Cutting the Chord: Interleaved and Demand Aware Skip Graphs

A Thesis
Presented to
The Established Interdisciplinary Committee
for Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Hrishee M. Shastri

May 2021

Approved for the Committee
(Mathematics and Computer Science)

James D. Fix

Marcus Robinson

Table of Contents

Introduction **1**
0.1 Chord 3
0.1.1 Network Topology 3
0.1.2 Routing 4
0.2 Thesis Overview 5
0.2.1 A Nod to Self-Adjusting Networks 5

Chapter 1: Skip Lists and Skip Graphs **7**
1.1 The Skip List 7
1.1.1 Structure 7
1.1.2 Search 8
1.1.3 Insert 8
1.2 Skip Graphs 10
1.2.1 Structure 10
1.2.2 Balanced Skip Graphs 14
1.2.3 The `search` algorithm 15

Chapter 2: The Minimum Expected Path Length Problem **19**
2.1 Preliminaries 19
2.2 Problem Statement 22
2.3 Hardness of MEPL-SG and MEPL-SG* 23
2.3.1 The Minimum Linear Arrangement Problem 23
2.3.2 Reductions 23
2.3.3 Variants 25

Chapter 3: The Optimal Skip Graph Problem **27**
3.1 Problem Statement 27
3.2 Enumerating Skip Graphs 27
3.3 Interleaved Skip Graphs 29
3.3.1 Cutting the Chord 30
3.3.2 Derivation of Average Path Length 32
3.3.3 Conjecture of Optimality 36

Chapter 4: Optimal Skip Graph Heuristics **39**
4.1 Contraction Heuristics 39

4.1.1 Edge Contractions 40
4.1.2 Comprehensive Weight Function 40
4.1.3 Edge Contraction Subroutines 45
4.2 Empirical Performance Over Random Demand 47
4.2.1 Scaling 47
4.2.2 Spread 50

Open Problems **55**

References **57**

List of Tables

4.1 Ratio between average WPL of `Random` and `CMM` in Fig. 4.7. 47
4.2 Mean and standard deviation (in parentheses) of weighted path lengths for `Random`, `CME`, and `CMM` as given in Fig. 4.13. 52

List of Figures

1 Underlying network on $n = 7$ nodes and an overlay network for $m = 4$. Each node is labeled by a number between 0 and 6. Note that the links between 3 and 1, between 4 and 6, and between 3 and 6 in the overlay network abstract the paths between them in the underlying network. 2

2 Chord on 16 nodes, C_{16}. 3

1.1 A skip list on $n = 6$ nodes with the two sentinel nodes $-\infty, \infty$, and height $H = 4$. In this example, x_0.height = 0, x_1.height = 2, where x_1 has key 1 and x_0 has key 0. 8

1.2 Skip list from Fig. 1.1 searches for queried key 5, starting at `HEAD` ($-\infty$). 9

1.3 Example skip graph $\mathcal{G}$ on $n = 6$ nodes. Identifiers are at the top right of each node. Note that the height of $\mathcal{G}$ is 3, and that every node appears in a singleton linked list. 12

1.4 The skip graph in Fig. 1.3 determines a graph. Edge colors only distinguish the level in which the connected nodes first appear as neighbors, but have no other function otherwise. 12

1.5 Skip graph from Fig. 1.3 represented as a binary prefix tree. 13

1.6 Sub-skip graph $\mathcal{G}_0$ of skip graph $\mathcal{G}$. 13

1.7 A skip list obtained from skip graph in Fig. 1.3 by grouping together all linked lists that contain key 31. 14

1.8 Balanced skip graph $\mathcal{B}$ (top) and skip graph $\mathcal{G}$ (bottom). 15

1.9 The skip graph search path is not always the shortest path. Search is also not commutative. Left: Prefix tree representation. Right: View as a network. 17

2.1 Example demand graph on 3 nodes modeling the uniform request distribution. 20

2.2 An embedding ϕ from the demand graph D to the path graph G. . . 22

2.3 Minimum linear arrangement ϕ (right) of the star graph on 9 nodes (left), where every edge has unit weight. Note that ϕ is a bijection from the nodes of the star to $\{1, ..., 9\}$. 24

2.4 Balanced skip graph construction in proof of Corollary 2.3.4 (left) produces a path graph, for $n = 2^3 = 8$. 26

3.1 An example prefix tree layout for $n = 3$. Each identifier is determined by a path from the root of the prefix tree to a leaf. Because the number of leaves in a prefix tree representation is the number of nodes in the skip graph, there are $n!$ ways to assign the nodes to the leaves. 28
3.2 Interleaved skip graph on $n = 8$ nodes. (Left) Binary prefix tree representation. (Right) Corresponding network topology. 30
3.3 The interleaved skip graph is isomorphic ($\simeq$) to BiChord with the straddling edges (in red) removed. 31
3.4 Node 1 performs a skip list search for node 12 in $\mathcal{I}_4$, the interleaved skip graph on 16 nodes, represented as a prefix tree. Note that the higher the level, the larger the power of two is for the rank difference between neighbors. 32
3.5 Route from 1 to 12 in Fig. 3.4 represented as a network (left). Routing from 1 to 12 in Chord (right). Note that the two search paths are identical. 33
3.6 (Left) $\mathcal{I}_4$ routes from 15 to 0. (Right) Chord encounters a straddling edge when routing from 15 to 0 in the clockwise direction. Assuming bidirectional links, routing counterclockwise yields the same search path as the interleaved skip graph. 33
3.7 (Left) Uniform clustering demand D with disjoint subsets $V_1 = \{0, 1, 6, 7\}$ and $V_2 = \{2, 3, 4, 5\}$. Edge weights and directed edges omitted for simplicity. (Middle) Conjectured optimal skip graph for D represented as a prefix tree, comprising of the interleaved skip graphs for V_1 and V_2. (Right) Corresponding network topology. 37

4.1 (Left) Before contracting edge (u, v). (Right) After contracting edge (u, v), merging u and v into a node w. 40
4.2 Contracting edge $(\mathcal{S}, \mathcal{T})$ in a heuristic graph H results in a new node, which corresponds to merging skip graphs $\mathcal{S}$ and $\mathcal{T}$ together into a new skip graph $\mathcal{S} \oplus \mathcal{T}$. 41
4.3 Computing edge weight $(\mathcal{S}, \mathcal{T})$ using the comprehensive weight function in Definition 4.1.3 43
4.4 Proof of Theorem 4.1.1. In this figure, $w_{\mathcal{S}} \neq w_{\mathcal{R}}$, but it could be possible for them to be equal, in which case $\delta = 0$. 45
4.5 (Left) Maximal matching. (Right) Maximum cost matching (which is also a maximal matching). 46
4.6 Example execution of the contraction heuristic in Example 4.1.2. Circles represents nodes in the heuristic graph H. Arrows transition between edge contractions. 46
4.7 Weighted path length of size n skip graphs returned by the heuristics. Each data point is averaged over 100 trials. Due to computational constraints, sufficient data could not be collected for $n > 80$. 48
4.8 Spine skip graph on $n = 6$ nodes. Each level contributes only at most one link to the network. 48

4.9 Skip graph output by `CME` when given uniform demand on $n = 20$ nodes. Note the spine sub-skip graph $\mathcal{G}_{01} = \{0, 1, 2, 5, 8, 11, 12, 15, 17, 19\}$. 49
4.10 Skip graph output by `CMM` when given uniform demand on $n = 20$ nodes. 50
4.11 A spine skip graph is optimal for single-source demand. 50
4.12 Histograms of approximation factors for $n = 7$. Here, the means are $\mu_{random} = 1.45$, $\mu_{CMM} = 1.05$, and $\mu_{CME} = 1.04$ and the standard deviations are $\sigma_{random} = 0.21$, $\sigma_{CMM} = 0.04$ and $\sigma_{CME} = 0.03$. 51
4.13 Histograms of weighted path length for $n = 30, 50, 70$ and 80. 52
4.14 Weighted path length over random demand, with the interleaved skip graph (Each data point averaged over at least 100 trials). 53

Abstract

Designing overlay network topologies that facilitate efficient routing for a given communication demand is an important task in network optimization. This thesis explores several demand-aware network design problems through the lens of the skip graph, an overlay network invented by Aspnes and Shah [1]. We show that the Minimum Expected Path Length (MEPL) problem remains NP-Complete for skip graphs under various constraints. We also present the interleaved skip graph, a highly connected skip graph with a striking resemblance to Chord [21], derive its average path length, and conjecture that it is optimal. Finally, we describe and implement two heuristics to find the optimal skip graph for a given communication demand, with empirical evidence that suggest they outperform random sampling.

Introduction

Suppose there are n linguists documenting the rare Reedian language in a village called Reed College. Each linguist is stationed at a different location on Reed's campus. To effectively coordinate their efforts, each linguist would like the ability to query any other linguist on the campus.

Suppose the linguists communicate with each other by way of cell phone, where each linguist has a distinct *identifier*: a cell number. Additionally, each cell phone only has enough space to store fewer than n other cell numbers[1]. The network of linguists can be modeled as a directed graph, where each node represents a linguist and a directed edge from node u to v represents the fact that linguist u has linguist v's cell number in their phone. This network is an example of a **peer-to-peer network**, which is a distributed system where each node (linguist, in this case) can both serve and request resource from other nodes. In particular, if node u desires to contact node v and does not have v's cell number stored, u must choose a *neighbor* (i.e. a linguist whose cell number u has stored) to pass the message further along to v. Choosing which neighbor to pass the message to is done via a *routing algorithm* (protocol).

Now, further suppose that m of the n ($m < n$) linguists are part of a more specialized task force dedicated to documenting the even rarer library-dwelling dialect of the Reedian language. For these m linguists to communicate amongst themselves using the existing network, the other $n - m$ linguists help relay messages to and from the m nodes. In this case, the m linguists will form a peer-to-peer **overlay network** built atop the existing network of n linguists. The overlay network is a logical (virtual) network, meaning that the links are abstractions for paths between nodes in the underlying network (Fig. 1). Every node in the overlay network operates under the same protocol. If there exists a path between every pair of the m nodes in the underlying network, then the underlying network can be abstracted away and the overlay network can be theoretically treated as a standalone undirected network.

Fundamentally, peer-to-peer overlay networks are used to efficiently search for resources shared amongst several machines, or *peers*. Many applications rely on overlay networks that are built atop the internet and thus require routing algorithms that scale well – that is, as the number of peers grow, routing between nodes in the network does not substantially degrade performance. Several peer-to-peer overlay network schemes have been designed and implemented in the last two decades, some of the most notable being Chord [21], Pastry [17], Tapestry [23], and CAN [16].

[1]Typically, each phone will only be able to store $O(\log n)$ or $O(1)$ numbers. All logarithms in this thesis are base 2 unless specified otherwise.

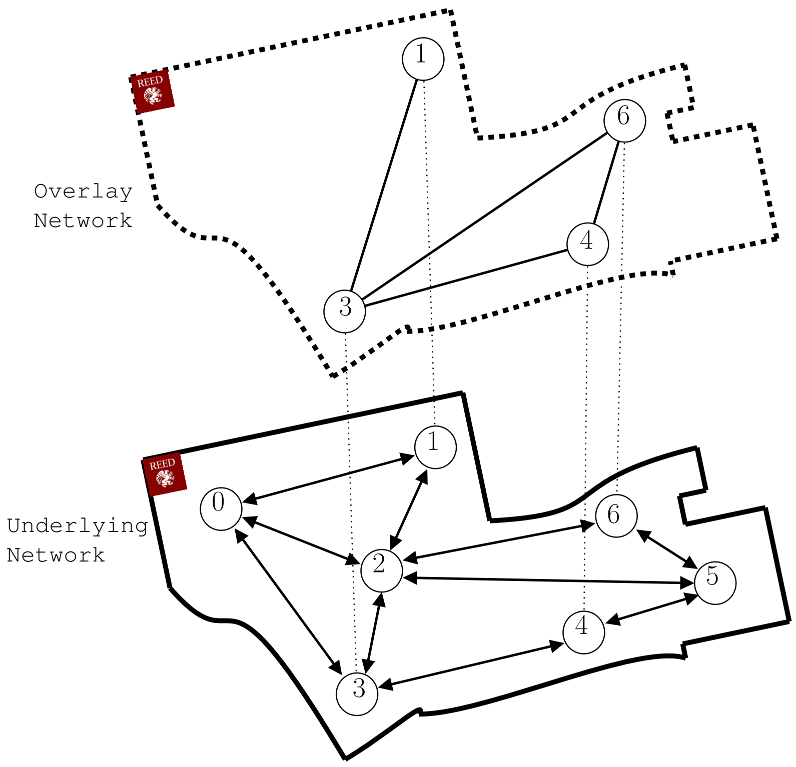
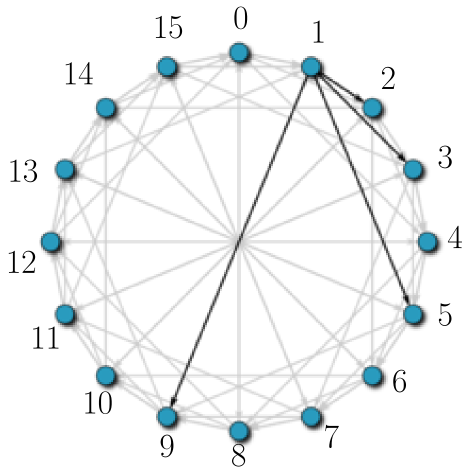
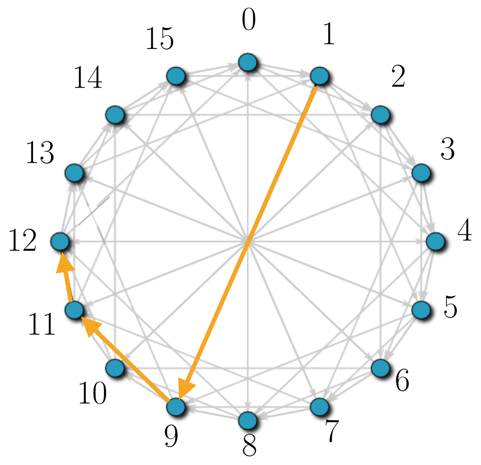


Figure 1: Underlying network on $n = 7$ nodes and an overlay network for $m = 4$. Each node is labeled by a number between 0 and 6. Note that the links between 3 and 1, between 4 and 6, and between 3 and 6 in the overlay network abstract the paths between them in the underlying network.



(a) Node 1's finger table contains the set of nodes $\{2, 3, 5, 9\}$.



(b) Routing on Chord. The route from node 1 to node 12 follows the path $1, 9, 11, 12$.

Figure 2: Chord on 16 nodes, C_{16}.

0.1 Chord

To get an idea of how these systems are designed, we briefly overview the structure and routing of Chord. The reader is encouraged to refer to the original paper [21] for a summary of insertion, deletion, fault tolerance, and implementation details.

0.1.1 Network Topology

Suppose we have n computers (nodes), each of which we identify with a distinct integer from $V = \{0, ..., n-1\}$. The basic idea is that Chord arranges these n nodes in a ring and mandates that each node maintain its own *finger table*. A specific node's finger table contains links to every node whose clockwise distance from itself on the ring is a power of two (Fig. 2a). More formally, the network topology given by Chord is a graph $C_n = (V, E)$ with directed edge set[2]

$$E = \bigcup_{u \in V} \{(u, (u + 2^i) \bmod n) \mid 0 \leq i \leq \lfloor \log_2 n \rfloor\}. \tag{1}$$

Note that this formulation allows each node to only store $\Theta(\log n)$ identifiers to other nodes. In our analogy with linguists, identifiers were cell numbers. In a real computer network, identifiers might typically be IP addresses. Here, we use integers for simplicity. Further note that links are not always bidirectional. If $(u, v) \in E$, then $(v, u) \in E$ when the clockwise distance between v and u equals the clockwise distance between u and v, which occurs only when u and v are directly opposite each other in the ring.

[2]Formulation borrowed from R. Seth Terashima's Reed Thesis [22].

0.1.2 Routing

Routing in Chord proceeds in a greedy fashion. If u wants to send a message to v, u forwards its message to the node w in its finger table that is closest to v, without overshooting v. Then w forwards the message in the same manner, and this process repeats until v is reached (Fig. 2b). At each routing step, the remaining distance to v is reduced by the largest power of two less than the clockwise ring distance between the current node and v.

To illustrate, consider the example in Fig. 2b, where node 1 aims to route to node 12. Node 1 first routes to node 9 because 8 is the largest power of two less than 11, the clockwise ring distance between node 1 and node 12. From there, node 9 routes to node 11 because 2 is the largest power of two less than $12 - 9 = 3$, and finally node 11 will deliver the message home to node 12, for a total cost of 3 hops. Note that node 9 should not route to node 13, even though node 13 is as close to node 12 on the ring as node 11 is. This is because overshooting 12 jeopardizes the route: 13 does not have 12 in its finger table.

A stylish way to reason about the route cost from u to v is to think in terms of binary representations. Recall that routing in Chord requires taking links that subtract a distinct power of two from the remaining distance. Thus, the number of '1' bits in the binary representation of the clockwise distance between u and v gives us the total number of hops to route from u to v, because each '1' bit represents a power of two that contributes to the total distance. For instance, consider routing from node 1 to node 12 in the previous example. The clockwise distance 11 has a binary representation of `1011`. This means that it takes three distinct powers of two to sum to 11 ($2^3 + 2^1 + 2^0$), each of which corresponds to a link taken by the route from node 1 to node 12. Thus the total number of hops is 3, as we showed. In general, the worst case routing cost in Chord must be $O(\log n)$ hops, as binary representations are already $O(\log n)$ bits to begin with.

We have established that in a size n network, Chord allows each node to only maintain $\Theta(\log n)$ links and also allows at most $O(\log n)$ hops for routing. Intuitively, this means that each node only needs to add one more link to its finger table as the network size doubles! Similarly, the worst case routing cost also only increases by one as the network size doubles.

BiChord

As an aside, there is a variant of Chord called BiChord [13] that maintains bidirectional links in the Chord graph C_n – that is, if $(u, v) \in E$, then $(v, u) \in E$ as well. Or more precisely, the edge set for BiChord is:

$$E = \bigcup_{u \in V} \Big(\{(u, (u + 2^i) \bmod n) \mid 0 \leq i \leq L\} \cup \{((u + 2^i) \bmod n, u) \mid 0 \leq i \leq L\} \Big)$$

where $L = \lfloor \log_2 n \rfloor$. The authors of BiChord show that routing in C_n can be made much more efficient by exploiting the bidirectionality of links. Others have independently presented routing algorithms for BiChord that are in fact optimal (of provably minimal hop distance) [9, 11].

0.2 Thesis Overview

Overlay networks are often designed with the assumption that each node is equally likely to communicate with every other node in the network. While this is a nice simplifying assumption, communication patterns (also called *demands*) can be highly non-uniform in many real world networks. As network designers, we gain an upper hand when we know the demand beforehand, because we can build an overlay network that is specifically tailored to the heartbeat of that communication pattern. This thesis explores various facets of this demand-aware network design paradigm through a specific type of overlay network called the skip graph [1].

Chapter 1 begins with a summary of a simple lookup data structure, the skip list [15], and an expose of the data structure inspired by the skip list, the skip graph. We characterize their structure, routing, and representations to serve as the foundation for the rest of the thesis.

Chapter 2 formulates the minimum expected path length problem (MEPL), which is a demand-aware node placement problem. In particular, we prove NP-completeness for MEPL when restricted to various families of skip graphs under different conditions.

Chapter 3 introduces the interleaved skip graph through the lens of the optimal skip graph problem, which is a demand-aware network design problem. We explore the interleaved skip graph's relationship with Chord, derive a closed form for its average path length, and discuss our conjecture for its optimality.

Chapter 4 describes, implements, and empirically analyzes heuristics for the optimal skip graph problem introduced in Chapter 3.

Finally, Chapter 5 concludes with several open problems and future directions arising from this thesis.

0.2.1 A Nod to Self-Adjusting Networks

As a nod to the origins of this thesis, we will very briefly mention self-adjusting networks. Before pivoting entirely to demand-aware skip graphs, the first several months of this thesis were originally devoted to the study of *self-adjusting networks*, specifically self-adjusting skip graphs. While demand aware networks are optimized for a given demand beforehand, self-adjusting networks dynamically adapt to the communication pattern over time — by incrementally and reactively transforming their own network topology after serving each communication request. The following papers may be of interest to those who would like to learn more:

- For a survey on the theoretical and algorithmic challenges in self-adjusting networks, see [4].

- For a self-adjusting skip graph, see [12], and for a self-adjusting skip list network, see [3].

- SplayNet [18], a self-adjusting peer-to-peer overlay network based on splay trees [20].

- See [8] for a comprehensive survey on practical developments in self-adjusting data center networks.

Chapter 1

Skip Lists and Skip Graphs

We begin by describing the form and function of the skip list, a simple lookup data structure [15] that we use as a stepping stone to understand skip graphs in more detail [1]. In particular, we overview skip graph structure and routing behavior in order to contextualize subsequent chapters.

1.1 The Skip List

In this section, we describe the form and function of the skip list in terms sufficient enough for this thesis. The reader is encouraged to refer to the full generality of the original paper [15] for more details.

1.1.1 Structure

We treat a **skip list** as an ordered collection of $n+2$ nodes $\mathcal{L} = \{x_{-\infty}, x_1, ..., x_n, x_\infty\}$ each holding a distinct key from $\{-\infty, k_1, ..., k_n, \infty\}$, where $k_1 < \cdots < k_n$ are non-negative integers. Each node $x \in \mathcal{L}$ has a field x.key where $x_{-\infty}$.key $= -\infty$, x_∞.key $= \infty$, and in general, x_i.key $= k_i$. The x_∞ and x_∞ nodes simply act as sentinels to mark the beginning and end of the skip list.

The nodes of a skip list $\mathcal{L}$ are organized as a hierarchy of doubly linked lists $\mathcal{L}_0 \subset \mathcal{L}_1 \subset \cdots \subset \mathcal{L}_H$ where H is the **height** of $\mathcal{L}$ and $\mathcal{L}_j$ is the linked list at level j. In particular, $\mathcal{L}_0 = \mathcal{L}$ with the order of the linked list determined by ascending order of key, and $\mathcal{L}_H = \{x_{-\infty}, x_\infty\}$. The height of each node x_i is x_i.height $= \max\limits_{h \leq H}\{h \mid x_i \in \mathcal{L}_h\}$, that is, the height of the highest linked list that contains x_i. This means $x_{-\infty}$.height $= x_\infty$.height $= H$ and for all other nodes, $0 \leq x_i$.height $< H$. We call $x_{-\infty}$ the **head** and x_∞ the **tail** of each linked list, and specifically, we let the head of the entire skip list be $\mathcal{L}$.`HEAD` $= x_{-\infty}$. See Fig. 1.1 for an example of a skip list.

Each node x_i maintains two arrays to determine the linked list structure: x_i.left and x_i.right for the nodes to the left and right of them at each level, respectively. Specifically, x.left$[j]$ gives the node to the left of x in linked list $\mathcal{L}_j$ and x.right$[j]$ gives the node to the right. Both x_∞.right$[j]$ and $x_{-\infty}$.left$[j]$ are assumed to be null,

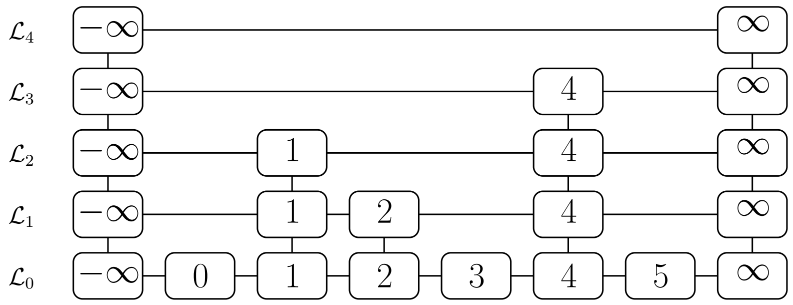


Figure 1.1: A skip list on $n = 6$ nodes with the two sentinel nodes $-\infty, \infty$, and height $H = 4$. In this example, x_0.height $= 0$, x_1.height $= 2$, where x_1 has key 1 and x_0 has key 0.

for all valid j. Because the linked lists are doubly linked, $x.\text{left}[j] = y$, if and only if $y.\text{right}[j] = x$ for all nodes x, y. For example, in Fig. 1.1, we have $x_1.\text{right}[2] = x_4$ and $x_4.\text{left}[1] = x_2$. Note that $x_{-\infty}.\text{right}[H] = x_\infty$ and $x_\infty.\text{left}[H] = x_{-\infty}$.

Given a node x, we define the **rank** of x in linked list $\mathcal{L}_h$ as the *position* of x in $\mathcal{L}_h$. Because each linked list in $\mathcal{L}$ contains nodes in ascending order of key, the rank of a node in $\mathcal{L}_j$ is just the position of its key in a sorted order of all the keys in $\mathcal{L}_j$. For the same reason, we require the following for all $0 \le i \le n$ and all $0 \le \ell \le x_i$.height:

$$x_i.\text{right}[\ell] = x_b,$$
$$x_i.\text{left}[\ell] = x_a$$

where $b = \min_j\{j \mid j > i \text{ and } x_j \in \mathcal{L}_\ell\}$ is the smallest rank larger than i in $\mathcal{L}_\ell$, and $a = \max_j\{j \mid j < i \text{ and } x_j \in \mathcal{L}_\ell\}$ is the largest rank smaller than i in $\mathcal{L}_\ell$.

1.1.2 Search

Searching for a key in a skip list $\mathcal{L}$ proceeds in a top-down fashion. The higher level linked lists facilitate search by acting as "express lanes" to the queried key at lower levels. Traversing the linked list at level ℓ, realizing that the current node has key greater than the target node, and then dropping to search level $\ell - 1$ would mean that only a sub-list of level $\ell - 1$ needs to be searched (Fig. 1.2). If a queried key k is not found, the search returns the node with greatest key less than k. A detailed description of the search algorithm appears in Algorithm 1.

Considering a search path in reverse gives the fact that the expected number of nodes traversed per linked list is at most 2. Since there are expected $O(\log n)$ levels, the expected search time is $O(\log n)$ [1].

1.1.3 Insert

The actual linked list structure of a skip list $\mathcal{L}$ is determined probabilistically. Inserting a node x in a skip list $\mathcal{L}$ proceeds as follows. First, we execute a search for x.key in $\mathcal{L}$, which will fail and instead return the node y with the largest key less than

```
Function search($\mathcal{L}$, searchKey):
    $node \leftarrow \mathcal{L}$.HEAD
    $\ell \leftarrow H$
    while $\ell > 0$ do
        while $node$.right[$\ell$].key $\leq$ searchKey do
            $node \leftarrow node$.right[$\ell$]
        if $node$.key $=$ searchKey then
            return $node$
        $\ell \leftarrow \ell - 1$
    return $node$
```

Algorithm 1: Search algorithm for skip list $\mathcal{L}$ with height H. searchKey is the queried key.

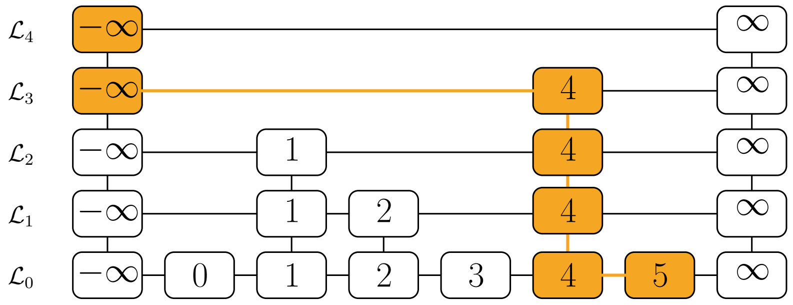


Figure 1.2: Skip list from Fig. 1.1 searches for queried key 5, starting at HEAD ($-\infty$).

x.key. We then relink $\mathcal{L}_0$ so that x is between y and y.right[0]. Next, we repeatedly flip a fair coin until it first turns up tails – let the number of heads be t. Generally, we will stop flipping the coin once t exceeds the height H of $\mathcal{L}$, so $t \leq H + 1$. Then x is inserted into all $\mathcal{L}_\ell$, for $0 \leq \ell \leq t$, so x.height $= t$. If $t = H + 1$, we initialize a new linked list $\mathcal{L}_t$ and then increment H by 1. The sentinel nodes $x_{-\infty}, x_\infty$ are inserted into $\mathcal{L}_t$ along with x.

Since insertion proceeds in this way, on average half the nodes from $\mathcal{L}_\ell$ will appear in $\mathcal{L}_{\ell+1}$. Thus the number of nodes at level ℓ is expected to be $|\mathcal{L}_\ell| = 2 + n/2^\ell$, with the exception of $|\mathcal{L}_H| = 2$. This indicates that H, the height of the skip list, is expected to be $O(\log n)$.

1.2 Skip Graphs

1.2.1 Structure

Aspnes and Shah [1] invented the **skip graph**, a distributed data structure and overlay network that is inspired by the skip list. Being a distributed data structure, it supports point-to-point search requests, insertion, and deletion in a highly distributed system. Furthermore, it allows each of these operations to be done in expected logarithmic time.

Very generally, a skip graph maintains a hierarchy of levels where each level contains several linked lists. Each of the n nodes appears in exactly one linked list per level, until the final level where every node appears in a length 1 linked list. The linked list at level ℓ that a node x belongs to is determined by the first ℓ symbols of its **membership vector**, which is a random word over the alphabet $\{0, 1\}$. Level 0 comprises of a single linked list that contains all n nodes in ascending order of key, and in general, each non-singleton linked list at level ℓ is a superset of two linked lists at level $\ell + 1$.

Formal Description: A **skip graph** is an ordered collection of n nodes $\mathcal{G} = \{x_1, ..., x_n\}$ where each node x_i has an integer key x_i.key $= k_i$, with $k_1 < \cdots < k_n$ as usual. The rank of a node x in $\mathcal{G}$ is its position in the level 0 list – that is, the rank of x_i is i. Each node x additionally maintains the following fields, which dictates the structure of $\mathcal{G}$:

$$\begin{aligned} x.\text{height} &= h \in \mathbb{N} \\ x.\text{id} &\in \{0,1\}^h \\ x.\text{key} &\in \mathbb{N} \\ x.\text{left}&[0, ..., h] \\ x.\text{right}&[0, ..., h]. \end{aligned}$$

The height h of node x is the number of linked lists in $\mathcal{G}$ that x appears in. The **identifier** of node x, x.id, is a simplified version of what Aspnes and Shah [1] call the membership vector. We write $w \mid \ell$ to be the first ℓ bits of a word w, and we let

ϵ be the empty word. Here, x.id is the first h bits of x's membership vector, as that is all that needs to be maintained. In particular, let the **height** of the skip graph $\mathcal{G}$ be $H = \max_{x \in \mathcal{G}}\{x.\text{height}\}$, that is, the maximum height over all nodes in $\mathcal{G}$.

- For each word $w \in \bigcup_{i=0}^{H}\{0,1\}^i$, let G_w be the doubly linked list associated with w.
- G_w belongs in level ℓ of $\mathcal{G}$ if and only if $|w| = \ell$, where $|w|$ is the length of w.
- For a node $x \in \mathcal{G}$, $x \in G_w$ if and only if x.id is a prefix of w, written $w \preceq x.\text{id}$
- G_w is sorted in ascending order of key.
- For every pair of nodes x, y in $\mathcal{G}$, we require that $x.\text{id} \neq y.\text{id}$, and that if $x.\text{height} \leq y.\text{height}$, then x.id is not a prefix of y.id. This is to ensure that every node is eventually contained in a linked list of length one.

This ruleset also guarantees that each node x appears in at most one linked list per level, as $x.\text{id}|i$ returns a single prefix p of length i for all $i \leq |x.\text{id}|$, determining that x goes into G_p. Consequently, G_ϵ denotes the level 0 linked list which contains all n nodes. We will consider $\mathcal{G}$ to only contain the linked lists G_w that are nonempty. For convenience, we will also write $\mathcal{G} = \{G_w\}$ so it is clear as to which variable (in this case G) is used to index linked lists.

Just as in our formulation of skip lists, each node x maintains two arrays x.left and x.right for their left and right neighbors at each level, respectively. Specifically, $x.\text{left}[j]$ is the node to the left of x in linked list $G_{x.\text{id}|j}$ and $x.\text{right}[j]$ is the node to the right. If x is the first or last node in a linked list, then its left or right neighbor in that list will be NULL, respectively. And since we require that each G_w is sorted in ascending order based on key, it must be that for each node x_i,

$$x_i.\text{left}[\ell] = x_a$$
$$x_i.\text{right}[\ell] = x_b$$

where $a = \max_j\{j < i \ : \ x_j.\text{id}|\ell = x_i.\text{id}|\ell\}$ and $b = \min_j\{j > i \ : \ x_j.\text{id}|\ell = x_i.\text{id}|\ell\}$. That is, a is the smallest rank greater than i in $G_{z|\ell}$ and b is the largest rank smaller than i in $G_{z|\ell}$, where $z = x$.id. See Fig. 1.3 for example of a skip graph.

Since links are bidirectional, a skip graph determines an undirected graph (i.e. a network topology) in the traditional sense, where an edge between nodes x and y is drawn if they are neighbors in some G_w in $\mathcal{G}$ (Fig. 1.4). The skip graph is simply a way of representing this information, so in general we will use $\mathcal{G}$ to refer to both the skip graph representation and the network topology it represents, since they are equivalent.

Prefix tree representation: A skip graph $\mathcal{G}$ can also be represented as a binary prefix tree. G_ϵ roots the tree, and every G_w has two children G_{w0} and G_{w1} (Fig. 1.5). The level of a linked list is then equal to its tree depth, and each of the n nodes in the system will be ultimately reside in a leaf, which corresponds to a linked list of length one.

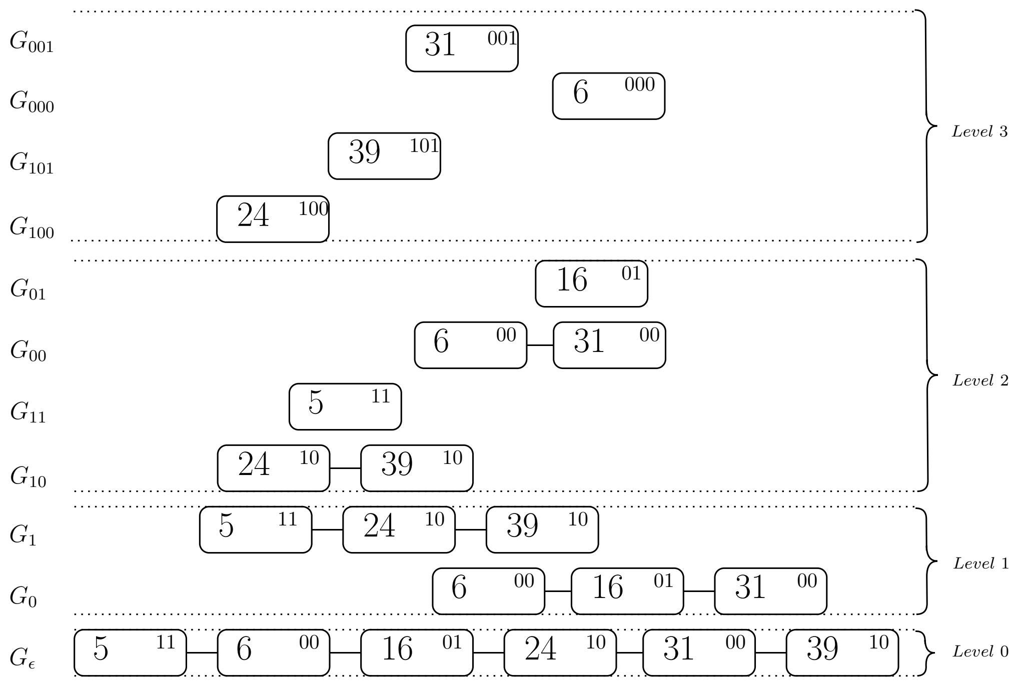


Figure 1.3: Example skip graph $\mathcal{G}$ on $n = 6$ nodes. Identifiers are at the top right of each node. Note that the height of $\mathcal{G}$ is 3, and that every node appears in a singleton linked list.

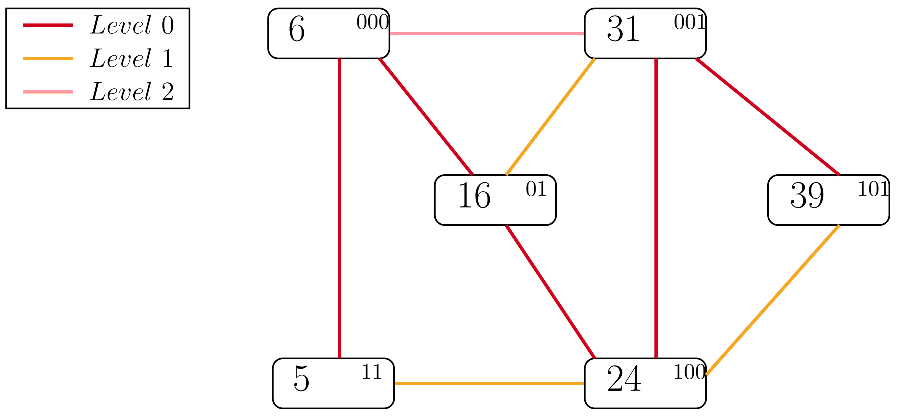


Figure 1.4: The skip graph in Fig. 1.3 determines a graph. Edge colors only distinguish the level in which the connected nodes first appear as neighbors, but have no other function otherwise.

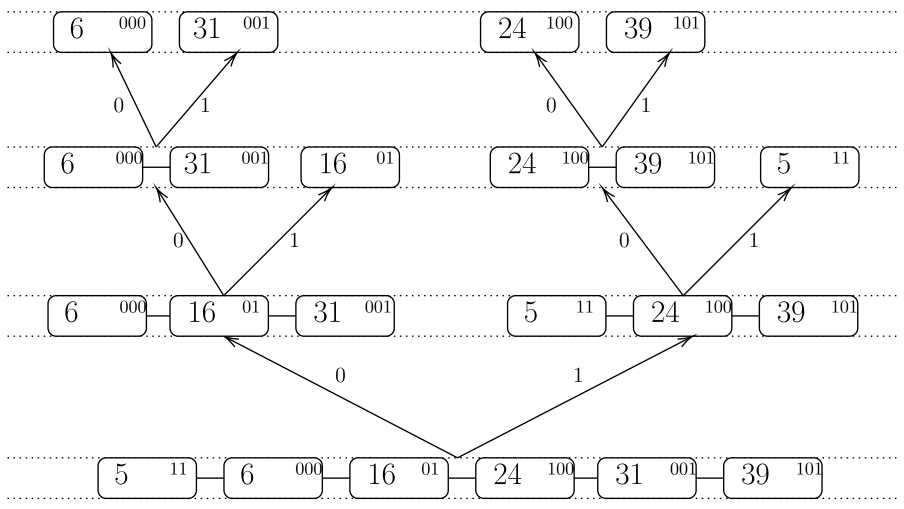


Figure 1.5: Skip graph from Fig. 1.3 represented as a binary prefix tree.

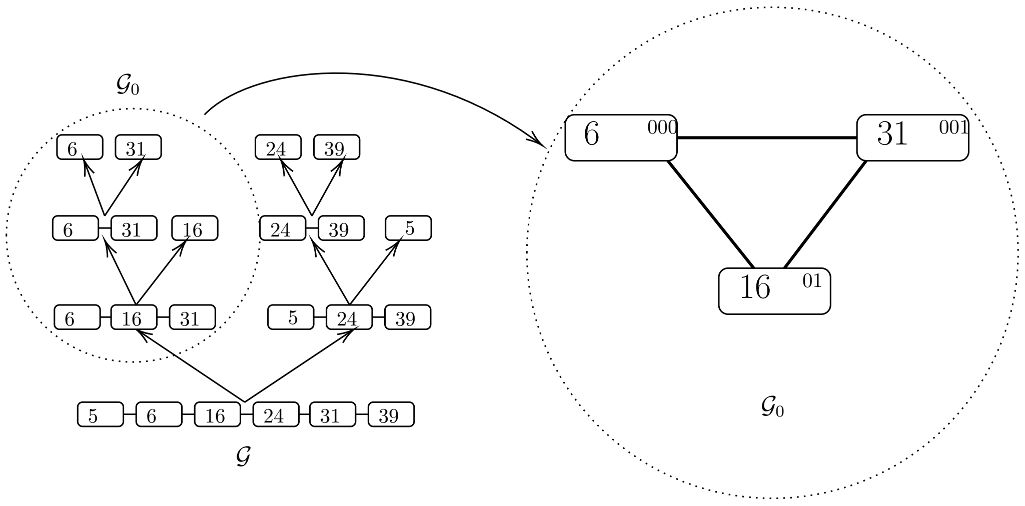


Figure 1.6: Sub-skip graph $\mathcal{G}_0$ of skip graph $\mathcal{G}$.

Sub-skip graphs: In Fig. 1.5, note that each child list fully determines another skip graph (a sub-skip graph) rooted at that child. We will write $\mathcal{G}_w$ to denote the sub-skip graph that only contains the nodes in G_w. The network that $\mathcal{G}_w$ represents is simply the network that $\mathcal{G}$ represents but only on the nodes in linked list G_w. For example, in Fig. 1.3 and Fig. 1.5, $\mathcal{G}_0 = \{x_6, x_{16}, x_{31}\}$ where here we use x_i to be the node with key i. See Fig. 1.6 for an example.

It is important to keep in mind that a skip graph is still a graph in the sense of Fig. 1.4; the prefix-tree (Fig. 1.5) and linked list hierarchy (Fig. 1.3) representations simply present that information in more useful ways.

View as a collection of skip lists: With the prefix tree representation in mind (Fig. 1.5), notice that tracing a path from any list to any other sublist fully determines a skip list. A skip graph can thus be thought of as a collection of skip lists that share

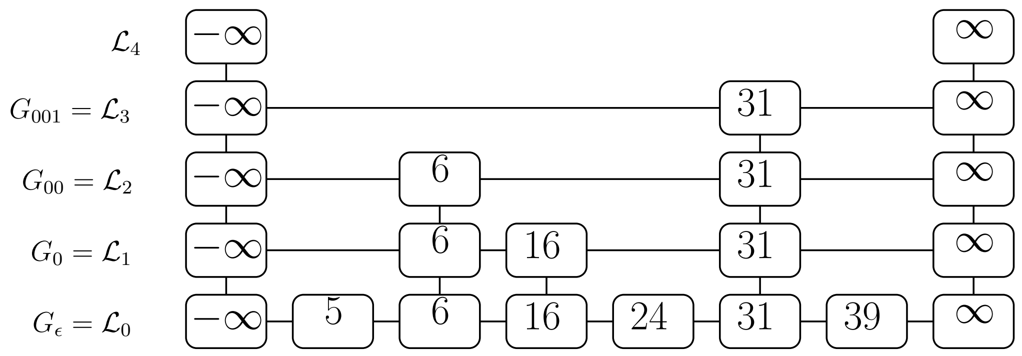


Figure 1.7: A skip list obtained from skip graph in Fig. 1.3 by grouping together all linked lists that contain key 31.

their lower levels. See Fig. 1.7 for an example.

In fact, this observation leads to the following result [1]:

Theorem 1.2.1. Let $\mathcal{G}$ be a skip graph with height H. For all $z \in \{0,1\}^H$ such that G_z is nonempty, the linked lists $G_{z|0}, G_{z|1}, G_{z|2}, G_{z|3}, ...$ form a skip list.

Proof. We summarize the proof from [1], which performs induction on i, the prefix length of each $G_{z|i}$. For $i = 0$, $G_{z|0} = G_\epsilon$ is the bottom-most list containing all nodes. A node x appears in $G_{z|i}$ if and only if $x.\text{id}|i = z|i$. Since there are 2 possibilities for symbol $i+1$, $\Pr[x \in G_{z|(i+1)} \mid x \in G_{z|i}] = 1/2$. Each x appears in $G_{z|(i+1)}$ independently of any other $x' \in G_{z|i}$, so $G_{z|0}, G_{z|1}, ...$ forms a skip list. □

We will refer to $\mathcal{L}_z = G_{z|0}, G_{z|1}, G_{z|2}, G_{z|3}, ..., G_{z|z.\text{height}}$ as the **skip list restriction** of node x with $x.\text{id} = z$. Since each child list determines a sub-skip graph, Theorem 1.2.1 easily extends to $G_{z|i}, ..., G_{z|j}$ for any $0 \leq i \leq j$. This perspective also sheds light on each node's view of the rest of the skip graph. Each node x with id w will belong in a leaf list in $\mathcal{G}$, which can be thought of as the node at the top level in its own skip list G_w (e.g. the node holding 5 is the top node of G_{11} in Fig. 1.5). This gives a more localized view of a skip graph, as each node x only needs to store identifiers to their neighbors in each level of $\mathcal{G}$.

1.2.2 Balanced Skip Graphs

We quickly define a special type of skip graph that will be useful.

Definition 1.2.1. A **balanced skip graph** $\mathcal{B}$ on $n = 2^H$ nodes is a skip graph where every node has a unique identifier of length exactly H.

A balanced skip graph on n nodes defines a bijection between the set of H-length bitstrings and the n nodes. This ensures that every balanced skip graph has height $H = \log n$, the minimum height across all skip graphs on n nodes.

However, the term "balanced" only refers to the fact that the skip graph has height $\log n$, which does not correlate with how effective and connected its network topology is (Example 1.2.1).

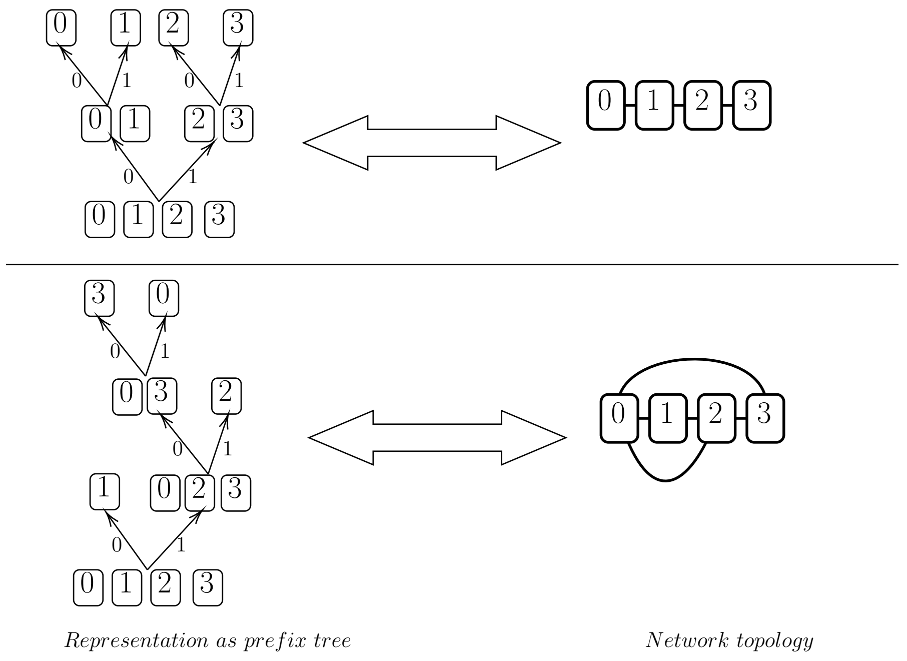


Figure 1.8: Balanced skip graph $\mathcal{B}$ (top) and skip graph $\mathcal{G}$ (bottom).

Example 1.2.1. Suppose we have a balanced skip graph $\mathcal{B}$ and a skip graph $\mathcal{G}$ both on four nodes, as in Fig. 1.8. Note that $\mathcal{G}$ determines a more connected network topology than $\mathcal{B}$ even though $\mathcal{G}$ has larger height than $\mathcal{B}$. In fact, because the network determined by $\mathcal{G}$ is just the network determined by $\mathcal{B}$ with extra edges, the cost to search from u to v (see Section 1.2.3) for any u, v is less than or equal to the cost to search from u to v in $\mathcal{B}$.

1.2.3 The `search` algorithm

We refer the reader to the original paper [1] for a more meticulous treatment of the skip graph routing algorithm, `search` [1]. In particular, they provide proofs of correctness and their pseudocode is presented to run in a distributed system. We present our pseudocode at a higher level by abstracting away the message-passing protocols.

Because a skip graph $\mathcal{G}$ describes an overlay network, we are interested in serving point-to-point search requests. Given a node pair (u, v) such that $u \in \mathcal{G}$, we want to search from u to v. More specifically, a node u can initiate a search for a key k, which then returns the node v with key k if found. If such a v is not found, then the search should return the node with the largest key less than k (if u.key $< k$) or the node with the smallest key greater than k (if u.key $> k$). Node u performs what is essentially a skip list search starting at u in its skip list restriction $\mathcal{L}_w$, where $w = u$.id. Because u initiates the search and not x_∞, the search can attempt to follow u's left pointer if

[1]They also discuss the skip graph's ability to perform efficient range queries. A range query is a powerful operation that returns some set of nodes whose keys lie in a specified range.

k is less than u.key. Note that once a direction is picked (following u's left or right pointer), the search will never traverse a pointer in the opposite direction. A detailed description of the algorithm appears in Algorithm 2.

```
Function search($\mathcal{G}$, fromNode, searchKey):
    $u \leftarrow$ fromNode
    $\ell \leftarrow |u.\text{id}|$
    while $\ell > 0$ do
        while $u$.right[$\ell$].key $\leq$ searchKey do
            $u \leftarrow u$.right[$\ell$]
        while $u$.left[$\ell$].key $\geq$ searchKey do
            $u \leftarrow u$.left[$\ell$]
        if $u$.key = searchKey then
            return $u$
        $\ell \leftarrow \ell - 1$
    return $u$
```

Algorithm 2: Skip graph search initiated by fromNode. searchKey is the queried key.

Theorem 1.2.2. In a skip graph $\mathcal{G}$ with n nodes, search takes $O(\log n)$ time.

Proof. A search initiated by u ($w = u$.id) at height h in $\mathcal{G}$ will follow the same search path as a search in the skip list formed by $G_{w|h}, ..., G_{w|1}, G_\epsilon$ if the search started at u and not $x_{-\infty}$. (Theorem 1.2.1). Because this is just a skip list search, it takes expected $O(\log n)$ time, as we showed in Section 1.1. □

Remark 1.2.1. Search is not optimal. Because the search algorithm routes greedily, it does not always follow the shortest path between u and v. To see this, consider the example in Fig. 1.9. When 0 searches for 3 using the skip graph search algorithm, it will traverse the path $0, 1, 2, 3$. However, the shortest path between 0 and 3 in the graph is $0, 4, 3$.

Remark 1.2.2. Search is not commutative. It is not always the case that `search`(u, v) = `search`(v, u). Again, consider the skip graph in Fig. 1.9. Here `search`$(3, 0) = 2$ since, starting at 3, the search path traverses 1 and then 0. However, `search`$(0, 3) = 3$ since, starting at 0, the search path traverses 1, 2, and then 3.

Remark 1.2.3. Going Forward. To greatly simplify matters, we will refer to both nodes and keys as their rank in the skip graph. In other words, nodes and keys will live in the range $\{0, ..., n-1\}$.

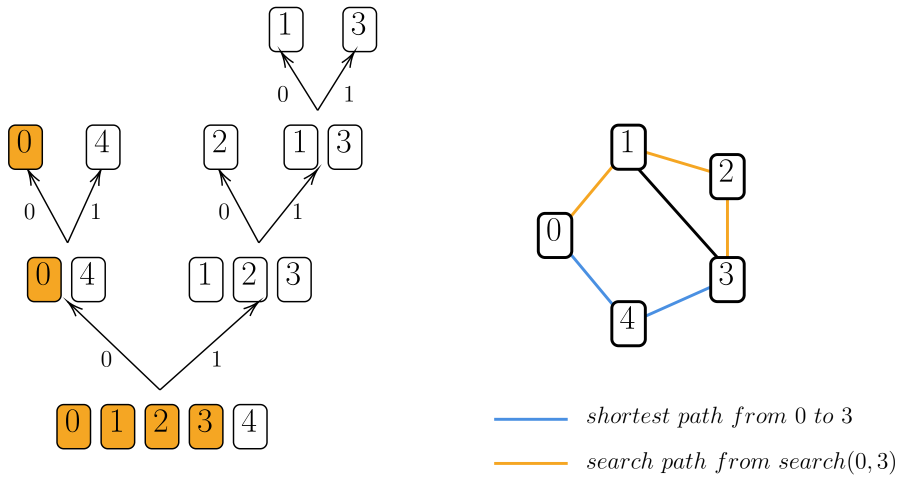


Figure 1.9: The skip graph search path is not always the shortest path. Search is also not commutative. Left: Prefix tree representation. Right: View as a network.

Chapter 2

The Minimum Expected Path Length Problem

In this chapter, we establish some useful measures for network efficiency – average path length, weighted path length, and expected path length. Then we explore some NP-Complete aspects to the minimum expected path length problem when restricted to skip graphs.

2.1 Preliminaries

Given a graph $G = (V, E)$, we first define the **average path length** of G to be the average length of the shortest paths between every node pair. More formally:

Definition 2.1.1. The **average path length** of a graph $G = (V, E)$ is

$$\mathrm{APL}(G) = \frac{1}{|V|^2} \sum_{(u,v) \in V \times V} d_G(u, v) \tag{2.1}$$

where $d_G(u, v)$ the cost to route from u to v in G, which we assume is the shortest path unless specified otherwise.

Example 2.1.1. Computing the average path length for K_n, the complete graph on n nodes, is simply $\mathrm{APL}(K_n) = \frac{1}{n^2} \sum\limits_{(u,v) \in V \times V} 1 = \frac{n^2}{n^2} = 1$. This matches our intuition – because every node pair is connected by an edge in K_n, the average path length should be one.

The average path length describes the average number of edges traversed when routing between two uniformly randomly chosen nodes in a network. Lower average path length corresponds to lower overall route lengths, which makes it a useful measure for network efficiency.

However, if we cannot assume that all node pairs are equally likely to communicate, then the average path length may not be a revealing measure of a network's efficiency. Some nodes may communicate with other nodes more frequently and thus

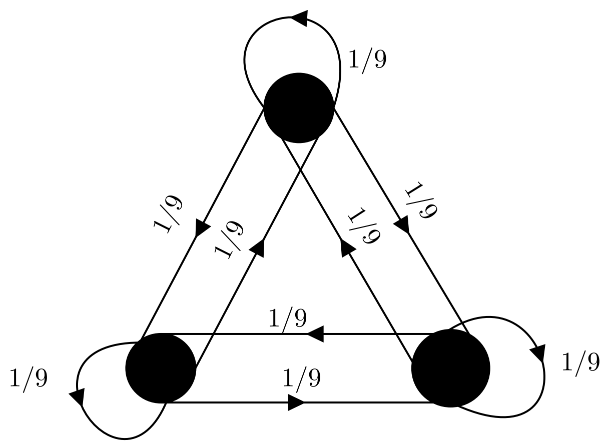


Figure 2.1: Example demand graph on 3 nodes modeling the uniform request distribution.

some paths in the network may be more frequently traversed than others. This suggests that in Definition 2.1.1, the contribution of the term $d_G(u, v)$ to the total sum should be proportional to the frequency that u communicates with v. So instead, if we are given the distribution that node pairs are sampled from (also called the request distribution or **demand**), we can define a similar measure called the **weighted path length**:

Definition 2.1.2. The **weighted path length** of a graph $G = (V, E)$ given a demand $D : V \times V \to [0, 1]$, $\sum\limits_{(u,v)\in V\times V} D(u, v) = 1$, is

$$\mathrm{WPL}(D, G) = \sum_{(u,v)\in V\times V} D(u, v) d_G(u, v). \tag{2.2}$$

We will model the demand on node pairs of $V = \{0, ..., n-1\}$ as a **demand graph** (Fig. 2.1), which we define to be a complete directed graph (with loops) on V where each edge (u, v) is weighted by $D(u, v)$, the probability of u requesting v, or equivalently, the *demand* between u and v.[1] Because we allow loops, we allow distributions where a node can communicate with itself with positive probability.[2] Edges with zero weight will be omitted. We will let D denote both demands and demand graphs interchangeably, with $D(u, v)$ being the probability that u requests v in both cases.

Example 2.1.2. Given $G = K_n$ and $D(u, v) = 1/n^2$ for all (u, v) (uniform demand), we compute $\mathrm{WPL}(D, G) = \sum_{(u,v)\in V\times V} \frac{1}{n^2} \cdot 1 = \frac{n^2}{n^2} = 1$. This matches our result in

[1]Alternatively, a demand could be described by an $n \times n$ matrix M, where entry $M_{u,v} = D(u, v)$.

[2]This may sound strange, but there are applications where allowing a node to request itself is not unreasonable. If each node stands for a cluster of computers, loops and non-loops represent intra-cluster and inter-cluster communication, respectively.

Example 2.1.1. If the demand describes the uniform distribution, then the weighted path length of G is precisely the average path length of G, regardless of G.

We can generalize Definition 2.1.2 and introduce what we call an **embedding** (also called a labeling or placement function [2]).

Definition 2.1.3. Given a demand graph $D = (V_D, E_D)$ that defines a request distribution and an undirected, unweighted graph $G = (V_G, E_G)$ that describes a network topology, an **embedding** $\phi : V_D \to V_G$ is a bijection that maps the nodes of D onto the nodes of G.

From the network perspective, the idea behind an embedding is to "assign" to each node in the network a particular set of communication partners and probabilities of communicating with those partners, as determined by the demand graph. This is typically done to minimize some objective, as we do shortly. The complementary way of looking at this is from the demand graph perspective, where we can think of the nodes in the demand graph D as "processes" that communicate with each other. Then the embedding function determines the location of each process in the network topology determined by G. Now, we define the **expected path length**:

Definition 2.1.4. Given a demand graph $D = (V_D, E_D)$ that describes a probability distribution on node pairs, an undirected, unweighted graph $G = (V_G, E_G)$ that describes a network topology, and an embedding $\phi : V_D \to V_G$, the **expected path length** is

$$\mathrm{EPL}(D, G, \phi) = \sum_{(u,v)\in E_D} D(u, v) d_G(\phi(u), \phi(v)). \tag{2.3}$$

Example 2.1.3. Consider the embedding ϕ from D to path graph G in Fig. 2.2. Specifically, $\phi(w) = a, \phi(y) = b, \phi(x) = c, \phi(z) = d$. From Definition 2.1.4, we can calculate

$$\begin{aligned}\mathrm{EPL}(D, G, \phi) &= \frac{1}{4} d_G(\phi(w), \phi(z)) + \frac{1}{4} d_G(\phi(w), \phi(y)) \\ &+ \frac{1}{6} d_G(\phi(y), \phi(x)) + \frac{1}{3} d_G(\phi(x), \phi(w)) \\ &= (1/4)(3) + (1/4)(1) + (1/6)(1) + (1/3)(2) \\ &= 1.5.\end{aligned}$$

.

Note that the definition of weighted path length (Definition 2.1.2) implicitly assumes knowledge of the embedding – the processes have already been placed, and their corresponding locations in the network are known. The only difference in the definition of expected path length (Definition 2.1.4) is that we make the embedding explicitly a parameter itself. Further note that even if D is the uniform distribution in Definition 2.1.4, the expected path length is still precisely the average path length of G, regardless of G and ϕ.

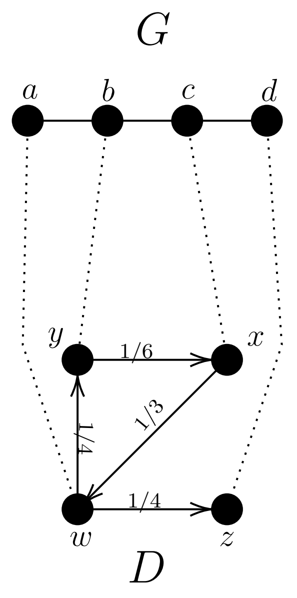


Figure 2.2: An embedding ϕ from the demand graph D to the path graph G.

2.2 Problem Statement

Given a demand D on the network and a network topology G, it is natural to seek a node assignment (i.e. an embedding) that minimizes the expected path length. In other words, we aim to find $\phi^* = \arg\min_\phi \mathrm{EPL}(D, G, \phi)$. The embedding ϕ^* corresponds to the most efficient way to configure the network when handed an immutable topology and request distribution. The decision variant of this problem is as follows.

Problem 2.2.1. The Minimum Expected Path Length Problem (MEPL). On input $\langle D, G, k\rangle$, where $D = (V_D, E_D)$ is a demand graph, $G = (V_G, E_G)$ is any undirected, unweighted graph ($|V_D| = |V_G|$), and k is a positive number, does there exist an embedding $\phi : V_D \to V_G$ such that $\mathrm{EPL}(D, G, \phi) \leq k$?

Chen et al. [2] show that MEPL is NP-Complete for general graphs using a reduction from k-CLIQUE. They also show that MEPL is NP-Complete in the special case where G is a 2-dimensional grid graph, using a reduction from the problem of whether a tree is embeddable in a 2-dimensional grid [5].

What about for the special case where G is a skip graph? We propose two variants of interest. First, recall Remark 1.2.1 where we showed that the shortest path between u and v in a skip graph can differ from the path traversed by the search algorithm. Thus, if $\mathcal{S}$ is a skip graph, we write $d^*_{\mathcal{S}}(u, v)$ to be the shortest path distance and $d_{\mathcal{S}}(u, v)$ to be the routing cost as determined by the skip graph's search algorithm.

Problem 2.2.2. Let **MEPL-SG*** be Problem 2.2.1 restricted to the case where G is a skip graph and where we use d^*_G to compute cost.

Problem 2.2.3. Let **MEPL-SG** be Problem 2.2.1 restricted to the case where G is a skip graph and where we use d_G to compute cost.

While MEPL-SG is more faithful to our conception of skip graphs in Section 1.2, we still pursue MEPL-SG* as it is a conceptually simpler problem.

2.3 Hardness of MEPL-SG and MEPL-SG*

2.3.1 The Minimum Linear Arrangement Problem

We first introduce the minimum linear arrangement (MLA) problem, whose decision variant is NP-Complete [10]. The MLA problem asks to find an embedding ϕ from G onto the path graph such that the cost of its linear arrangement (Eq. (2.4)) is minimized. Note that for any two nodes u, v (labeled in order of appearance) in a path graph, $|u - v|$ is the distance between them. And since G is weighted with non-negative values, normalizing the edge weights helps us think of it as a demand graph. So Eq. (2.4) mirrors Eq. (2.3), and the connection between MLA and MEPL seems quite natural. The decision variant of MLA is formalized as follows [19]:

Problem 2.3.1. The Minimum Linear Arrangement Problem (MLA). On input $\langle G, w, k\rangle$ where $G = (V, E)$ is an undirected, connected, weighted graph with $|V| = n$ and non-negative edge weight function w and k is a positive number, does there exist a bijection $\phi : V \to \{1, ..., n\}$ such that

$$\sum_{(u,v)\in E} w(u,v)|\phi(u) - \phi(v)| \leq k? \tag{2.4}$$

Example 2.3.1. Consider the star graph on n nodes with unit edge weights. The only edges in this graph are those with an endpoint at the center vertex r, and so the minimal linear arrangement is to have r map to the middle (i.e. $\phi(r) = \lfloor n/2\rfloor + 1$) and the remaining nodes can be mapped anywhere (Fig. 2.3). In the resulting linear arrangement, there are two nodes of distance 1 from r, two nodes of distance 2, up to two nodes of distance $\lfloor n/2\rfloor$ if n is odd or one node of distance $n/2$ if n is even. Using Eq. (2.4), this gives a linear arrangement cost of

$$\left(\sum_{i=0}^{\lfloor n/2\rfloor-1} 2(\lfloor n/2\rfloor - i)\right) - \lfloor n/2\rfloor(n \bmod 2) = \lfloor n/2\rfloor\left(\lfloor n/2\rfloor + 1\right) - \lfloor n/2\rfloor(n \bmod 2).$$

2.3.2 Reductions

We first proceed with a helpful lemma that allows us to perform the reduction.

Lemma 2.3.1. There exists a skip graph on n nodes that is isomorphic to a path graph on n nodes.

Proof. Let L denote the path graph on n nodes. Consider S_ϵ, the level 0 list of a skip graph $\mathcal{S}$, which contains all n nodes in a linked list. Clearly, S_ϵ forms a path, and is thus isomorphic to L. To ensure that no additional links are added to $\mathcal{S}$, we

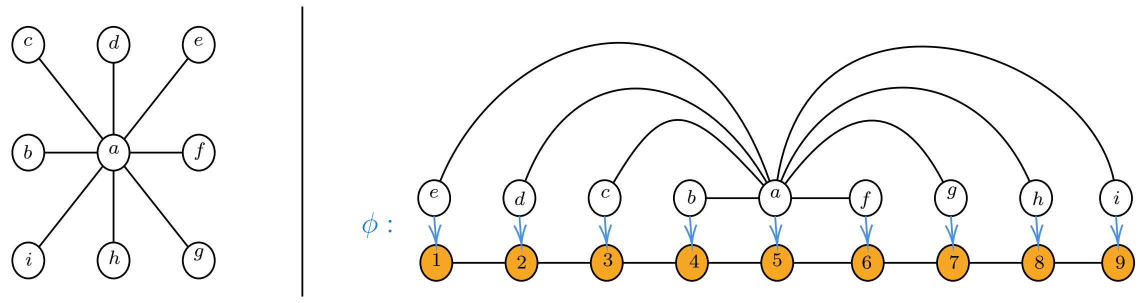


Figure 2.3: Minimum linear arrangement ϕ (right) of the star graph on 9 nodes (left), where every edge has unit weight. Note that ϕ is a bijection from the nodes of the star to $\{1, ..., 9\}$.

can promote all elements in a contiguous subset of S_ϵ to S_0 and all elements in the complement of that subset (which must also be contiguous) to S_1. Then we continue to do this recursively until all nodes are in a linked list of length 1. This ensures that every node will only be adjacent to nodes it was already adjacent to in the previous level, and thus no new links will be added. □

Theorem 2.3.2. MEPL-SG* is `NP`-Complete.

Proof. First, note that MEPL-SG* is in `NP` because given any problem instance $\langle D, G, k\rangle$ and an embedding ϕ, checking if ϕ satisfies $EPL(D, G, \phi) \leq k$ can be done in polynomial time.

Now, given any instance $I = \langle G, w, k\rangle$ of MLA, we can transform it into an instance $f(I) = \langle D, L, k'\rangle$ of MEPL-SG* as follows.

- The main idea is that D is the same as G but with 1) directed edges and 2) normalized edge weights so as to conform to a valid probability distribution. Formally, let $G = (V, E)$. Let W be the total weight of all the edges in G, so $W = \sum_{(u,v)\in E} w(u, v)$. While every $(u, v) \in E$ is an undirected edge, we can interpret E as a directed edge set by randomly choosing either u or v to be the direction. Then $D = (V, E)$, but with edge weights $w(u, v)/W$ for every edge $(u, v) \in E$.[3]

- L is the path graph on $|V|$ nodes, which is also a skip graph due to Lemma 2.3.1.

- $k' = k/W$.

Now, we show that MLA accepts $I \iff$ MEPL-SG* accepts $f(I)$.
$(\implies)$ Suppose MLA accepts I. Then there exists a bijection $\phi : V \to \{1, ..., |V|\}$

[3]Note that we can assume that $W > 0$. The case where $W = 0$ means that every edge has 0 weight, which is trivially solvable in polynomial time because Eq. (2.4) is always satisfied. Thus the problem must be hard when $W > 0$.

such that

$$\sum_{(u,v)\in E} w(u,v)|\phi(u)-\phi(v)| \leq k.$$

Now since $L = \{1, ..., |V|\}$,

$$\begin{aligned}
&\implies \sum_{(u,v)\in E} w(u,v) d_L^*(\phi(u),\phi(v)) \leq k \\
&\implies \frac{1}{W}\sum_{(u,v)\in E} w(u,v) d_L^*(\phi(u),\phi(v)) \leq \frac{k}{W} \\
&\implies \sum_{(u,v)\in E} \frac{w(u,v)}{W} d_L^*(\phi(u),\phi(v)) \leq \frac{k}{W} \\
&\implies \mathrm{EPL}(D,L,\phi) \leq \frac{k}{W} = k',
\end{aligned}$$

so MEPL-SG* accepts $f(I)$.
($\Longleftarrow$) Suppose MEPL-SG* accepts $f(I)$. Then there exists a bijection $\phi : V \to \{1, ..., |V|\}$ such that

$$\begin{aligned}
&\mathrm{EPL}(D,L,\phi) \leq k' \\
&\implies \sum_{(u,v)\in E} \frac{w(u,v)}{W} d_L^*(\phi(u),\phi(v)) \leq \frac{k}{W} \\
&\implies \frac{1}{W}\sum_{(u,v)\in E} w(u,v) d_L^*(\phi(u),\phi(v)) \leq \frac{k}{W} \\
&\implies \sum_{(u,v)\in E} w(u,v)|\phi(u)-\phi(v)| \leq k.
\end{aligned}$$

And so MLA accepts I.

□

Corollary 2.3.3. MEPL-SG is NP-Complete.

Proof. The proof is identical to Theorem 2.3.2 because the route traced by the skip graph search algorithm is identical to the shortest path when the skip graph is a path. In other words, $d_L = d_L^*$, so we can substitute d_L^* with d_L in the proof above. □

2.3.3 Variants

Remark 2.3.1. Bipartite demand is NP-Complete. The MLA problem remains NP-Complete when the input graph G is restricted to bipartite graphs [7]. Therefore the same reduction in Theorem 2.3.2 can be used to show that the MEPL problem remains NP-Complete when the demand graph is bipartite.

Remark 2.3.2. Extension to other graph families. The reduction in Theorem 2.3.2 can also be used to show that the general MEPL (Problem 2.2.1), when G

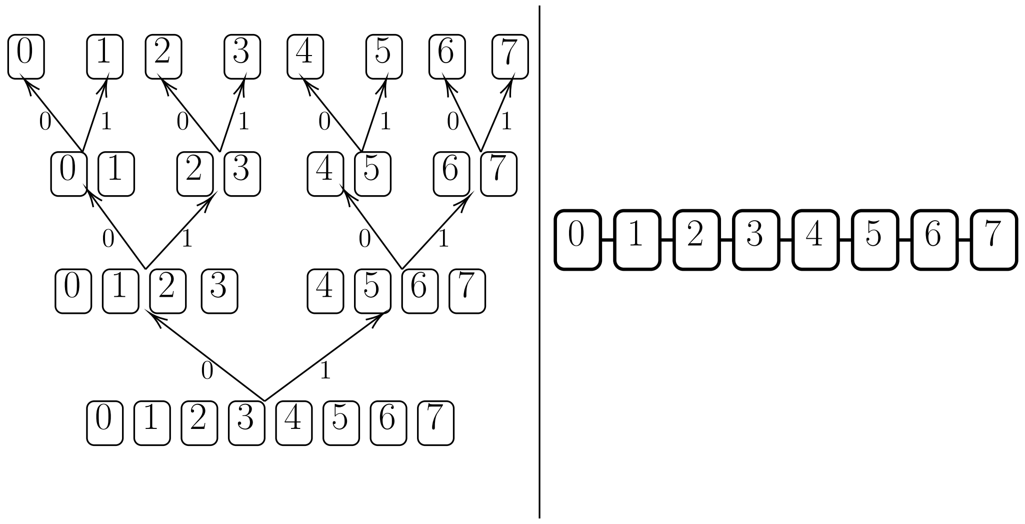


Figure 2.4: Balanced skip graph construction in proof of Corollary 2.3.4 (left) produces a path graph, for $n = 2^3 = 8$.

is a restricted family of graphs, remains NP-Complete if the path is a special case of the graph family. It thus follows that MEPL remains NP-Complete when G is restricted to trees, bounded-degree graphs, and bipartite graphs, for instance. Presumably, this relationship can in turn be used to show that several other graph families remain NP-Complete under the MEPL problem.

For example, one application of Remark 2.3.2 is to show NP-Completeness even if we restrict ourselves to a smaller family of skip graphs.

Problem 2.3.2. Let **MEPL-SG-BAL*** be Problem 2.2.1 restricted to the case where G is a balanced skip graph on $n = 2^k$ nodes, and we use d_G^* to compute cost. Similarly define **MEPL-SG-BAL** to be the same problem but where we use d_G to compute cost.

Corollary 2.3.4. MEPL-SG-BAL* and MEPL-SG-BAL are both NP-Complete.

Proof. It suffices to show that there exists a balanced skip graph on $n = 2^k$ nodes that is isomorphic to the path graph on n nodes (Remark 2.3.2). To show this, recall the proof of Lemma 2.3.1. Starting from S_ϵ which contains all n nodes in increasing order $\{0, ..., n-1\}$, promote all elements in the contiguous subset $\{0, ..., n/2-1\}$ to S_0 and the complement $\{n/2, ..., n-1\}$ to S_1. Doing this recursively produces a path graph (Fig. 2.4). Because we partition in exactly half each step, the recursion has depth $\log_2(n) = k$, so every node has an identifier of length k. $\square$

Chapter 3

The Optimal Skip Graph Problem

In this chapter, we first formulate the problem of constructing optimal skip graphs for a given demand. We then enumerate skip graphs to underscore the difficulty of the problem, and then turn our attention to the interleaved skip graph, where we discuss its resemblance to Chord and derive a closed form for its average path length. Finally, we conjecture that the interleaved skip graph has optimal average path length.

3.1 Problem Statement

Another natural question is: given a node set V and demand graph $D = (V, E_D)$, can a skip graph $\mathcal{G}^* = (V, E_{\mathcal{G}^*})$ such that the weighted path length is minimized be found? More formally, stated as an optimization problem:

Problem 3.1.1. Optimal Skip Graph Problem (OSG). Given a demand D over the node pairs of V, find a skip graph $\mathcal{G}^* = (V, E)$ such that

$$\mathcal{G}^* = \arg\min_{\mathcal{G}} \mathrm{WPL}(D, \mathcal{G}), \tag{3.1}$$

where we use the skip graph routes $d_{\mathcal{G}}$ to compute cost within WPL.

The difference between OSG and MEPL (Problem 2.2.1) is that in OSG, the network topology itself is subject to optimization. Whether this freedom makes OSG an easier or harder problem than MEPL for certain demands and network topologies is an open question [4]. If we are optimizing over binary tree networks rather than skip graphs in Eq. (3.1), then the optimal network can be computed in polynomial time using dynamic programming [18]. On the other extreme, when we are optimizing over general trees and D is restricted to the uniform distribution, the problem is `NP`-Complete [14].

3.2 Enumerating Skip Graphs

Here we enumerate the number of skip graphs on n nodes, where for this section we define a skip graph as a valid prefix tree representation and count the number

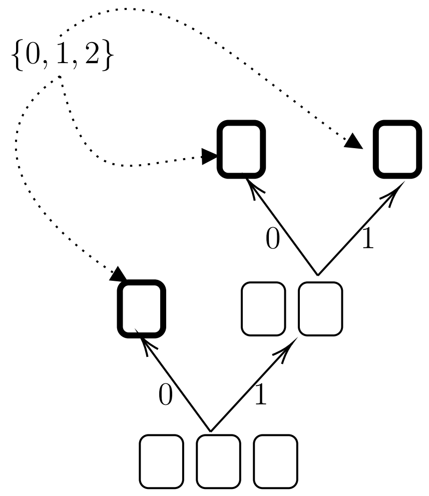


Figure 3.1: An example prefix tree layout for $n = 3$. Each identifier is determined by a path from the root of the prefix tree to a leaf. Because the number of leaves in a prefix tree representation is the number of nodes in the skip graph, there are $n!$ ways to assign the nodes to the leaves.

of such prefix trees [1] We can decouple the enumeration of prefix tree layouts from the assignment of nodes to identifiers by observing that for each prefix tree layout, there are $n!$ ways to assign the identifiers determined by the layout to the n nodes (Fig. 3.1).

The problem becomes one of enumerating the number of prefix tree layouts, which we can characterize recursively. Let T_n be the number of prefix tree layouts for n nodes (e.g Fig. 3.1 is one such layout for $n = 3$ nodes). Then the number of skip graphs on n nodes is $n!T_n$, as discussed. To compute T_n, we have

$$T_n = \sum_{i=1}^{n-1} T_{n-i}T_i \qquad T_1 = 1 \tag{3.2}$$

because when we choose to allocate i nodes for the left subtree, we automatically allocate $n - i$ nodes for the right subtree, where i can range from 1 to $n - 1$. We multiply T_i by T_{n-i} because for each left subtree T_i, we can pair with T_{n-i} different choices for the right subtree. Now, notice that $T_n = \mathcal{C}_{n-1}$, where $\mathcal{C}_k$ is the k^{th} Catalan number given by the recurrence

$$\mathcal{C}_k = \sum_{i=0}^{k} \mathcal{C}_{k-i}\mathcal{C}_i, \quad \mathcal{C}_0 = 1 \tag{3.3}$$

and has closed form $\frac{(2k)!}{(k+1)!k!}$. This gives $T_n = \frac{(2n-2)!}{n!(n-1)!}$, and so the number of skip

[1]There may be several different prefix tree representations for a given network topology (e.g. the path graph, as seen in Lemma 2.3.1). So two skip graphs that differ in prefix tree representation may otherwise have isomorphic network topologies.

graphs on n nodes is

$$n!T_n = \frac{(2n-2)!}{(n-1)!} \tag{3.4}$$

Depending on the implementation, an exhaustive search even for small n ($n < 20$) may be prohibitive. It remains an interesting question to enumerate the number of non-isomorphic skip graphs. For example, in Eq. (3.2), we could iterate the sum from $i = 1$ to $\lfloor n/2 \rfloor$ to account for the fact that two skip graphs that have mirrored prefix tree representations are isomorphic – but this only accounts for isomorphisms among skip graphs with identical prefix tree layouts.

3.3 Interleaved Skip Graphs

In this section, we construct the interleaved skip graph, which is a highly connected skip graph that we conjecture to have optimal (minimal) average path length.

Definition 3.3.1. The **interleaved skip graph** on n nodes is a skip graph $\mathcal{I} = \{I_w\}$ created by the following procedure:

1. Initialize $I_\epsilon = [0, ..., n-1]$ and let $w = \epsilon$.
2. Promote every odd indexed element in I_w to I_{w1} and every even indexed element to I_{w0}.
3. Do this recursively for every w until I_w comprises of a single node.

Note that when n is a power of two, the interleaved skip graph is a balanced skip graph.

Example 3.3.1. Observe the interleaved skip graph on 8 nodes in Fig. 3.2. Starting with I_ϵ, we promote $0, 2, 4, 6$ to I_0 and $1, 3, 5, 7$ to I_1. Then we promote $0, 4$ to I_{00}, $2, 6$ to I_{01}, $1, 5$ to I_{10}, and $3, 7$ to I_{11}. The remaining promotions put every element in a linked list of length one, at which point $|w| = 3 = \log 8$.

Note how this construction ensures that there no two nodes appear as neighbors in more than one level. For example, at level 0, 4 is connected to 3 and 5. At level 1, 4 is connected to 2 and 6, and at level 2, 4 is connected to 0. Further note the recursive nature of this construction: if $\mathcal{I}_k$ is the interleaved skip graph on 2^k nodes, then it contains two copies of $\mathcal{I}_{k-1}$ with the nodes relabeled. That is, $\mathcal{I}_k = \{0, ..., 2^k\}$ contains $\mathcal{I}'_{k-1} = \{1, 3, 5, ..., 2^k - 1\}$ and $\mathcal{I}_{k-1} = \{0, 2, 4, ..., 2^k\}$, which both have the same structure on different node sets.

The term "interleaved" stems from the fact that in the recursive step we are promoting every other node to the same sub-skip graph. This ensures that every node receives at least one new edge from each level (not including the final promotion step, where each node rests in a linked list of length one). In general, every node in a list at level i will gain two new neighbors in its list at level $i + 1$ if it is at least

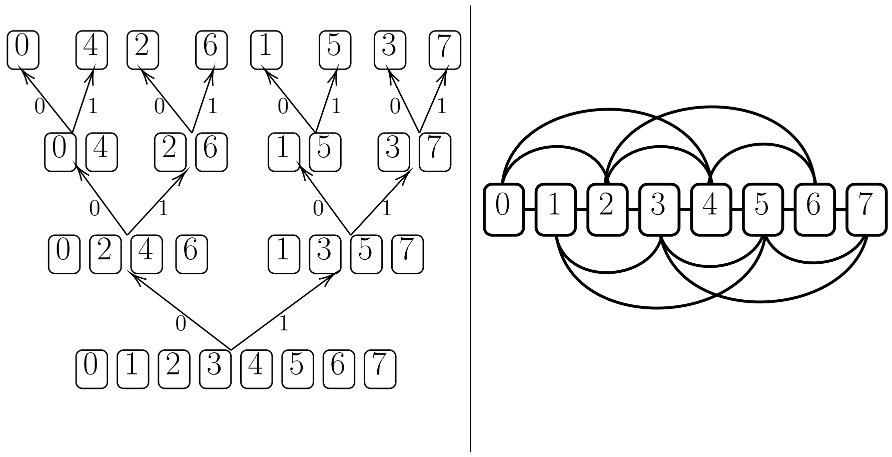


Figure 3.2: Interleaved skip graph on $n = 8$ nodes. (Left) Binary prefix tree representation. (Right) Corresponding network topology.

distance two from either end of the list at level i (e.g. any node that is not $0, 1, 6, 7$ in the level 0 list in Fig. 3.2). Nodes that are less than distance two from either end of the list at level i will gain one new neighbor when promoted, unless that list only contains two nodes. First, we discuss the interleaved skip graph's uncanny connection to Chord, and then we proceed to derive a closed form for its average path length, assuming n is a power of two. Then, we discuss our conjecture of optimality.

3.3.1 Cutting the Chord

The interleaved skip graph $\mathcal{I}$ bears a striking resemblance to Chord – or rather, BiChord, since links are bidirectional[2]. In particular, it seems to be a version of BiChord with the nodes laid out on a path instead of a ring[3].

Similarity in Structure

By promoting every other node to the same sub-skip graph, $\mathcal{I}$ induces links between two nodes if their rank difference (the distance between them on the path) is a power of two. The result is isomorphic to a version of BiChord where edges that "straddle" the cut-point between $n-1$ and 0 are removed. Formally, edge (u, v) in BiChord – with either $v = (u + 2^i) \bmod n$ or $u = (v + 2^i) \bmod n$ for the appropriate i – **straddles** the aforementioned cut-point if both of the following are true (Fig. 3.3):

- $u + 2^i \neq (u + 2^i) \bmod n$ or $v + 2^i \neq (v + 2^i) \bmod n$.
- The clockwise distance between u and v is not equal to the clockwise distance between v and u.

[2]Interestingly, we discovered the interleaved skip graph independent from Chord. It was not until several weeks later that we realized their resemblance.

[3]Hence the catchphrase of this thesis: "Cutting the Chord." Credit goes to my co-adviser Marcus for the pun.

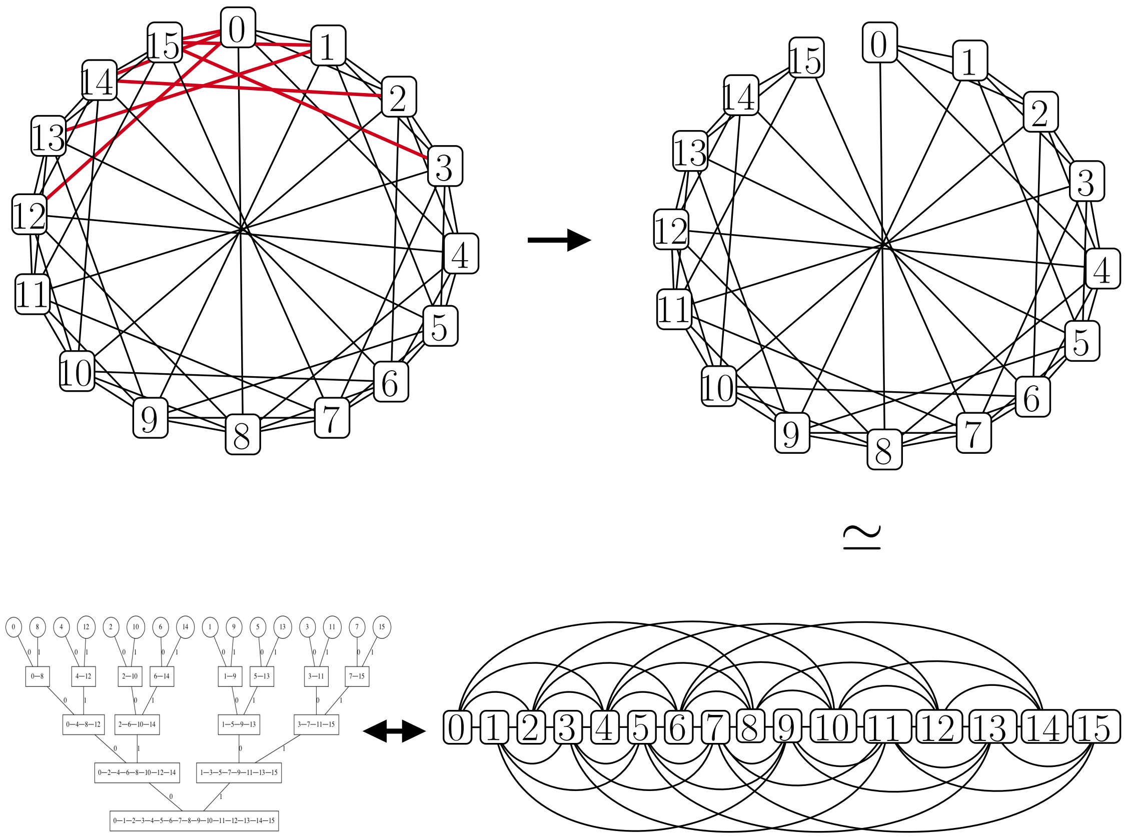


Figure 3.3: The interleaved skip graph is isomorphic ($\simeq$) to BiChord with the straddling edges (in red) removed.

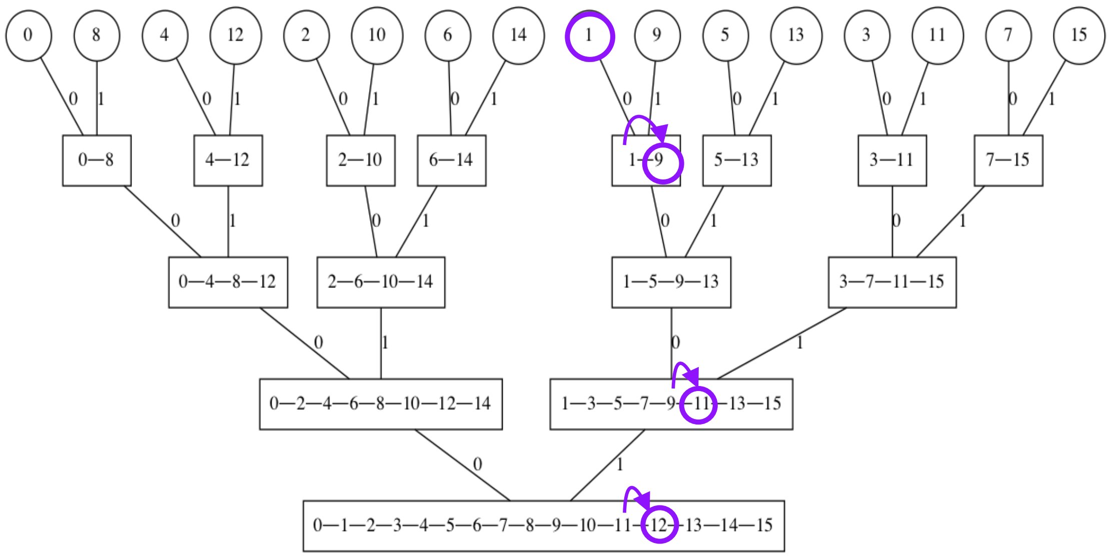


Figure 3.4: Node 1 performs a skip list search for node 12 in $\mathcal{I}_4$, the interleaved skip graph on 16 nodes, represented as a prefix tree. Note that the higher the level, the larger the power of two is for the rank difference between neighbors.

Similarity in Routes

The skip graph routing algorithm on $\mathcal{I}$ is nearly identical to Chord's protocol, the only difference being the absence of the straddling edges. For a node u to route to v in $\mathcal{I}$, recall that u initiates a skip list search for v within its skip list restriction $\mathcal{L}_{u.\text{id}}$. Within $\mathcal{L}_{u.\text{id}}$, every node is directly linked to a node that is a power of two away in rank. More importantly, the larger their rank difference, the higher up they are linked together within $\mathcal{L}_{u.\text{id}}$. A consequence of this is that the skip graph routing algorithm will greedily route similar to Chord: if the current node is w and the destination is v, w will forward the message to the neighbor that reduces the remaining rank distance to v by the largest power of two less than that remaining distance (Fig. 3.4, Fig. 3.5).

The caveat is that routes on $\mathcal{I}$ will differ from that of Chord if the latter uses a straddling edge. In this case, the skip graph route is identical to Chord's route *if* Chord routes in the opposite direction (e.g. if Chord's protocol routes clockwise and encounters a straddling edge, then perform the search counterclockwise instead). Of course, this assumes bidirectional links, so it is perhaps more accurate to think of it as BiChord with Chord's protocol (Fig. 3.6).

3.3.2 Derivation of Average Path Length

In this section, we derive the average path length of $\mathcal{I}_k$, the interleaved skip graph on $n = 2^k$ nodes, using the skip graph route cost $d_{\mathcal{I}_k}(u, v)$ rather than the shortest path. For simplicity, let us write $\Upsilon_k = d_{\mathcal{I}_k}$ for the remainder of this proof.

Let $V = \{0, ..., 2^k - 1\}$ be the node set of $\mathcal{I}_k$, and let S_k be the **total path length**

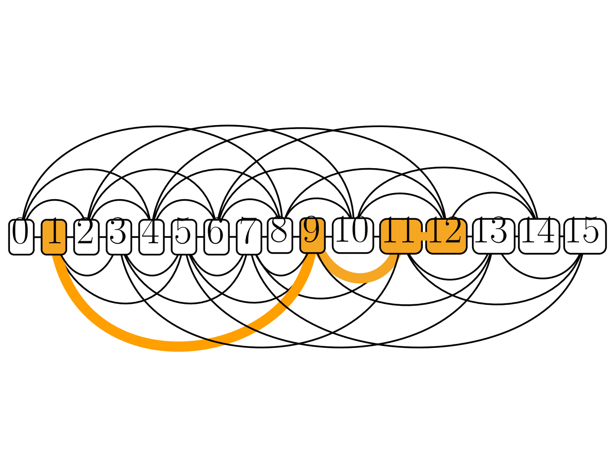


Figure 3.5: Route from 1 to 12 in Fig. 3.4 represented as a network (left). Routing from 1 to 12 in Chord (right). Note that the two search paths are identical.

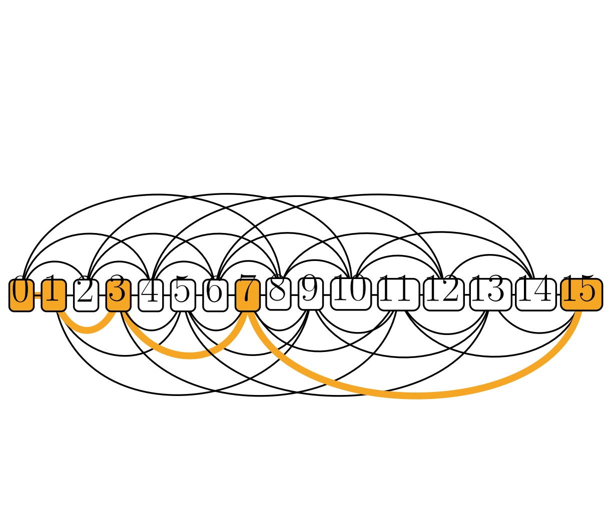


Figure 3.6: (Left) $\mathcal{I}_4$ routes from 15 to 0. (Right) Chord encounters a straddling edge when routing from 15 to 0 in the clockwise direction. Assuming bidirectional links, routing counterclockwise yields the same search path as the interleaved skip graph.

of $\mathcal{I}_k$:

$$S_k = \sum_{(u,v)\in V} \Upsilon_k(u,v). \tag{3.5}$$

Then, we have the following recurrence:

Theorem 3.3.1.

$$S_k = 4S_{k-1} + 2^{2k-1} - (k-1)2^{k-1}, \quad S_1 = 2. \tag{3.6}$$

Proof. We know that $\mathcal{I}_k$ contains two copies of $\mathcal{I}_{k-1}$ as sub-skip graphs. Labeling nodes by their index in the level 0 list, let $V_0 = \{u \mid u \in V \text{ and } u \text{ is even}\}$ and let $V_1 = V \setminus V_0$. Also let $A \sqcup B = (A \times B) \cup (B \times A)$. Then, by the recursive nature of $\mathcal{I}_k$,

$$\begin{aligned}
S_k &= \sum_{(u,v)\in V} \Upsilon_k(u,v) &(3.7)\\
&= \left(\sum_{(u,v)\in V_0} \Upsilon_{k-1}(u,v)\right) + \left(\sum_{(u,v)\in V_1} \Upsilon_{k-1}(u,v)\right) + \sum_{(u,v)\in V_0\sqcup V_1} \Upsilon_k(u,v) &(3.8)\\
&= 2S_{k-1} + \sum_{(u,v)\in V_0\sqcup V_1} \Upsilon_k(u,v) &(3.9)\\
&= 2S_{k-1} + X_k, &(3.10)
\end{aligned}$$

where the term $X_k = \sum_{(u,v)\in V_0\sqcup V_1} d'_{I_k}(u,v)$ accounts for the average path length due to node pairs that have opposite parity – i.e. those where the skip graph route paths must drop down to the level 0 list to complete.

For every $(u,v) \in V_0 \sqcup V_1$, note that $\Upsilon_k(u,v) = \Upsilon_{k-1}(u,t) + 1$, where t is the node with the largest key less than v (if $u < v$) or the node with smallest key greater than v (if $u > v$). This lets us write

$$X_k = \sum_{(u,v)\in V_0\sqcup V_1} (\Upsilon_{k-1}(u,t) + 1) = \left(\sum_{(u,v)\in V_0\sqcup V_1} \Upsilon_{k-1}(u,t)\right) + 2^{2k-1} \tag{3.11}$$

since $|V_0 \sqcup V_1| = 2(2^{k-1} \cdot 2^{k-1}) = 2^{2k-1}$. Let $\mathcal{I}^u_{k-1}$ be the copy of $\mathcal{I}_{k-1}$ that contains u. Since $t = v - 1$ or $t = v + 1$, when calculating the sum in Eq. (3.11), every possible skip graph search path in $\mathcal{I}^u_{k-1}$ is traversed *except* the search paths in $\mathcal{I}^u_{k-1}$ that end at 0 (if u is even) or at $2^k - 1$ (if u is odd). This is because when u searches for t in $\mathcal{I}^u_{k-1}$, 0 and $2^k - 1$ will not be used by the skip graph search algorithm to route from any u to v ($(u,v) \in V_0 \sqcup V_1$), unless u is either 0 or $2^k - 1$ to begin with. This lets us decompose Eq. (3.11) to

$$\begin{aligned}
X_k = &\left(\sum_{(u,w)\in V_0} \Upsilon_{k-1}(u,w)\right) - \sum_{u\in V_0} \Upsilon_{k-1}(u,0) &(3.12)\\
&+ \left(\sum_{(u,w)\in V_1} \Upsilon_{k-1}(u,w)\right) - \sum_{u\in V_1} \Upsilon_k(u,2^k-1) + 2^{2k-1} &(3.13)\\
=&2S_{k-1} - 2Z_{k-1} + 2^{2k-1} &(3.14)
\end{aligned}$$

where we let $Z_{k-1} = \sum_{u \in V_0} \Upsilon_{k-1}(u, 0) = \sum_{u \in V_1} \Upsilon_{k-1}(u, 2^k - 1)$. In other words, Z_k computes the sum total distances to route from any node to one of the endpoints in the level 0 list of $\mathcal{I}_k$ (either 0 or $2^k - 1$). We can characterize Z_k using the recursive nature of $\mathcal{I}_k$:

$$\begin{aligned}
Z_k &= \sum_{u \in V} \Upsilon_k(u, 0) && (3.15)\\
&= \left(\sum_{u \in V_0} \Upsilon_{k-1}(u, 0) \right) + \left(\sum_{u \in V_1} \Upsilon_k(u, 1) + 1 \right) && (3.16)\\
&= Z_{k-1} + \left(Z_{k-1} + \sum_{u \in V_1} 1 \right) && (3.17)\\
&= 2Z_{k-1} + 2^{k-1}. && (3.18)
\end{aligned}$$

Note that Eq. (3.17) follows because 1 is an endpoint of the level 0 list of $\mathcal{I}_{k-1}^u$ (for u odd), and because $|V_1| = |V|/2 = 2^k/2 = 2^{k-1}$.

For the base case, $Z_1 = 1$, because the cost to route from 1 to 0 in $\mathcal{I}_1$ is 1. Solving Eq. (3.15) yields

$$\begin{aligned}
Z_k &= 2Z_{k-1} + 2^{k-1}, \quad Z_1 = 1 && (3.19)\\
&= \sum_{i=1}^{k} 2^{i-1} \cdot 2^{k-i} = \sum_{i=1}^{k} 2^{k-1} && (3.20)\\
&= k2^{k-1}. && (3.21)
\end{aligned}$$

Substituting into Eq. (3.14), we have

$$X_k = 2S_{k-1} - 2(k-1)2^{k-2} + 2^{2k-1}, \qquad (3.22)$$

which we finally substitute into Eq. (3.10) to obtain

$$\begin{aligned}
S_k &= 2S_{k-1} + 2S_{k-1} - 2(k-1)2^{k-2} + 2^{2k-1} && (3.23)\\
&= 4S_{k-1} + 2^{2k-1} - (k-1)2^{k-1}. && (3.24)
\end{aligned}$$

For the base case, $S_1 = \Upsilon_1(0, 1) + \Upsilon_1(1, 0) = 2$. □

Now, we leverage the recurrence in Theorem 3.3.1 to calculate a closed form for S_k.

Corollary 3.3.2. $S_k = 2^{k-1}(2^k(k-1) + k + 1)$.

Proof. We proceed by induction. For the base case, we know $S_1 = 2$ from Theorem 3.3.1, which matches $2^{1-1}(2^1(1-1) + 1 + 1) = 2$. Assume $S_k = 2^{k-1}(2^k(k-1) + k + 1)$ for some $k \geq 1$. By Theorem 3.3.1, we have

$$S_{k+1} = 4S_k + 2^{2(k+1)-1} - k2^k. \qquad (3.25)$$

By the inductive hypothesis,

$$= 2^2(2^{k-1}(2^k(k-1)+k+1)) + 2^{2(k+1)-1} - k2^k \tag{3.26}$$
$$= 2^{2k+1}(k-1) + k2^{k+1} + 2^{k+1} + 2^{2k+1} - k2^k \tag{3.27}$$
$$= 2^{2k+1}k + 2^{k+1}(k+1) - k2^k \tag{3.28}$$
$$= 2^k(2^{k+1}k - k + 2(k+1)) \tag{3.29}$$
$$= 2^k(2^{k+1}k + k + 2). \tag{3.30}$$

□

The average path length is just the total path length divided by the number of node pairs $\left(2^k\right)^2$, which gives us:

$$\mathrm{APL}(\mathcal{I}_k) = \frac{S_k}{2^{2k}} = \frac{2^{k-1}(2^k(k-1)+k+1)}{2^{2k}} \tag{3.31}$$
$$= \frac{2^k(k-1)+k+1}{2^{k+1}} \tag{3.32}$$
$$= \frac{k}{2} + \frac{k+1}{2^{k+1}} - \frac{1}{2} \tag{3.33}$$
$$= \frac{k}{2} - \Theta(1). \tag{3.34}$$

Just to be safe, we have experimentally verified this value for all $0 \le k \le 10$. Unsurprisingly, $\mathrm{APL}(\mathcal{I}_k) = \Theta(k) = \Theta(\log n)$. For comparison, the average path length of Chord is $k/2$ [21] and the average path length of the optimal protocol[4] on BiChord is $k/3 + \Theta(1)$ [9].

3.3.3 Conjecture of Optimality

We conjecture that the interleaved skip graph is optimal:

Conjecture 3.3.1. The interleaved skip graph on $n = 2^k$ nodes has minimal average path length (when using skip graph routes as the distance function) across all skip graphs on $n = 2^k$ nodes.

The intuition behind this conjecture is that by promoting every *other* node to the same sub-skip graph, we are maximizing the number of links added to the skip graph on a per level basis. Under uniform demand (which average path length assumes), every request (node pair) is equally likely to be sampled, so all search paths are weighted equally. Because the interleaved skip graph is constructed without regard to any demand, adding as many links as possible to the network at each level such that each node gains at least one new link seeks to shorten search path lengths about equally well across all node pairs. We believe that this allows for minimal average path length across all skip graphs.

[4]The optimal protocol means routes are shortest paths.

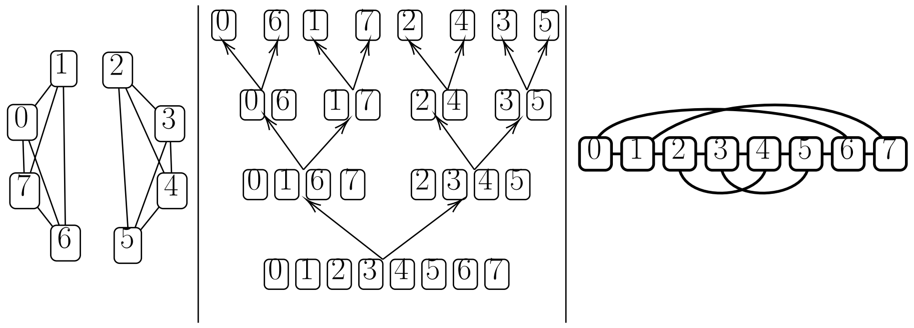


Figure 3.7: (Left) Uniform clustering demand D with disjoint subsets $V_1 = \{0, 1, 6, 7\}$ and $V_2 = \{2, 3, 4, 5\}$. Edge weights and directed edges omitted for simplicity. (Middle) Conjectured optimal skip graph for D represented as a prefix tree, comprising of the interleaved skip graphs for V_1 and V_2. (Right) Corresponding network topology.

We have spent a non-trivial amount of time and effort trying to prove a relaxed version of this conjecture: that the interleaved skip graph is optimal across all *balanced* skip graphs, which seems like a more approachable problem while still being a meaningful result. For $k \leq 4$, we have computationally verified that this is indeed the case. Perhaps a proof will turn up soon!

Extension to Uniform Clustering Demands

Assuming the conjecture is true, we can extend it to demands that describe clustering behaviour, which often arise in practice [2]. A **uniform clustering demand** defines clusters of nodes that only communicate uniformly amongst themselves. More formally, it is given by a demand D that, when given disjoint subsets $V_1, V_2, ...$ of the nodes $V = \{0, ..., n-1\}$ such that the size of each V_i is a power of two, is:

$$D(u, v) = \begin{cases} 1/C & u, v \text{ in the same } V_i \\ 0 & \text{else} \end{cases} \tag{3.35}$$

where C is a normalizing constant.

The requirement that cluster sizes are powers of two is simply so that the conjecture is applicable – the optimal skip graph for a uniform clustering demand would be the skip graph obtained by combining the interleaved skip graphs for each V_i (Fig. 3.7).

Chapter 4

Optimal Skip Graph Heuristics

In this chapter, we describe and analyze the performance of a few heuristics for the optimal skip graph problem. Recall that the objective of the optimal skip graph problem is to find a skip graph with minimal weighted path length given a demand graph D on node set $V = \{0, ..., n-1\}$. We are unsure if there exists a polynomial time algorithm to find an optimal solution, so we design heuristics that seem to offer good performance in practice, in terms of both running time and solution quality.

4.1 Contraction Heuristics

Our heuristics follow the same basic structure. Given a demand graph D describing pairwise request probabilities for the n nodes $V = \{0, ..., n-1\}$, the general blueprint is as follows.

Definition 4.1.1. A **contraction heuristic** is a family of heuristics that share the following steps:

1. Initialize a **heuristic graph** H, which is an undirected, weighted complete graph on vertex set V with edge weights given according to some weight function $h(u, v)$. In this initialization step, each node in H can be thought of as a size one skip graph (i.e. a skip graph comprised of a single node.)

2. Using an **edge contraction**[1] subroutine, choose some set of edges in H to contract, and the order in which to contract them, if necessary [2]. For each edge that is being contracted, the two endpoints are merged together into a single node, which represents two skip graphs merging into a larger skip graph. The new node now represents this larger skip graph.

3. Using h, update edge weights for all edges that are now incident to any node that was born from an edge contraction in the previous step.

4. Repeat steps 2 - 4 until H contains only one node, which must contain a skip graph of size n. Output this skip graph.

[1]This is defined on the following page.

[2]Order of contraction only matters for edges that share an endpoint.

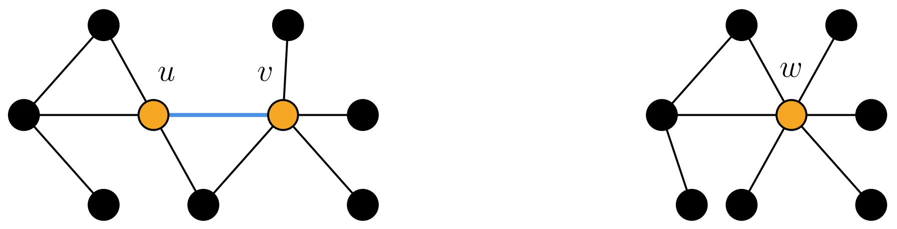


Figure 4.1: (Left) Before contracting edge (u, v). (Right) After contracting edge (u, v), merging u and v into a node w.

The two modifiable parameters in a contraction heuristic are the edge weight function h and the edge contraction subroutine. In general, the subroutine will make decisions based on the edge weights of H, and a good heuristic will have the edge weights depend on the input demand graph D as well. Designing the subroutine with the weight function in mind and vice versa is also essential to a good heuristic.

4.1.1 Edge Contractions

We first define what an edge contraction is. Given an undirected graph $G = (V, E)$ and an edge $(u, v) \in E$, an **edge contraction** at (u, v) will merge u and v into a single vertex w that is adjacent to all nodes that u and v were previously adjacent to. Edge (u, v) is subsequently deleted. We do not allow multiple edges from w to a node – if both u and v share a neighbor x, w will only have a single edge to x. See Fig. 4.1 for an example. In particular, note that contracting an edge in the complete graph on n nodes yields the complete graph on $n - 1$ nodes.

A contraction in the heuristic graph H corresponds to merging two skip graphs into one. Merging two skip graphs $\mathcal{S} = \{S_w\}$ and $\mathcal{T} = \{T_w\}$ involves making them sub-skip graphs of a new skip graph $\mathcal{R} = \{R_w\}$ where $R_{0w} = S_w$ and $R_{1w} = T_w$ for all w, and R_ϵ is the linked list that is the union of nodes in $\mathcal{S}$ and $\mathcal{T}$ in sorted order. We will denote $\mathcal{S} \oplus \mathcal{T}$ to be the skip graph returned by merging $\mathcal{S}$ and $\mathcal{T}$. See Fig. 4.2 for an example of this.

4.1.2 Comprehensive Weight Function

Because each node in the heuristic graph H represents a skip graph, our edge weight function h is essentially a map from the space of skip graph pairs to the real numbers. At any point in a contraction heuristic's execution, a node in H can represent any skip graph with a k-subset of V as its node set, for all $1 \leq k \leq |V|$.

One strategy for h is to have it encode the *improvement* in weighted path length due to merging two skip graphs. Since the contraction heuristic iteratively merges smaller skip graphs to eventually create a skip graph on the entirety of V, we design h so that it takes into account nodes that are outside of the skip graphs we are considering.

Definition 4.1.2. Let D be the demand graph on $V = \{0, ..., n-1\}$ and let $\mathcal{S}, \mathcal{R}$ be skip graphs where $\mathcal{S} \subseteq \mathcal{R} \subseteq V$.

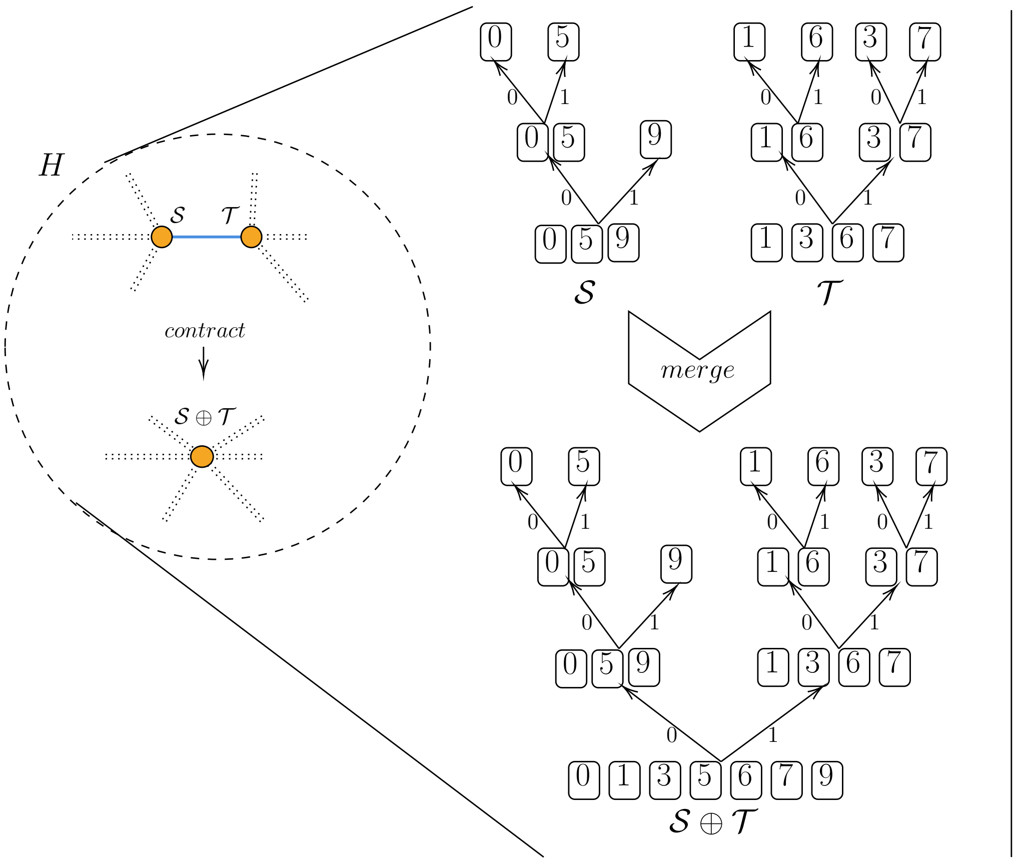


Figure 4.2: Contracting edge $(\mathcal{S}, \mathcal{T})$ in a heuristic graph H results in a new node, which corresponds to merging skip graphs $\mathcal{S}$ and $\mathcal{T}$ together into a new skip graph $\mathcal{S} \oplus \mathcal{T}$.

Then the **potential improvement** in weighted path length between $\mathcal{S}$ and $\mathcal{R}$ given D is

$$\Delta_D(\mathcal{S},\mathcal{R}) = \sum_{(u,v)\in\mathcal{S}\times V} D(u,v)\left(d_{\mathcal{S}}(u,w_{\mathcal{S}}) + |w_{\mathcal{S}} - v| - (d_{\mathcal{R}}(u,w_{\mathcal{R}}) + |w_{\mathcal{R}} - v|)\right) \quad (4.1)$$

where $w_{\mathcal{S}}$ is the largest node at most v (if $u < v$) or the smallest node at least v (if $u > v$) present in $\mathcal{S}$. We will refer to Δ_D as simply the potential improvement.

Let us dissect Eq. (4.1) for a moment, focusing on the term $d_{\mathcal{S}}(u,w_{\mathcal{S}}) + |w_{\mathcal{S}} - v| - (d_{\mathcal{R}}(u,w_{\mathcal{R}}) + |w_{\mathcal{R}} - v|)$. This term describes how much the cost of the search path from u to v is improved by $\mathcal{R}$ from $\mathcal{S}$:

- $d_{\mathcal{S}}(u,w_{\mathcal{S}})$ gives us the cost to route from u to v as far as possible *within* $\mathcal{S}$.
- $|w_{\mathcal{S}} - v|$ gives us the leftover cost to route to the destination v, from wherever the search path ended in $\mathcal{S}$.

Therefore the hope is that $\mathcal{R}$, which is a skip graph that contains the nodes in $\mathcal{S}$, can have a smaller $d_{\mathcal{R}}(u,w_{\mathcal{R}}) + |w_{\mathcal{R}} - v|$ – that is, if it can get closer to v with less cost. In general, we can think of the term $d_{\mathcal{S}}(u,w_{\mathcal{S}}) + |w_{\mathcal{S}} - v|$ as the "route cost so far" from u to v, since the nodes in $V \setminus \mathcal{S}$ have not yet been incorporated.

The potential improvement is a measure of how much the weighted path length *improves* across source-destination pairs where sources are specifically in $\mathcal{S}$. Summing over the pairs in $\mathcal{S}\times V$ allows us to consider destinations outside $\mathcal{S}\cup\mathcal{R}$. The intuition behind this is that we can measure the cumulative improvement of potentially incomplete search paths to every possible destination, rather than the search paths that begin and end entirely within $\mathcal{S}$. Thus, assuming that $\mathcal{R}$ contains $\mathcal{S}$ as a sub-skip graph (as in Definition 4.1.3, where $\mathcal{R} = \mathcal{S}\oplus\mathcal{T}$), we obtain a more faithful picture of how much those search paths to destinations outside $\mathcal{S}$ are improved by $\mathcal{R}$. Because we are striving to minimize weighted path length, the larger $\Delta_D(\mathcal{S},\mathcal{R})$ is, the more $\mathcal{R}$ is an improvement over $\mathcal{S}$.

Definition 4.1.3. Let D be a demand graph on $V = \{0, ..., n-1\}$. Given two skip graphs $\mathcal{S} \subseteq V$ and $\mathcal{T} \subseteq V$ in the heuristic graph, the **comprehensive weight function** on D is given by

$$h_D(\mathcal{S},\mathcal{T}) = \Delta_D(\mathcal{S},\mathcal{S}\oplus\mathcal{T}) + \Delta_D(\mathcal{T},\mathcal{S}\oplus\mathcal{T}). \quad (4.2)$$

In other words, $h_D(\mathcal{S},\mathcal{T})$ measures the potential improvement gained from merging $\mathcal{S}$ and $\mathcal{T}$ – and since we aim to minimize weighted path length, the larger h_D is, the more improvement you get from merging $\mathcal{S}$ and $\mathcal{T}$. We call h_D the comprehensive weight function because, via the potential improvement, it accounts for all nodes in V despite the fact that $\mathcal{S}\cup\mathcal{T}$ is only a subset of V.

Example 4.1.1. Suppose we have the setup in Fig. 4.3, with skip graphs $\mathcal{S} = \{2,4\}, \mathcal{T} = \{0\}$ as shown and node set $V = \{0,1,2,3,4\}$. Then $\mathcal{S}\oplus\mathcal{T} = \{0,2,4\}$ as shown. We would like to compute $h_D(\mathcal{S},\mathcal{T})$, where D is uniform demand on the node pairs of V, so $D(u,v) = 1/25$ for all u, v.

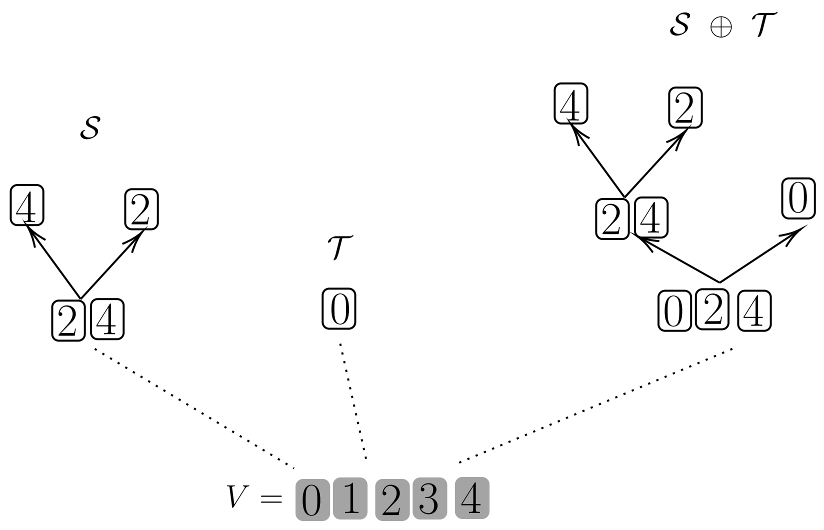


Figure 4.3: Computing edge weight $(\mathcal{S}, \mathcal{T})$ using the comprehensive weight function in Definition 4.1.3

.

- First, we compute $\Delta_D(\mathcal{S}, \mathcal{S} \oplus \mathcal{T})$. Since $\mathcal{S}$ is a sub-skip graph of $\mathcal{S} \oplus \mathcal{T}$, we only have to consider the pairs $\{(2,0),(2,1),(2,3),(4,0),(4,1),(4,3)\}$. This is because the search paths between $(4,2),(2,4)$ do not change between $\mathcal{S}$ and $\mathcal{S} \oplus \mathcal{T}$ and the pairs $(2,2),(4,4)$ are trivially 0 in Eq. (4.1).

  Let $\mathcal{R} = \mathcal{S} \oplus \mathcal{T}$ for simplicity. We compute the term in the sum of Eq. (4.1) for each of these pairs, namely $d_{\mathcal{S}}(u, w_{\mathcal{S}}) + |w_{\mathcal{S}} - v| - d_{\mathcal{R}}(u, w_{\mathcal{R}}) - |w_{\mathcal{R}} - v|$ (we factor out $D(u,v)$ since it is uniform):

  - For $(u,v) = (2,0)$: $(0+2-1-0) = 1$
  - For $(u,v) = (2,1)$: $(0+1-0-1) = 0$
  - For $(u,v) = (2,3)$: $(0+1-0-1) = 0$
  - For $(u,v) = (4,0)$: $(1+2-2-0) = 1$
  - For $(u,v) = (4,1)$: $(1+1-1-1) = 0$
  - For $(u,v) = (4,3)$: $(0+1-0-1) = 0$

  And so $\Delta_D(\mathcal{S}, \mathcal{S} \oplus \mathcal{T}) = \frac{1}{25}(1+0+0+1+0+0) = 2/25$.

- Next we compute $\Delta_D(\mathcal{T}, \mathcal{S} \oplus \mathcal{T})$, where $\mathcal{T} \times V = \{(0,0),(0,1),(0,2),(0,3),(0,4)\}$. Excluding $(0,0)$, computing the term in the sum of Eq. (4.1) for each pair gives:

  - For $(u,v) = (0,1)$: $(0+1-0-1) = 0$
  - For $(u,v) = (0,2)$: $(0+2-1-0) = 1$
  - For $(u,v) = (0,3)$: $(0+3-1-1) = 1$
  - For $(u,v) = (0,4)$: $(0+4-2-0) = 2$

  And so $\Delta_D(\mathcal{T}, \mathcal{S} \oplus \mathcal{T}) = \frac{1}{25}(0+1+1+2) = 4/25$.

This gives $h_D(\mathcal{S}, \mathcal{T}) = 2/25 + 4/25 = 6/25$.

Note in Example 4.1.1, $\mathcal{S} \oplus \mathcal{T}$ improves routing costs for pairs $(2, 0)$, $(4, 0)$, $(0, 2)$, $(0, 3)$ and $(0, 4)$ and does not improve nor worsen the routing costs for the other pairs. This leads to the following property.

Theorem 4.1.1. For any $\mathcal{S}, \mathcal{T} \subset V$ such that $\mathcal{S} \cap \mathcal{T} = \varnothing$, the skip graph $\mathcal{S} \oplus \mathcal{T}$ does not worsen the route cost so far for any $(u, v) \in V \times V$. In other words, for all such valid (u, v),

$$d_{\mathcal{S}}(u, w_{\mathcal{S}}) + |w_{\mathcal{S}} - v| \geq d_{\mathcal{R}}(u, w_{\mathcal{R}}) + |w_{\mathcal{R}} - v| \tag{4.3}$$

where $\mathcal{R} = \mathcal{S} \oplus \mathcal{T}$ and $w_{\mathcal{S}}$ is as in Definition 4.1.2.

Proof. We have the following:

- If $(u, v) \in (\overline{\mathcal{S} \cup \mathcal{T}}) \times V$, the source u is neither in $\mathcal{S}$ nor $\mathcal{T}$, and so the search path from u to v does not exist in both $\mathcal{S}$ and $\mathcal{R}$.

- Suppose $(u, v) \in (\mathcal{S} \cup \mathcal{T}) \times V$. Without loss of generality, let $u \in \mathcal{S}$ (the case where $u \in \mathcal{T}$ is identical). In this case, $d_{\mathcal{R}}(u, w_{\mathcal{R}}) = d_{\mathcal{S}}(u, w_{\mathcal{S}}) + \delta$ where δ is the rank difference between $w_{\mathcal{R}}$ and $w_{\mathcal{S}}$ in linked list R_ϵ. Since R_ϵ is a subset of V, it must be that $\delta \leq |w_{\mathcal{R}} - w_{\mathcal{S}}|$, the rank difference between them in V. Now observe that $|w_{\mathcal{S}} - v| = |w_{\mathcal{R}} - v| + |w_{\mathcal{R}} - w_{\mathcal{S}}|$ because $w_{\mathcal{R}}$ must be between (or equal to one of) $w_{\mathcal{S}}$ and v. Combining these observations (Fig. 4.4) yields:

$$|w_{\mathcal{S}} - v| = |w_{\mathcal{R}} - v| + |w_{\mathcal{R}} - w_{\mathcal{S}}| \tag{4.4}$$
$$|w_{\mathcal{S}} - v| \geq |w_{\mathcal{R}} - v| + \delta \tag{4.5}$$
$$d_{\mathcal{S}}(u, w_{\mathcal{S}}) + \delta + |w_{\mathcal{S}} - v| \geq d_{\mathcal{R}}(u, w_{\mathcal{R}}) + |w_{\mathcal{R}} - v| + \delta \tag{4.6}$$
$$d_{\mathcal{S}}(u, w_{\mathcal{S}}) + |w_{\mathcal{S}} - v| \geq d_{\mathcal{R}}(u, w_{\mathcal{R}}) + |w_{\mathcal{R}} - v|. \tag{4.7}$$

□

Corollary 4.1.2.

$$h_D(\mathcal{S}, \mathcal{T}) \geq 0. \tag{4.8}$$

Proof. By Theorem 4.1.1, every term in the sum of $h_D(\mathcal{S}, \mathcal{T})$ will be positive. □

Corollary 4.1.2 tells us that merging two skip graphs in a contraction heuristic cannot worsen the weighted path length across source-destination pairs whose route costs are at least partially measurable (i.e. across source-destination pairs in $V \times V$ where the source u is either in $\mathcal{S}$ or $\mathcal{T}$, because then we can at least partially compute the search path starting at u).

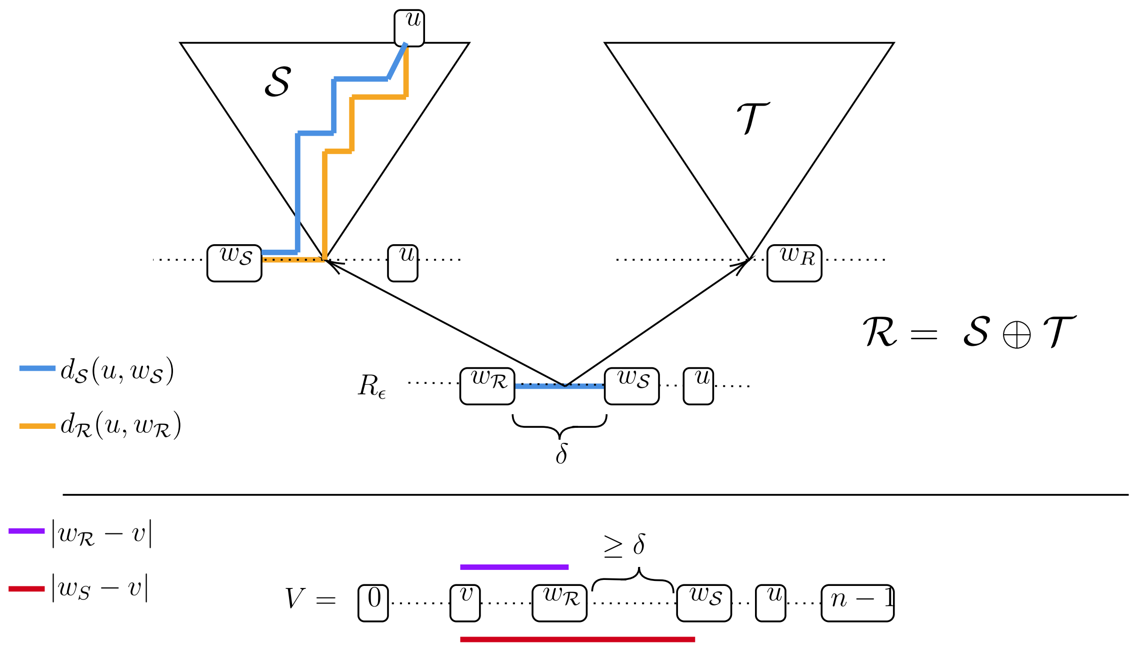


Figure 4.4: Proof of Theorem 4.1.1. In this figure, $w_\mathcal{S} \neq w_\mathcal{R}$, but it could be possible for them to be equal, in which case $\delta = 0$.

4.1.3 Edge Contraction Subroutines

In our case, an edge contraction subroutine is given a heuristic graph H that has edge weights according to the comprehensive weight function h_D (Definition 4.1.3). The subroutine selects a subset of edges of H to contract, and the order in which to contract them, if necessary. We present two subroutines that only select edges that do not share endpoints, so we do not need to consider the order of contraction.

Maximum Cost Edge

The first subroutine is to just choose the edge with the largest weight. We will call it the `MaxEdge` subroutine. `MaxEdge`(H) returns the edge in heuristic graph H with the largest weight. The intuition is that we scan through all edges in H and merge the two skip graphs that gain the most improvement from merging.

Maximum Cost Matching

Our second subroutine `MaxMatching` is similar to `MaxEdge` in the sense that it is also greedy. First, a quick review on matchings:

- Given an undirected graph G, recall that a **matching** on G is a subset of its edges such that no two edges share an endpoint.

- A **maximal matching** is a matching M on G that is not a subset of any other matching on G (i.e. every edge that is not in M shares an endpoint with some edge in M). See Fig. 4.5.

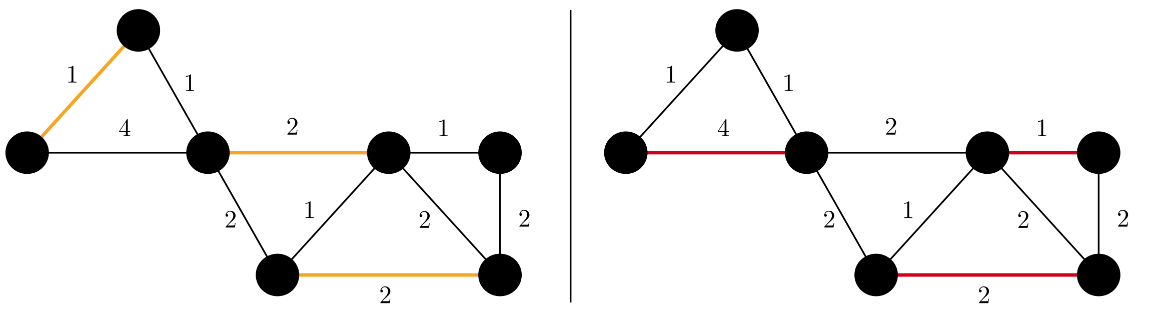


Figure 4.5: (Left) Maximal matching. (Right) Maximum cost matching (which is also a maximal matching).

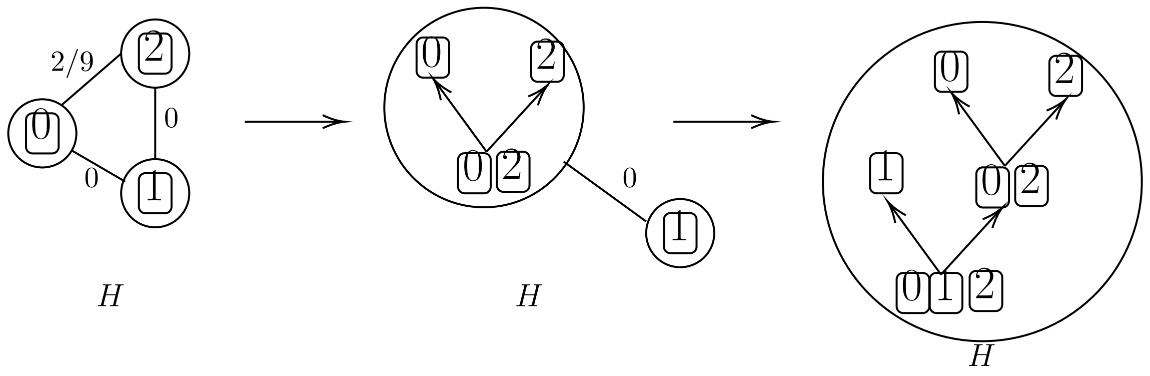


Figure 4.6: Example execution of the contraction heuristic in Example 4.1.2. Circles represents nodes in the heuristic graph H. Arrows transition between edge contractions.

- A **maximum cost matching** is a matching where the sum of edge weights of all edges in the matching is maximized. Due to Corollary 4.1.2, all edge weights are positive, so any maximum cost matching on a heuristic graph H must be a maximal matching as well. See Fig. 4.5. The maximum cost matching can be computed in polynomial time using Edmond's Blossom Algorithm [6].

We define `MaxMatching`(H) to return a maximum cost matching on H. The intuition is that it generalizes the greedy approach of just picking the heaviest edge. Instead, we merge as many skip graphs as we possibly can in H in a way that maximizes the total improvement.

Example 4.1.2. Suppose D is once again, uniform demand on $V = \{0, 1, 2\}$, so $D(u, v) = 1/9$ for all (u, v). See Fig. 4.6 for the execution of the contraction heuristic on this example, given comprehensive edge weights h_D. Note that since $|V| = 3$, both `MaxEdge` and `MaxMatching` perform identically, as the cardinality of any matching on a graph with 3 nodes will be 1.

Table 4.1: Ratio between average WPL of `Random` and `CMM` in Fig. 4.7.

n	$\frac{\text{WPL of Random}}{\text{WPL of CMM}}$
10	1.41
30	1.60
50	1.61
70	1.67

4.2 Empirical Performance Over Random Demand

In this section, we empirically evaluate the performance of the contraction heuristics from the previous section over random demand. We are interested in solution quality – the weighted path length of the skip graphs output by our heuristics. Let `CME` be the contraction heuristic with the `MaxEdge` subroutine and let `CMM` be the contraction heuristic with the `MaxMatching` subroutine, both using the comprehensive weight function. Let `Random` denote the random heuristic – that is, `Random` returns a size n skip graph chosen uniformly at random, independent of the input demand.

Randomly generated demand graphs in the following experiments are generated by first weighting each edge in the demand graph with a uniformly chosen value between 1 and 10 and then normalizing by the sum of the weights. All experiments were executed on an Intel core i5 processor with a 2.7GHz clock speed.[3].

4.2.1 Scaling

To get an idea of how the solution quality of our heuristics scale, we plot the weighted path length (averaged over 100 trials, each trial being a random demand graph on n nodes) against n for the skip graphs returned by `Random`, `CME`, and `CMM` in Fig. 4.7. Using `Random` as a point of comparison lets us ascertain whether our heuristics might offer an improvement over simply choosing a skip graph at random.

In particular, note that `CMM` outperforms `Random` and `CME` as n grows, with the gap between `Random` and `CMM` increasing (Table 4.1).

The inferior scaling of `CME` could be explained by the fact that the heuristic has a tendency to repeatedly merge skip graphs of size 1 with skip graphs of size greater than 1. This is because `MaxEdge` can only contract a single edge per iteration, and perhaps due to the comprehensive weight function, is more likely to choose an edge whose contraction merges a size 1 skip graph with a larger skip graph. This means `CME` is likely to output a skip graph $\mathcal{G}$ that contains a large **spine** – that is, a large sub-skip graph of $\mathcal{G}$ such that for all non-singleton linked lists G_w within that sub-skip graph, either G_{w0} or G_{w1} is a singleton (Fig. 4.9). Large spines only contribute at most one link to the network per level (Fig. 4.8), leading to a sparser network. This may help explain `CME`'s inferior performance, at least over random demand.

On the other hand, `CMM` uses the `MaxMatching` subroutine, which contracts all

[3]Code necessary for the experiments are available in a public repository: https://github.com/shastrihm/DemandAwareSkipGraphs

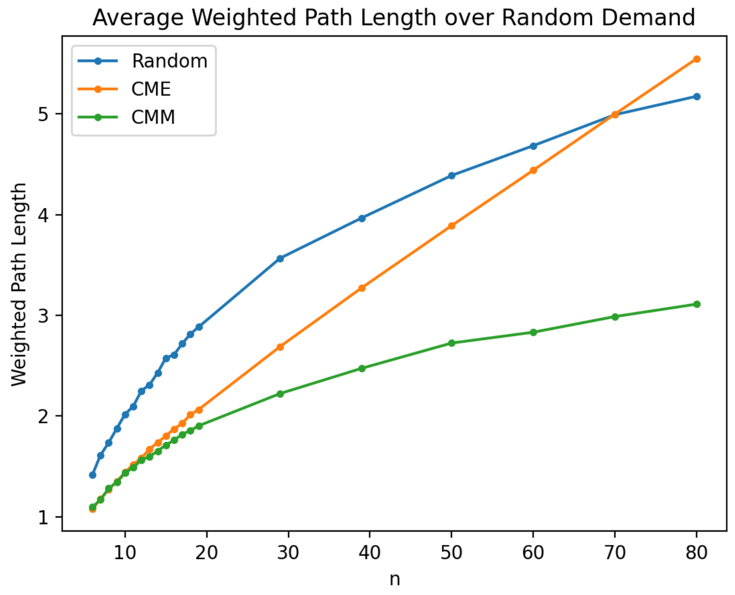


Figure 4.7: Weighted path length of size n skip graphs returned by the heuristics. Each data point is averaged over 100 trials. Due to computational constraints, sufficient data could not be collected for $n > 80$.

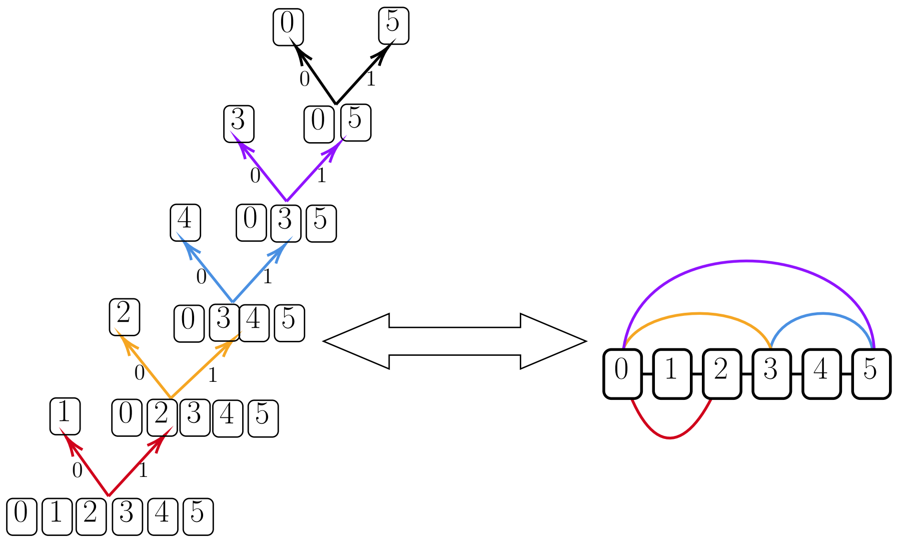


Figure 4.8: Spine skip graph on $n = 6$ nodes. Each level contributes only at most one link to the network.

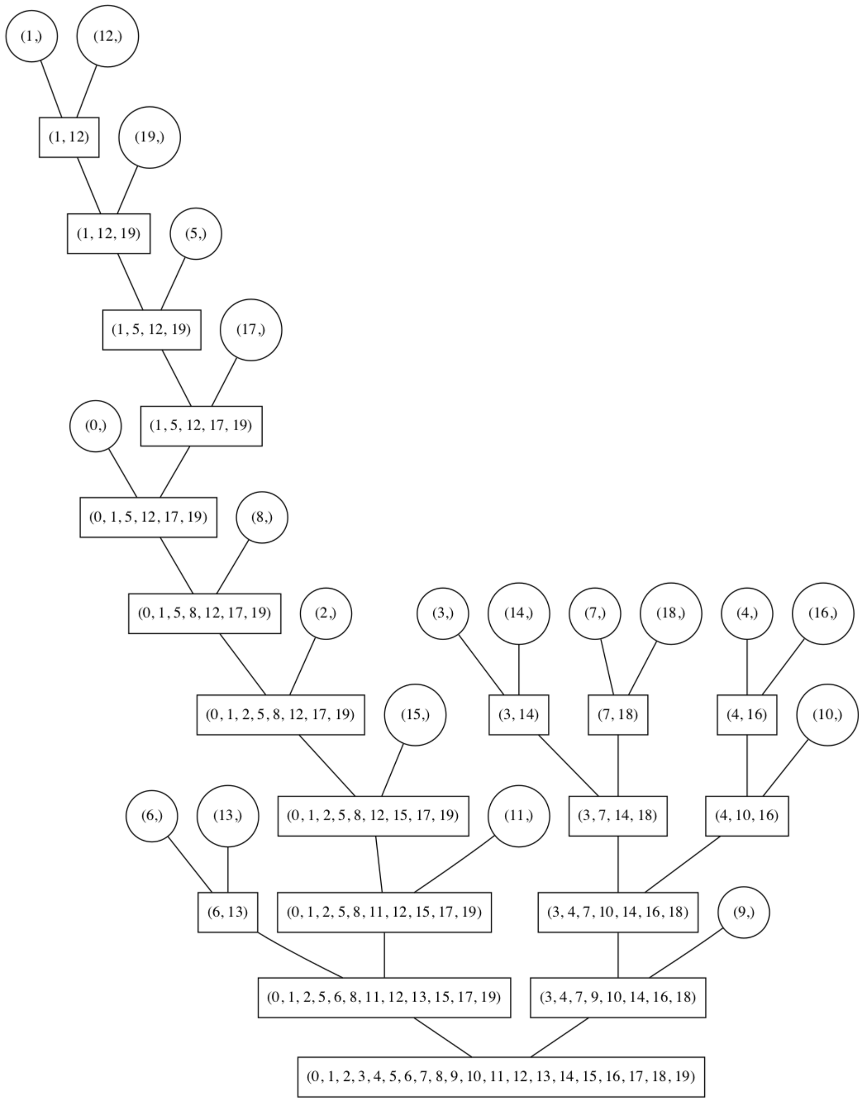


Figure 4.9: Skip graph output by `CME` when given uniform demand on $n = 20$ nodes. Note the spine sub-skip graph $\mathcal{G}_{01} = \{0, 1, 2, 5, 8, 11, 12, 15, 17, 19\}$.

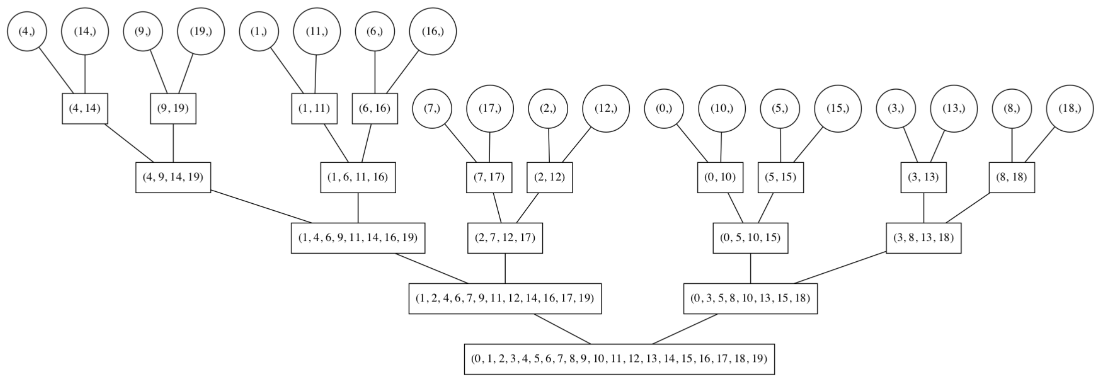


Figure 4.10: Skip graph output by `CMM` when given uniform demand on $n = 20$ nodes.

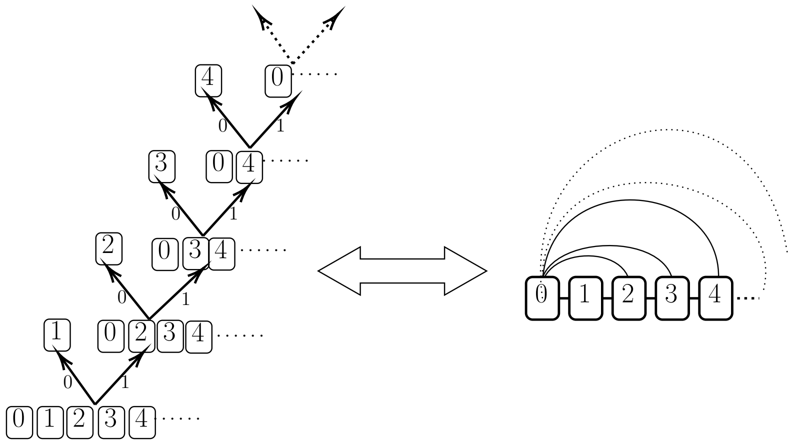


Figure 4.11: A spine skip graph is optimal for single-source demand.

edges in a maximum matching every iteration. Therefore the resulting skip graph will likely appear more balanced and less sparse when compared to `CME`, and additionally helps explain why it performs better across random demand (Fig. 4.10).

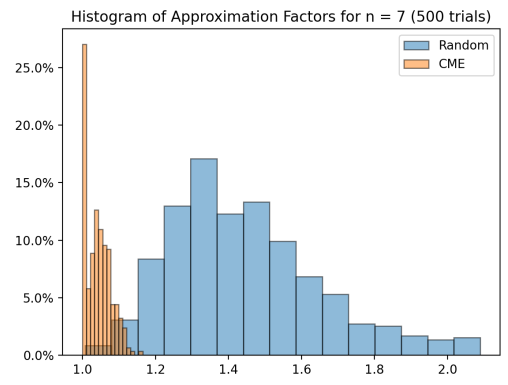
This does not discount the existence of demands where `CME` may outperform `CMM` – perhaps demands that benefit from spine skip graphs and are neglected by more balanced ones.

Example 4.2.1. An example of a demand that benefits from a spine skip graph is a *single-source demand* – e.g. $D(0, v) > 0$ and $D(u, v) = 0$ if $u \neq 0$, for all v.

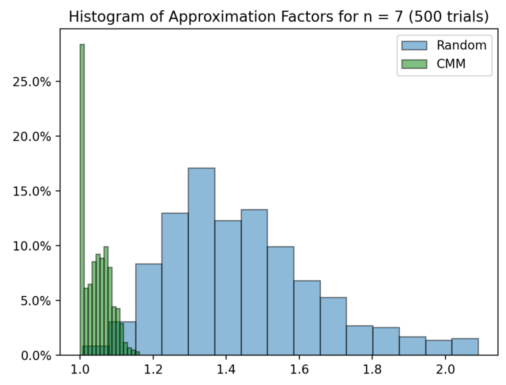
In this case, the optimal skip graph connects 0 to every node, which can be given by a spine (Fig. 4.11). A balanced skip graph would only be able to induce at most $\log n$ neighbors for 0 and thus would neglect this demand. Unfortunately, neither `CME` nor `CMM` discovers the optimal solution. This may be a future avenue to improve the heuristics.

4.2.2 Spread

Here, we plot histograms of weighted path lengths for skip graphs output by `Random`, `CME`, and `CMM` over random demand to get a sense of how spread out the distributions



(a) Histogram of approximation factors for CME and Random, over 500 random demands on $n = 7$ nodes.



(b) Histogram of approximation factors for CMM and Random, over 500 random demands on $n = 7$ nodes.

Figure 4.12: Histograms of approximation factors for $n = 7$. Here, the means are $\mu_{random} = 1.45$, $\mu_{CMM} = 1.05$, and $\mu_{CME} = 1.04$ and the standard deviations are $\sigma_{random} = 0.21$, $\sigma_{CMM} = 0.04$ and $\sigma_{CME} = 0.03$.

are.

For very small n, computing the optimal skip graph for a given demand is tractable. We first compute a histogram of approximation factors for $n = 7$. For a given demand D, the approximation factor α is simply defined as $\alpha = ALG/OPT$ where ALG is the weighted path length of the skip graph output by the heuristic in question and OPT is the optimal weighted path length. Because this is a minimization problem, $\alpha \geq 1$ with $\alpha = 1$ signifying that ALG is optimal.

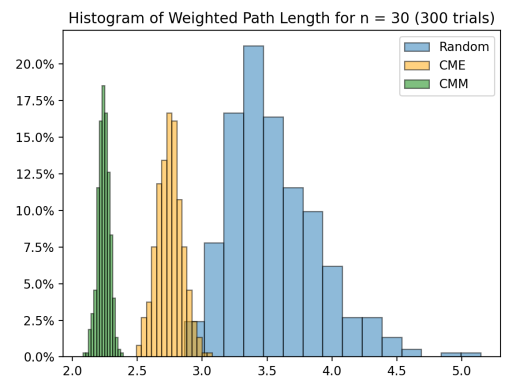
In Fig. 4.12, note that both CMM and CME are tightly clustered around $\alpha = 1.0$ while Random is spread much more between $\alpha = 1.0$ to $\alpha \approx 2.0$. This is reflected in the standard deviation, where $\sigma_{CMM} = 0.04, \sigma_{CME} = 0.03$ are almost an order of magnitude smaller than $\sigma_{random} = 0.21$. Furthermore, CME found the optimal solution (i.e. $\alpha = 1$) 22% of the time and CMM found the optimal solution 23% of the time, while Random never found the optimal solution.

For larger n, computing the optimal solution is no longer tractable. We instead plot histograms of weighted path length (Fig. 4.13).

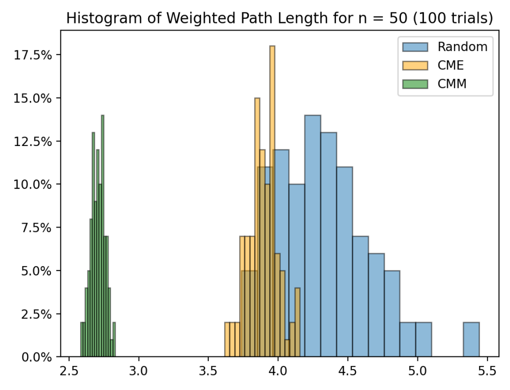
In Fig. 4.13, the fact that the distribution of CME shifts further to the right as n grows (eventually surpassing Random) reflects the scaling behavior in Fig. 4.7. Not only does CMM outperform both CME and CMM, but its spread is slightly smaller than CME and significantly smaller than Random, even as n grows, as illustrated by the standard deviations in Table 4.2.

The standard deviation of CMM is approximately an order of magnitude smaller than Random even as n grows. According to these results, if this trend holds across random demand, obtaining a skip graph from CMM will likely result in a weighted path length very close to the mean, which is already significantly smaller than Random.

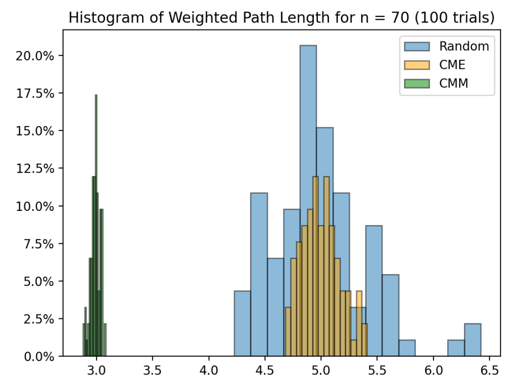
Remark 4.2.1. Interestingly, the interleaved skip graph (which we have conjectured



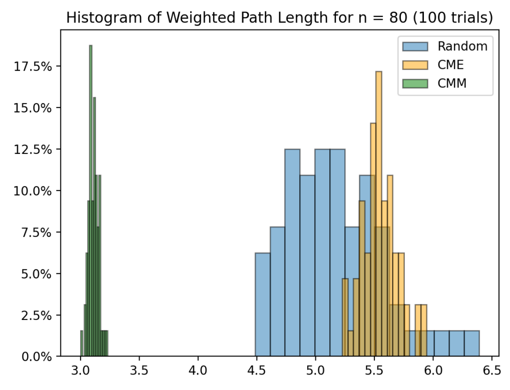
(a) Across 300 random demands for $n = 30$.



(b) Across 100 random demands for $n = 50$.



(c) Across 100 random demands for $n = 70$.



(d) Across 100 random demands for $n = 80$.

Figure 4.13: Histograms of weighted path length for $n = 30, 50, 70$ and 80.

Table 4.2: Mean and standard deviation (in parentheses) of weighted path lengths for `Random`, `CME`, and `CMM` as given in Fig. 4.13.

	$n = 30$	$n = 50$	$n = 70$	$n = 80$
`CMM`	2.24 (0.05)	2.71 (0.05)	2.99 (0.04)	3.11 (0.04)
`CME`	2.74 (0.10)	3.89 (0.11)	5.00 (0.17)	5.54 (0.16)
`Random`	3.55 (0.35)	4.32 (0.35)	5.00 (0.42)	5.17 (0.41)

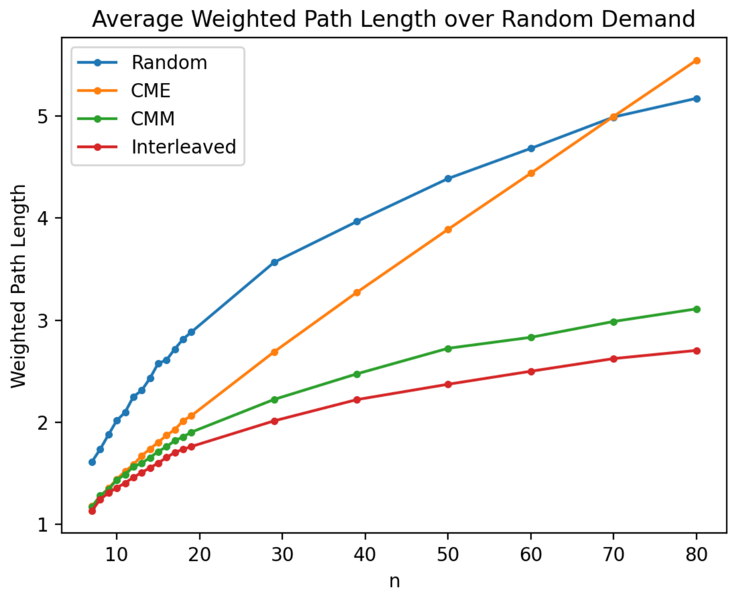


Figure 4.14: Weighted path length over random demand, with the interleaved skip graph (Each data point averaged over at least 100 trials).

to be optimal for uniform demand) outperforms `CMM` and `CME` over random demand, despite the fact that it does not depend on the input demand at all (Fig. 4.14). A possible explanation is that uniformly chosen random demands will average out to uniform demand, which allows the interleaved skip graph to perform well. Regardless, this speaks to a weakness of our heuristics and perhaps to the strength of the interleaved, suggesting that the extent to which our heuristics are exploiting the input demand can be significantly improved upon. Whether this occurs through an improved heuristic function, edge contraction subroutine, separate heuristic framework, or some combination of the three can be a subject of further inquiry.

Open Problems

Our exploration of network design through the lens of interleaved and demand aware skip graphs has gifted us with a rich assortment of potential future directions. Some could take an entire thesis to explore while others could be resolved in a page. In order of what I personally find most pressing to least:

1. In Section 3.3.3, we posed a conjecture that the interleaved skip graph on $n = 2^k$ has optimal average path length over all size $n = 2^k$ skip graphs. Proving this, or even the relaxed conjecture that it is optimal over all such *balanced* skip graphs, is a concrete question that can be immediately attacked. Extending it to when n is not a power of two is also an open problem.

2. Understanding the hardness of the optimal skip graph problem (Problem 3.1.1) would clarify the need for heuristics. It would be especially neat if the problem admits a polynomial time algorithm.

3. Our definition of the optimal skip graph problem assumed that the input demand graph D and the skip graph to be optimized both had identical node sets V. A caveat with this formulation is that it causes the demands between nodes that have rank difference 1 to be essentially irrelevant to the problem. In other words, regardless of whether $D(u, u+1)$ is small or large, u and $u+1$ will be linked together in the resulting skip graph, because we mandate that lists be in sorted order.

   A formulation of the optimal skip graph problem that might be more practically interesting (and left for future work) is to combine our notions of MEPL and OSG: to find an embedding of demands onto machines *while* searching for a skip graph topology to interconnect the machines that together minimizes expected path length. To do this, we make the node set V for the input demand graph D be unrelated to the set of nodes V' with which we are trying to find an optimal skip graph. Then, the problem becomes one of finding an embedding $\phi^* : V \to V'$ and a skip graph $\mathcal{G}^*$ such that $(\phi^*, \mathcal{G}^*) = \arg\min_{\phi,\mathcal{G}} \mathrm{EPL}(D, \mathcal{G}, \phi)$ for an input demand D.

4. Considering the interleaved skip graph's close relationship with BiChord, we are left to wonder whether we can draw inspiration from [13], [11], and [9] to devise improved or even optimal routing algorithms for the interleaved skip graph. Perhaps these improvements could help improve the general skip graph routing algorithm as well.

5. Understanding the hardness of MEPL (Problem 2.2.3) when restricted to the interleaved skip graph is an interesting question. Our reductions in Section 2.3 obviously do not work because the interleaved skip graph is not isomorphic to the path graph, but it still seems possible that this special case is `NP`-complete.

6. In Section 3.2, we enumerated skip graphs by equating them with permutations on prefix tree leaves, and obtained a result for the number of skip graphs which distinguishes between skip graphs with different prefix tree representations but otherwise isomorphic network topologies. Tightening this result by enumerating skip graphs that determine *non-isomorphic* topologies appears to be a more challenging exercise.

Of course, this list is not exhaustive. Happy researching!

References

[1] Aspnes, J., & Shah, G. (2007). Skip graphs. *Acm transactions on algorithms (talg)*, *3*(4), 37–es.

[2] Avin, C., Borokhovich, M., Haeupler, B., & Lotker, Z. (2015). Self-adjusting grid networks to minimize expected path length. *Theoretical Computer Science*, *584*, 91–102.

[3] Avin, C., Salem, I., & Schmid, S. (2020). Working set theorems for routing in self-adjusting skip list networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, (pp. 2175–2184). IEEE.

[4] Avin, C., & Schmid, S. (2018). Toward demand-aware networking: A theory for self-adjusting networks.

[5] Bhatt, S. N., & Cosmadakis, S. S. (1987). The complexity of minimizing wire lengths in vlsi layouts. *Information Processing Letters*, *25*(4), 263–267.

[6] Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of mathematics*, *17*, 449–467.

[7] Even, S. (1975). Np-completeness of several arrangement problems. *Technical Report; Department of computer Science*, *43*.

[8] Foerster, K.-T., & Schmid, S. (2019). Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *ACM SIGACT News*, *50*(2), 62–79.

[9] Ganesan, P., & Manku, G. S. (2004). Optimal routing in chord. In *ACM SIAM Symposium on Distributed Algorithms (SODA) 2004*.

[10] Garey, M. R., Johnson, D. S., & Stockmeyer, L. (1974). Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, (p. 47–63). New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/800119.803884`

[11] Green, J. (2003). *Optimal routing in BiChord*. Reed College.

[12] Huq, S., & Ghosh, S. (2017). Locally self-adjusting skip graphs.

[13] Jiang, J., Pan, R., Liang, C., & Wang, W. (2005). Bichord: An improved approach for lookup routing in chord. In *East European Conference on Advances in Databases and Information Systems*, (pp. 338–348). Springer.

[14] Johnson, D. S., Lenstra, J. K., & Kan, A. R. (1978). The complexity of the network design problem. *Networks*, *8*(4), 279–285.

[15] Pugh, W. (1990). Skip lists: A probabilistic alternative to balanced trees. *33*(6). `https://doi.org/10.1145/78973.78977`

[16] Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001). A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (pp. 161–172).

[17] Rowstron, A., & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, (pp. 329–350). Springer.

[18] Schmid, S., Avin, C., Scheideler, C., Borokhovich, M., Haeupler, B., & Lotker, Z. (2015). Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking*, *24*(3), 1421–1433.

[19] Seitz, H. (2010). *Contributions to the minimum linear arrangement problem.* Ph.D. thesis.

[20] Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, *32*(3), 652–686. `https://doi.org/10.1145/3828.3835`

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, *31*(4), 149–160.

[22] Terashima, R. S. (2010). *Message routing in random graphs : lattice and ring models for peer-to-peer and social networks.* Reed College.

[23] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, *22*(1), 41–53.