, we get that

$$GGSC(C_j) \leq 3 \cdot 2^j |C_j|$$

and the factor of 3 comes from the fact that $GGSC$ pays for at most 3 connections within a class for a given request (Algorithm 13 lines 12, 17, 20).

$$3 \cdot 2^j |C_j| = 12k \frac{2^{j-2}}{k} |C_j|$$

Now consider the dual auxiliary graph for the cost class C_j, whose vertices are terminals of C_j whose dual-LP-balls were placed without intersection and whose edges are terminals whose dual-LP-balls could not be due to resulting in infeasible variable assignments, i.e., the dual-LP-balls intersected previously placed ones. From Lemma 5.3.2 we know that this dual auxiliary graph is acyclic, so if there are $|D_j|$ nodes there are at most $|D_j|-1$ edges. Then we get that $|C_j| \leq 2|D_j|-1$. Note that $|D_j|$ is then the number of dual variables whose dual-LP-balls in dual solution D_j were successfully increased in radius to $\frac{2^{j-2}}{k}$ without intersecting previously placed dual-LP-balls in that class. Then

$$12k \frac{2^{j-2}}{k} |C_j| = 12k \left(\frac{2^{j-2}}{k} (2|D_j| - 1) \right)$$

$$= 12k \left(2\frac{2^{j-2}}{k} |D_j| - \frac{2^{j-2}}{k} \right) = 24k(\vec{1}^T \cdot \vec{y}_j) - 3\frac{2^j}{k}.$$

Since $\vec{1}^T \cdot \vec{y}_j \leq OPT(R)$ and $3\frac{2^j}{k} > 0$,

$$24k(\vec{1}^T \cdot \vec{y}_j) - 3\frac{2^j}{k} = O(k \cdot OPT(R))$$

Summing over all cost classes, we get

$$GGSC(R) = O\left(\sum_{j=1}^{\log n} GGSC(C_j) \right) = \sum_{j=1}^{\log n} O\left(k \cdot OPT(R)\right) = O(k \log n) \cdot OPT(R).$$

□

Chapter 6

Conclusions and Future Work

We provide a brief summary of the results of this thesis, and discuss some related open problems.

We consider online Steiner cover problems in hypergraphs, which are generalizations of ordinary graphs, whose internal relations go beyond what can be captured in a pairwise manner. Results in this setting are more often than not backwards compatible with the more common ordinary graph setting, given there is no restriction on the size of edges to be greater than 2. The results in this thesis have considered only the upper restriction on edge sizes, hypergraphs of rank at most k, generalizing some results from ordinary graphs into the hypergraph setting.

In Chapter 4 we posed the online Steiner cover problem in Hypergraphs, where given a hypergraph $H = (V, E, w)$ offline and request sequence $R = (r_1, r_2, \ldots, r_n)$ for $r_i \in V$, $\forall i \in [n]$ online, we must connect each request to the root r_1 in the sequence R via a minimal edge-weight subhypergraph of H. This problem is a generalization of the Steiner tree problem in graphs, where for the hypergraph case we do not impose the constraint that the solution be a hypertree but instead focus on minimizing the edge-weight of a solution only. We analyzed the $Greedy$ algorithm to show that the competitive ratio is $O(k \cdot \log n)$, where k is the rank of the hypergraph. This analysis was done from two perspectives: the first was a combinatorial approach in Section 4.2, and the second was an LP-based approach in Section 4.3. Both these approaches had proven successful in the past for the

graph Steiner tree problem, where the rank of the hypergraph is at most 2 [39]. In the combinatorial approach (Section 4.2), we implemented the $HDFS$ algorithm to construct a 2-uniform path graph out of some optimal solution, where each edge of the optimal solution appeared at most $2(k-1)$ times in this construction. Then, the analysis of [39] could be applied directly to bound the cost of shortest path distances between requests in this path with those which were revealed previously. In the LP-approach (Section 4.3), the primal and dual LP relaxations of the problem were formulated and an algorithm for assigning dual variables was considered in order to bound the cost of $Greedy$ not by some optimal offline solution (not directly, anyway), but by the primal-dual bounds obtained from LP theory. By splitting the primal solution into $\log n$ cost classes, we could construct a separate, feasible dual solution for each class. Each primal cost class was then bounded above by a factor $2k$ of its corresponding dual solution. By summing over the primal cost classes we bounded the solution by $2k$ times the sum of dual solutions, and since each dual solution is less than the cost of any optimal solution this gave us the desired bound.

In Chapter 5 we continued to generalize online Steiner problems in graphs to hypergraphs by considering the online generalized Steiner cover problem in hypergraphs. This problem is a natural extension of the online Steiner forest problem in graphs. The requests now arrive in pairs $R = (p_1 = (s_1, t_1), p_2 = (s_2, t_2), \ldots, p_n = (s_n, t_n))$, where $s_i, t_i \in V \ \forall i \in [n]$ and the goal is to find a minimal edge-weight subhypergraph of H where only the terminals which arrived together are required to be connected. For this problem, we show that the corresponding $Greedy$ algorithm has a competitive ratio bounded as follows

$$\Omega(k \log n) \leq \rho(Greedy) \leq O(k \log^2 n).$$

The analysis here was done with a combinatorial approach in Section 5.2, again adapted from the 2-uniform case of previous work [6]. The idea of this approach was to bound the occurences of costly requests. This was done by constructing a correspondence between the number of requests exceeding a certain threshold and the number of edges of an auxiliary graph. Many adaptations were needed to construct this auxiliary graph from a hypergraph instead of an ordinary graph, but once this was done the argument from [6] followed. We then showed that another algorithm,

called $GGSC$, which can be seen as the greedy approach empowered with the insight gained from keeping track of the dual LP, obtains a competitive ratio of $O(k \log n)$ in Section 5.3. The algorithm attempted to assign the dual variables, conceptualized as growing a dual LP ball about each request. If the process terminated successfully, the algorithm proceeded in the usual greedy fashion. If the process was unsuccessful, and the dual LP ball would have resulted in an infeasible variable assignment, then the algorithm added extra paths to connect nearby terminals corresponding to these broken constraints before recomputing the shortest path of the current request and finally adding it to the solution.

Open Problem 6.1. It is conjectured that $Greedy$ algorithm is $O(\log n)$-competitive for the online Steiner forest problem in ordinary graphs [6]. Inspired by that, we conjecture that the simple $Greedy$ algorithm for GOSCH problem is $O(k \log n)$-competitive for hypergraphs of rank k. Regardless of validity of this conjecture, it remains an important open problem to establish the tight bound on the competitiveness of $Greedy$ for the GOSCH problem.

Open Problem 6.2. This thesis analyzed the competitive ratio on simple and k-restricted hypergraphs. A natural next step would be to further restrict the family of hypergraphs for the considered problems to *linear* hypergraphs, for which any pair of edges intersect in at most one vertex. The lower bound provided here was a k-uniform hypergraph, but it was certainly not linear. The number of edges for a linear k-uniform hypergraph on N nodes is $O\left(\frac{N^2}{k^2}\right)$. Hypergraphs with a polynomial upper bound on the number of edges are of use in practical applications because many computational problems become tractable.

Open Problem 6.3. It is interesting to analyze OSCH and GOSCH in stochastic settings, such as known and unknown i.i.d. as well as under the random order input model.

Open Problem 6.4. The online Steiner problem in ordinary graphs has been studied under the oracle advice model in [10], and under prediction with error (a.k.a. ML-based advice) model in [7] and [67]. So far, the considered prediction and advice models have all entailed advice about the requests themselves. It would be interesting to study other kinds of advice, for example, advice based on an optimal solution, both in the setting of ordinary graphs as well as hypergraphs.

Open Problem 6.5. We studied a generalization of the online Steiner problem in graphs by considering the setting of hypergraphs, but there are more ways one can generalize. For example, inspired by the survivable network design problem [69], we can consider an extension of GOSCH in which requests arrive as triples (s_i, t_i, r_i) where s_i and t_i are terminal vertices which must be connected and r_i is the connection requirement, i.e., the number of distinct hyperedge-disjoint paths which must connect those terminals.

Bibliography

[1] Noga Alon, Baruch Awerbuch, and Yossi Azar. The online set cover problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 100–105, 2003.

[2] Noga Alon and Yossi Azar. On-line steiner trees in the euclidean plane. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, SCG '92, page 337–343, New York, NY, USA, 1992. Association for Computing Machinery.

[3] Noga Alon, Dana Moshkovitz, and Shmuel Safra. Algorithmic construction of sets for k-restrictions. *ACM Trans. Algorithms*, 2:153–177, 2006.

[4] Spyros Angelopoulos. Parameterized analysis of the online priority and node-weighted steiner tree problems. *Theor. Comp. Sys.*, 63(6):1413–1447, August 2019.

[5] Giorgio Ausiello and Luigi Laura. Directed hypergraphs: introduction and fundamental algorithms—a survey. *Theoretical Computer Science*, 658:293–306, 2017.

[6] Baruch Awerbuch, Yossi Azar, and Yair Bartal. On-line generalized steiner problem. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, page 68–74, USA, 1996. Society for Industrial and Applied Mathematics.

[7] Yossi Azar, Debmalya Panigrahi, and Noam Touitou. *Online Graph Algorithms with Predictions*, pages 35–66. Society for Industrial and Applied Mathematics, 2022.

[8] Egon Balas and Manfred W Padberg. On the set-covering problem. *Operations Research*, 20(6):1152–1161, 1972.

[9] Étienne Bamas, Marina Drygala, and Andreas Maggiori. An improved analysis of greedy for online steiner forest. In *ACM-SIAM Symposium on Discrete Algorithms*, 2021.

[10] Kfir Barhum. Tight bounds for the advice complexity of the online minimum steiner tree problem. In Viliam Geffert, Bart Preneel, Branislav Rovan, Július Štuller, and A. Min Tjoa, editors, *SOFSEM 2014: Theory and Practice of Computer Science*, pages 77–88, Cham, 2014. Springer International Publishing.

[11] Gregor Baudis, Clemens Gröpl, Stefan Hougardy, Till Nierhoff, and Hans Jürgen Prömel. Approximating minimum spanning sets in hypergraphs and polymatroids. In *Proc. ICALP*. Citeseer, 2000.

[12] Claude Berge. *Hypergraphs: Combinatorics of Finite Sets*. "North-Holland", 1989.

[13] Piotr Berman and Chris Coulston. On-line algorithms for steiner tree problems (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 344–353, New York, NY, USA, 1997. Association for Computing Machinery.

[14] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets möbius: fast subset convolution, 2006.

[15] Béla Bollobas. *Extremal Graph Theory*. Dover Publications, Inc., USA, 2004.

[16] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 2005.

[17] Jarosław Byrka, Fabrizio Grandoni, Thomas Rothvoss, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1), February 2013.

[18] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1):353–371, 2000.

[19] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[20] Gérard Cornuéjols, George Nemhauser, and Laurence Wolsey. The uncapicitated facility location problem. Technical report, Cornell University Operations Research and Industrial Engineering, 1983.

[21] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2006.

[22] Erik D. Demaine, MohammadTaghi Hajiaghayi, and Philip N. Klein. Node-weighted steiner tree and group steiner tree in planar graphs. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, pages 328–340, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[23] Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered pcp and the hardness of hypergraph vertex cover. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 595–601, New York, NY, USA, 2003. Association for Computing Machinery.

[24] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014.

[25] Stefan Dobrev, Jeff Edmonds, Dennis Komm, Rastislav Královič, Richard Královič, Sacha Krug, and Tobias Mömke. Improved analysis of the online set cover problem with advice. *Theoretical Computer Science*, 689:96–107, 2017.

[26] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[27] Cees Duin and Stefan Voß. Steiner tree heuristics — a survey. In *Operations Research Proceedings 1993*, pages 485–496, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

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

[29] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

[30] Yue Gao, Shuyi Ji, Xiangmin Han, and Qionghai Dai. Hypergraph computation. *Engineering*, 40:188–201, 2024.

[31] Naveen Garg, Anupam Gupta, Stefano Leonardi, and Piotr Sankowski. Stochastic analyses for online combinatorial optimization problems. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, page 942–951, USA, 2008. Society for Industrial and Applied Mathematics.

[32] Naveen Garg, Goran Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *Journal of Algorithms*, 37(1):66–84, 2000.

[33] Michel X. Goemans, Neil Olver, Thomas Rothvoß, and Rico Zenklusen. Matroids and integrality gaps for hypergraphic steiner tree relaxations. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 1161–1176, New York, NY, USA, 2012. Association for Computing Machinery.

[34] Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, page 307–316, USA, 1992. Society for Industrial and Applied Mathematics.

[35] MohammadTaghi Hajiaghayi, Vahid Liaghat, and Debmalya Panigrahi. Near-optimal online algorithms for prize-collecting steiner problems. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I 41*, pages 576–587. Springer, 2014.

[36] S. L. Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1(2):113–133, 1971.

[37] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[38] Florian Hörsch and Zoltán Szigeti. Steiner connectivity problems in hypergraphs. *Information Processing Letters*, 183:106428, 2024.

[39] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[40] David S Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49, 1973.

[41] David S. Johnson, Maria Minkoff, and Steven Phillips. The prize collecting steiner tree problem: theory and practice. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 760–769, USA, 2000. Society for Industrial and Applied Mathematics.

[42] R. M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, 1975.

[43] Hervé Kerivin and A Ridha Mahjoub. Design of survivable networks: A survey. *Networks: An International Journal*, 46(1):1–21, 2005.

[44] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *Journal of Computer and System Sciences*, 74(3):335–349, 2008. Computational Complexity 2003.

[45] P. Klein and R. Ravi. A nearly best-possible approximation algorithm for node-weighted steiner trees. *Journal of Algorithms*, 19(1):104–115, 1995.

[46] Simon Korman. On the use of randomization in the online set cover problem. *Weizmann Institute of Science*, 2:34, 2004.

[47] A Yu Levin. Algorithm for the shortest connection of a group of graph vertices. In *Dokl. Akad. Nauk SSSR*, volume 200, pages 773–776, 1971.

[48] Ivana Ljubić. Solving steiner trees: Recent advances, challenges, and perspectives. *Networks*, 77:177 – 204, 2020.

[49] Nelson Maculan. The steiner problem in graphs**this work was partially supported by conselho nacional de desenvolvimento cientifico e tecnologico, cnpq 300195-83 and by finep. In Silvano Martello, Gilbert Laporte, Michel Minoux, and Celso Ribeiro, editors, *Surveys in Combinatorial Optimization*, volume 132 of *North-Holland Mathematics Studies*, pages 185–211. North-Holland, 1987.

[50] Bálint Molnár. Applications of hypergraphs in informatics: A survey and opportunities for research. *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE SECTIO COMPUTATORICA (ISSN: 0138-9491)*, 42:261–282, 09 2014.

[51] A. Moss and Y. Rabani. Approximation algorithms for constrained node weighted steiner tree problems. *SIAM Journal on Computing*, 37(2):460–481, 2007.

[52] J. Nederlof. Fast polynomial-space algorithms using inclusion-exclusion. *Algorithmica*, 65(4):868–884, 2013.

[53] Debmalya Panigrahi. COMPSCI 638: Graph Algorithms, Lecture 16. https://courses.cs.duke.edu/fall19/compsci638/fall19_notes/lecture16.pdf, 2019. [Online; accessed 22-02-2024].

[54] Jaap Pedersen and Ivana Ljubić. *Prize-Collecting Steiner Tree Problem and Its Variants*, pages 1–11. Springer International Publishing, Cham, 2025.

[55] Tobias Polzin and Siavash Vahdati Daneshmand. On steiner trees and minimum spanning trees in hypergraphs. *Operations Research Letters*, 31(1):12–20, 2003.

[56] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability pcp characterization of np. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 475–484, 1997.

[57] Gabriel Robins and Alex Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discret. Math.*, 19:122–134, 2005.

[58] K.H. Rosen. *Handbook of Discrete and Combinatorial Mathematics*. Discrete Mathematics and Its Applications. Taylor & Francis, 1999.

[59] Faryad Darabi Sahneh, Stephen Kobourov, and Richard Spence. Approximation algorithms for priority steiner tree problems. In Chi-Yeh Chen, Wing-Kai Hon, Ling-Ju Hung, and Chia-Wei Lee, editors, *Computing and Combinatorics*, pages 112–123, Cham, 2021. Springer International Publishing.

[60] Arie Segev. The node-weighted steiner tree problem. *Networks*, 17(1):1–17, 1987.

[61] David B Shmoys, Éva Tardos, and Karen Aardal. Approximation algorithms for facility location problems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 265–274, 1997.

[62] Hao Tang, Genggeng Liu, Xiaohua Chen, and Naixue Xiong. A survey on steiner tree construction and global routing for vlsi design. *IEEE Access*, 8:68593–68622, 2020.

[63] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[64] Jens Vygen. Faster algorithm for optimum steiner trees. *Information Processing Letters*, 111(21):1075–1079, 2011.

[65] David Michael Warme. *Spanning trees in hypergraphs with applications to Steiner trees*. University of Virginia, 1998.

[66] Pawel Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987.

[67] Chenyang Xu and Benjamin Moseley. Learning-augmented algorithms for online steiner tree. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(8):8744–8752, Jun. 2022.

[68] Martin Zachariasen and Pawel Winter. *Obstacle-Avoiding Euclidean Steiner Trees in the Plane: An Exact Algorithm*, pages 286–299. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[69] Liang Zhao, Hiroshi Nagamochi, and Toshihide Ibaraki. A primal–dual approximation algorithm for the survivable network design problem in hypergraphs. *Discrete Applied Mathematics*, 126:275–289, 03 2003.

[70] Tomasz Łuczak. Size and connectivity of the k-core of a random graph. *Discrete Mathematics*, 91(1):61–68, 1991.