Dynamic Necklace Splitting

Rishi Advani
University of Illinois Chicago, IL, USA

Abolfazl Asudeh
University of Illinois Chicago, IL, USA

Mohsen Dehghankar
University of Illinois Chicago, IL, USA

Stavros Sintos
University of Illinois Chicago, IL, USA

Abstract

The necklace splitting problem is a classic problem in fair division with many applications, including data-informed fair hash maps. We extend necklace splitting to a dynamic setting, allowing for relocation, insertion, and deletion of beads. We present linear-time, optimal algorithms for the two-color case that support all dynamic updates. For more than two colors, we give linear-time, optimal algorithms for relocation subject to a restriction on the number of agents. Finally, we propose a randomized algorithm for the two-color case that handles all dynamic updates, guarantees approximate fairness with high probability, and runs in polylogarithmic time when the number of agents is small.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics; Theory of computation → Algorithmic game theory; Theory of computation → Online algorithms

Keywords and phrases Necklace splitting, dynamic algorithms, fair division

1 Introduction

The necklace splitting problem, first introduced by Bhatt and Leiserson [6], is a classic combinatorial fair division problem. In this work, we extend it to a dynamic setting, enabling applications such as data-informed fair hash maps and improved load-balancing among multiple servers.

1.1 Problem Setup

We are given a string S of m *beads*. We denote the jth element in S by $S[j]$ and the substring from the j_1th element to the j_2th element by $[j_1, j_2]$.[1] Each bead is a certain *color* $i \in [n]$. Let $m_i = |\{b \in S \mid b \text{ is color } i\}|$. Let k be the number of agents. For simplicity, assume that k divides m_i for all i. The goal is to find a set P consisting of cuts of S with the property that the resulting set of intervals can be allocated to the agents in such a way that each agent receives exactly m_i/k beads of color i. An overview of the notation is given in Table 1.

We are interested in the dynamic case. We are given an instance of the necklace splitting problem solved using an offline algorithm (see Section 2). We allow the following dynamic updates:

Relocation Bead $S[j_1]$ is moved to index j_2.
Insertion αk beads of the same color are inserted into S.
Deletion αk beads of the same color are deleted from S.

We can also allow for recoloring of beads by performing successive deletion and insertion.

[1] For ease of notation, when $j_1 > j_2$, we write $[j_1, j_2]$ instead of $[j_1, m] \cup [1, j_2]$.

Table 1 Key notation used in the paper.

Notation	Description
S	The string representing the necklace
P	The set of cuts in S
m	The number of beads in S
m_i	The number of beads of color i in S
n	The number of distinct colors of beads in S
k	The number of agents
G	The neighborhood graph associated with P
T	The neighborhood tree associated with S and P
A	An arbitrary agent

1.2 Contributions

Here we give a brief overview of our contributions:

- We introduce and formalize the problem of dynamic necklace splitting.
- We design a linear-time algorithm for swapping adjacent beads when $n = 2$ that achieves the optimal bound of $2(k-1)$ cuts. We design two linear-time algorithms for relocation of arbitrary distance when $n = 2$ that achieve the optimal bound of $2(k-1)$ cuts. We also design a linear-time algorithm for relocation with looser bounds for general $n \geq 2$ (Section 3).
- We introduce the MinNodeMaxFlow problem and prove it to be NP-complete. We design an approximation algorithm for special cases that we then use as part of an algorithm to efficiently perform batch relocation when $n = 2$ with the optimal number of cuts (Section 4).
- We design two linear-time algorithms for relocation with general $n \geq 2$ when $m = nk$ that achieve the optimal bound of $n(k-1)$ cuts (Section 5).
- We adapt our algorithm for batch relocation to be used for efficient insertion and deletion (Section 6).
- We design a randomized, polylogarithmic-time algorithm for relocation, insertion, and deletion that produces an approximately fair set of cuts with high probability when $n = 2$ (Section 7).

See Table 2 for a comparison of our algorithms.

1.3 Data Structures

The choice of which data structures to use to store information about the necklace has a significant impact on the running time of our algorithms. We implement the necklace itself as a doubly linked list, allowing for efficient dynamic updates. Each node/bead also stores its index, the agent to whom it belongs, and a pointer to the next node/bead belonging to the same agent. Given any bead, this allows us to find the subsequence of beads belonging to the same agent in $O(m/k)$ time.

Note that the cuts are stored implicitly and can be explicitly generated in $O(m)$ time by iterating through the necklace and identifying pairs of consecutive beads where the associated agents switch. Alternatively, with minimal added cost, we can maintain a hash table mapping each possible pair of agents to the set of cuts adjacent to both agents.

Table 2 Key details of dynamic algorithms presented in the paper.

Algorithm	n	Update	Exact?	Optimal # of cuts?	Running time (per bead)
Swap	2	Swap	✓	✓	$O\left(\frac{m}{k}\right)$
Path	2	Any	✓	✓	$O\left(k + \frac{k'm}{k}\right)$
ColorPath	2	Any	✓	✓	$O\left(k + \frac{k'm}{k}\right)$
Fence	Any	Relocation	✓	✗	$O\left(\frac{m}{kn}\right)$
BatchPath	2	Any	✓	✓	$O\left(\log k + \frac{k'm}{km'}\right)$
DenseSwap	m/k	Swap	✓	✓	$O(n)$
DenseJump	m/k	Relocation	✓	✓	$O(k+n)$
Approx	2	Any	✗	N/A	$O\left(k^2 2^{2k} \varepsilon^{-2} (\log m)^2 + \log m\right)$

For certain algorithms, we need to make use of a *neighborhood graph* G where each vertex represents an agent and two vertices are joined by an edge if the corresponding agents possess adjacent beads. We implement G as an adjacency list. If the beads corresponding to k' agents are reassigned among the same agents, G can be updated in $O(k'm/k)$ time. We remove all edges incident to the k' agents and then determine which edges to add by iterating through the $k'm/k$ beads assigned to those agents.

1.4 Related Work

Soon after the necklace splitting problem was originally introduced [6], Goldberg and West [9] proved that a solution always exists if $k = 2$. Alon and West [3] gave a simpler proof using the Borsuk–Ulam theorem [7], and Alon [1] generalized the results to $k > 2$. Finally, Alon and Graur [2] discovered an efficient approximation algorithm for finding a solution with few cuts. Alon and Graur [2] also consider an online variation of the problem. They derive lower and upper bounds on the number of cuts needed when $k = 2$ based on the value of n. They also generalize some of these results to the $k > 2$ case.

In addition to necklace splitting, other fair division problems can be extended to dynamic settings. Kash et al. [13] study a fair division problem with divisible goods where agents with Leontief preferences arrive over time and goods must be irrevocably allocated. Benade et al. [5] study the problem of allocating indivisible goods that arrive in an online manner, again with irrevocable decisions. He et al. [11] generalize this to a setting where reallocation is allowed but expensive.

1.5 Applications

In this section, we discuss several key applications of the dynamic necklace splitting problem.

1.5.1 Fair Hash Maps

A core application of dynamic necklace splitting is designing practical fair hash maps. In addition to ensuring fairness, such data-informed hash maps can even be faster than traditional hash maps [14, 17]. Shahbazi et al. [19] use the static necklace splitting problem

to design hash maps that satisfy group fairness. However, making dynamic updates to these hash maps is infeasible without a dynamic solution to the necklace splitting problem.

For a concrete application, consider the use of fair hash maps to maintain user information on a social network. Whenever new users join the network, their data needs to be efficiently added to the hash map, and when users leave, their data needs to be efficiently removed. As users' attributes change, their data needs to be efficiently updated (corresponding to relocation of beads in a necklace).

Another concrete application involves table joins in data lakes. Consider an organization seeking to share its data with third parties. The data is stored in a data lake and includes sensitive information as primary keys for certain tables. To protect patient privacy, the data needs to be hashed, which introduces the risk of hash collision. A fair hashing scheme must be used to ensure any resulting errors when joining on the hashed columns do not disproportionately impact a specific demographic group [19]. For this system to be efficient, it needs to be easy to update the hashes as the data changes.

1.5.2 Load-Balancing

Consider the problem of load-balancing among multiple servers, where tasks need to be completed in a specific order. Each agent corresponds to a server, and each bead color corresponds to a different type of task. We want to spread the load across the servers evenly, but we also want to minimize communication costs [12]. This necessitates having as few cuts as possible.

If the order of tasks needs to be modified, new tasks need to be added, or existing tasks need to be canceled, we would want to be able to update the computation plan without recreating it from scratch. As a special case, if computation has already begun, we can still perform dynamic updates that don't affect the already-completed tasks using dynamic necklace splitting.

1.5.3 Bucketization

Another key application is ensuring minority representation in bucketization. For example, when partitioning data for machine learning tasks, it is crucial that the training data is accurately represented by the test data for reliable results [18]. Intentionally designing the bucketization process to maintain fairness helps prevent downstream fairness issues [15, 16]. Here, each agent represents a bucket, and the color of each bead represents the grouping attribute. If the "red" data is more sparse than the "blue" data, it is important that the relative proportions are preserved to have accurate results.

2 Offline Algorithm

We present a simplified version of the algorithm of Shahbazi et al. [19] here, as it is used as a subroutine in our algorithms for the $n = 2$ case. Without loss of generality, we refer to the colors as red and blue. Initialize a doubly linked list L such that, for each $j \in [0, m-1]$, $L[j]$ is the number of red beads in the substring $[j, j + m/k - 1]$. Initialize a hash set H that contains all indices j where $L[j] = m_1/k$. This takes $O(m)$ time. By the discrete intermediate value theorem, there is a substring with m_1/k red beads and m_2/k blue beads, so H is nonempty. We remove the smallest index j from H and allocate the sublist of length m/k beginning at j in L to the first agent. We remove all indices within $m/k - 1$ beads of j in L from H, update the values of L for $m/k - 1$ beads preceding j, add new indices to H

as necessary, and update the element preceding j in L to point to the next unallocated bead. We repeat this process $k-1$ more times to allocate the remaining beads. Each step takes $O(m/k)$ time, so in total, the algorithm takes $O(m)$ time. When we use this algorithm as a subroutine, we will often run it on a subsequence of length $k'm/k$ with only k' agents. For this special case, the algorithm takes only $O(k'm/k)$ time.

As shown by Alon and Graur [2], this algorithm produces at most $2(k-1)$ cuts. It is also known that no algorithm can guarantee fewer than $2(k-1)$ cuts.[2]

We now walk through a sample execution of the offline algorithm.

▶ **Example 1.** Suppose we are splitting the following necklace between three agents.

Each agent needs to receive two red beads and two blue beads. The first interval with the correct number of beads of each color is assigned to the first agent.

The second agent receives the first, second, seventh, and eighth beads, and the third agent receives the remaining beads.

Each agent receives two red beads and two blue beads in total. ⌟

3 Relocation with Two Colors

We now study the dynamic update of relocation with two colors.

3.1 Adjacent Indices

First, we consider the case where we restrict ourselves to relocating a bead to an adjacent index. Notice that this is equivalent to swapping two adjacent beads. For brevity, we will refer to relocation from j to $j+1$ as swapping beads $S[j]$ and $S[j+1]$.

We now show how we can maintain a valid set of cuts after swapping two adjacent beads $S[j]$ and $S[j+1]$. If they are the same color or belong to the same agent, we can maintain the same set of cuts P. The only interesting case is when $S[j]$ and $S[j+1]$ are different colors and belong to different agents. Let A_1 be the owner of $S[j]$ and A_2 the owner of $S[j+1]$. Consider the subsequence of beads $S' \subseteq S$ belonging to either A_1 or A_2. We remove from P the cuts adjacent to both A_1 and A_2. We run the offline algorithm on S' (in $O(m/k)$ time) and add the (at most two) new cuts to P. We will henceforth refer to this procedure as Swap.

Next, we prove upper bounds on the number of cuts in P after the update. We start with a relatively trivial bound.

▶ **Proposition 2.** *After r swaps, Swap produces a set of cuts of size at most $2(k-1)+r$.*

Proof. During each update, there is at least one cut adjacent to both A_1 and A_2. We remove this cut and add at most two cuts. Thus, with each update, we add at most one new cut. After r swaps, we will have at most $2(k-1)+r$ cuts. ◀

[2] See Appendix A for proof.

In fact, we can show that no extra cuts are needed.

▶ **Theorem 3.** *Swap produces a set of cuts of size at most* $2(k-1)$ *and has time complexity* $O(m/k)$.

Proof. For the first update, we can assume that the original allocation has the structure of one given by an execution of the offline algorithm (up to order of interval selection). In addition, we will show that each update preserves that invariant, allowing us to use those properties in the proof of the bound.

Assume without loss of generality that A_1 is allocated beads before A_2. Notice that the order in which agents are assigned beads can, in some cases, be altered without affecting the resulting allocations. In particular, since A_1 shares a common boundary with A_2, there can be no agent whose allotted set of beads "encloses" that of A_1 but not A_2. Thus, we can conceptually alter the order of execution of the original offline algorithm without loss of generality such that A_1 was allocated beads immediately before A_2. The only changes we have to make are in the cases where a cut is added during A_1's turn that is adjacent to an agent A_3 whose beads are allocated after A_1 but before A_2. In those cases, we add the cut during A_3's turn instead.

Next, notice that each agent has two cuts allotted in the offline bound of $2(k-1)$ cuts (with the exception of the final agent, who has none). Thus, during each update, we can simply remove the cuts corresponding to A_1 (resulting in at most $2(k-2)$ total cuts) and add new cuts by rerunning the offline algorithm for the beads originally allocated to A_1 and A_2 (returning to the bound of $2(k-1)$). The only special case is when A_1 is not enclosed within A_2 and has no other adjacent agent at the time of allocation. In that case, we only remove one of two cuts corresponding to A_1, but we may add two cuts back. However, if we add two cuts, we can note that A_2 only has one corresponding cut, so A_1 can "donate" one cut to A_2, maintaining the property that each non-final agent has at most two corresponding cuts.

Finally, we note that none of the steps in the update process violate the stated invariant that the allocation is one that could have been generated by an execution of the offline algorithm (up to order of interval selection), concluding our proof. ◀

3.2 Nonadjacent Indices

Next, we consider the case where beads need to be relocated an arbitrary distance away. Let A_1 be the owner of $S[j_1]$ and A_2 the owner of $S[j_2]$.

3.2.1 The Path Algorithm

If the distance between A_1 and A_2 in the neighborhood graph G is sufficiently small, we can efficiently perform relocations without adding any extra cuts. First, we find a shortest path between A_1 and A_2 in G. Let k' be the number of agents along that path. Then, we move $S[j_1]$ to index j_2. Finally, we rerun the offline algorithm on the substring belonging to the k' agents. We will henceforth refer to this procedure as Path.

Note that the number of edges in G is at most the number of cuts. Thus, finding a shortest path takes $O(k)$ time using breadth-first search, giving a running time of $O(k + k'm/k)$ for Path. In many practical applications, the distance that beads need to be relocated is relatively small, making Path very efficient.[3]

[3] See Appendix B for a theoretical analysis of the lengths of paths in G.

▶ **Theorem 4.** *Path produces a set of cuts of size at most* $2(k-1)$ *and has time complexity* $O(k + k'm/k)$.

Proof. For the first update, we can assume that the original allocation has the structure of one given by an execution of the offline algorithm (up to order of interval selection). In addition, we will show that each update preserves that invariant, allowing us to use those properties in the proof of the bound.

Each agent A' in the path shares a common boundary with the next agent in the path, A'', so there can be no agent whose allotted set of beads "encloses" that of A' but not A''. Without loss of generality, this enables us to conceptually alter the order of execution of the original offline algorithm such that A' and A'' were assigned beads in consecutive turns. Following the same logic, we can consider all of the k' agents to have had consecutive turns.

Next, notice that each agent has two cuts allotted in the offline bound of $2(k-1)$ cuts (with the exception of the final agent, who has none). Thus, during each update, we can simply remove the cuts corresponding to the k' agents (resulting in at most $2(k-k')$ total cuts) and add new cuts by rerunning the offline algorithm for the beads originally allocated to those agents (returning to the bound of $2(k-1)$).

Finally, we note that none of the steps in the update process violate the stated invariant that the allocation is one that could have been generated by an execution of the offline algorithm (up to order of interval selection), concluding our proof. ◀

3.2.2 The ColorPath Algorithm

Without loss of generality, assume $S[j_1]$ is red. We now construct a more intricate neighborhood graph, G'. Unlike G, this graph is directed and weighted. For each pair of agents A' and A'', there is an edge from A' to A'' if the two agents possess adjacent beads. If any such pair of beads has a red bead on the side of A', we call the edge *good*, and its weight is 0; otherwise, we call it *bad*, and its weight is 1.

First, we find a shortest path from A_2 to A_1 in G'. Let k' be the number of agents along that path. Then, we move $S[j_1]$ to index j_2. Next, for each good edge, we move the corresponding red bead across the adjacent cut. For each subpath of bad edges, we rerun the offline algorithm on the substring corresponding to the subpath. We will henceforth refer to this procedure as ColorPath.

Note that the number of edges in G' is at most twice the number of cuts. Finding a shortest path takes $O(k)$ time using Dial's algorithm [8], giving a running time of $O(k+k'm/k)$ for ColorPath. Asymptotically, this is the same as Path, but in practice, ColorPath will likely be faster.

▶ **Proposition 5.** *ColorPath produces a set of cuts of size at most* $2(k-1)$ *and has time complexity* $O(k + k'm/k)$.

3.2.3 The Fence Algorithm

If the number of agents k' in the path is large, we may instead wish to add extra cuts in return for a reduced running time. After moving $S[j_1]$ to index j_2, instead of rerunning the offline algorithm, we simply add (at most two) cuts around it as necessary. We will henceforth refer to this procedure as Fence.

The main disadvantage of Fence is that the resulting allocation is not guaranteed to be one that could have been generated by an execution of the offline algorithm. This means that, after a single execution of Fence, the guarantees of Theorems 3 and 4 (and Proposition 5) no

longer hold.[4] As such, once Fence has been used, any further relocations must be performed using Fence as well. Furthermore, once the guaranteed number of cuts has passed some user-determined tolerance level, the whole necklace should be reallocated from scratch using the offline algorithm.

Suppose $2k$ extra cuts are permitted to be added via relocations before reallocation from scratch is required. Consider the sequence of relocations starting from the first execution of Fence to the first relocation that leads to over $2k$ extra cuts. The amortized running time of Fence (including the time needed for the reallocation from scratch) is then

$$O(1) + \frac{O(m)}{\Omega(k)} = O\left(\frac{m}{k}\right).$$

As such, Fence may be preferred over Path and ColorPath if k' is frequently large.

We can also adapt Fence to the case of $n > 2$ in a straightforward way. Using the offline algorithm of Alon and Graur [2, Theorem 5] (which has a running time of $O(m)$), we can guarantee at most $n(k-1)\lceil 4 + \log_2(3k \max_{i\in[n]} m_i)\rceil + 2r$ cuts after r relocations. If $2kn$ extra cuts are permitted to be added before reallocation from scratch is required, the amortized running time is then

$$O(1) + \frac{O(m)}{\Omega(kn)} = O\left(\frac{m}{kn}\right).$$

▶ **Proposition 6.** *After r relocations, Fence produces a set of cuts of size at most $2(k+r-1)$ if $n = 2$ and $n(k-1)\lceil 4+\log_2(3k \max_{i\in[n]} m_i)\rceil + 2r$ if $n > 2$ and has time complexity $O\left(\frac{m}{kn}\right)$.*

4 Batch Relocation with Two Colors

To make relocation more efficient, we can perform batch updates. Instead of moving a single bead, we will move m' beads (of the same color)[5] from indices $j_1^1, \ldots, j_1^{m'}$ to indices $j_2^1, \ldots, j_2^{m'}$. Let $\mathbf{A_1}$ and $\mathbf{A_2}$ be the corresponding sets of agents. If there are any agents in both sets, we can remove them from both. Let k'' be the number of agents remaining. Let m'' be the number of beads that need to change owners.

We construct a flow network $G'' = (V'', E'')$ from G as follows. For each node in V, there is a corresponding node in V''. In addition, there is a source s and sink t. For each node $u \in V'' \setminus \{s, t\}$, let $\Delta(u)$ denote the net change in the number of beads that the corresponding agent owns. For each node u corresponding to an agent in $\mathbf{A_1}$, there is an edge (s, u) with capacity $c_{su} \coloneqq -\Delta(u)$. For each node u corresponding to an agent in $\mathbf{A_2}$, there is an edge (u, t) with capacity $c_{ut} \coloneqq \Delta(u)$. For each edge in E, there is a corresponding edge (u, v) in E'' with infinite capacity.

Define an *active* node to be one with positive incoming and outgoing flow. We want to find a max flow (of value m'') such that the number of active nodes is minimized. We will henceforth refer to this problem as MinNodeMaxFlow.

▶ **Proposition 7.** *MinNodeMaxFlow is NP-complete.*[6]

[4] Proposition 2 can technically still be made to work with some adjustments. After usage of Fence, a modified Proposition 2 would guarantee that the number of cuts increases by one with each execution of Swap.

[5] Our techniques can be adapted in a straightforward manner to work with beads of both colors being relocated simultaneously, but for ease of exposition, we restrict ourselves to a single color here.

[6] See Appendix C for proof.

We now describe the construction of a spanning tree of G that will allow us to approximately solve MinNodeMaxFlow. Consider the process of generating the neighborhood graph G while the offline necklace splitting algorithm is being run. When a new node is added, we initially assign it a *level* of 1. If the corresponding agent's interval encloses any other agents' intervals, then we increment the levels of all the nodes corresponding to agents with enclosed intervals.

For example, the node corresponding to the first agent allocated beads is initially on level 1. If the second agent's interval encloses that of the first, the first agent's node is moved to level 2. If the third agent's interval encloses that of the second, then we have the first agent's node on level 3 and the second agent's node on level 2.

We continue the process of updating the levels until all agents have been assigned their intervals and the graph G is finalized. Then, we construct a new graph by removing all edges between nodes on the same level, other than those on level 1. We will denote this *neighborhood tree* by T.

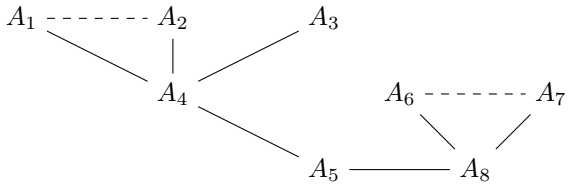


Figure 1 An example of a neighborhood tree. The dashed lines indicate additional edges present in the corresponding neighborhood graph.

▶ **Proposition 8.** *T is a spanning tree of G.*

Proof. First, we prove that T is connected. G is trivially connected since the necklace it represents is a contiguous collection of beads. We need to show that the edges removed from G to construct T are unnecessary for connectivity. Consider any edge removed. By construction, neither endpoint is on level 1. Thus, both endpoints are joined to a node on the previous level corresponding to an enclosing agent, and consequently, they remain connected.

Next, we prove that T is acyclic. Assume for contradiction that there exists a cycle in T. Let ℓ be the lowest level represented in the cycle. Consider a node u in the cycle on level ℓ. Both of its neighbors in the cycle must be on level $\ell - 1$. However, by construction, each node is joined to at most one node on the level below it. We have a contradiction, so T must be acyclic. Therefore, T is a spanning tree of G. ◀

We now describe how we can efficiently produce an approximate solution to MinNodeMaxFlow using the properties of T. We construct a flow network T' from T analogous to G''. The only difference from G'' is the lack of edges between nodes on the same level (other than level 1).

▶ **Lemma 9.** *The number of active nodes in a solution to MinNodeMaxFlow on T' is at most twice the number of active nodes in a solution on G''.*

Proof. Consider an optimal solution to MinNodeMaxFlow on G''. Assume, without loss of generality, that there is no pair of nodes u and v such that both (u, v) and (v, u) have positive flow. Let k' be the number of active nodes. For each edge (u, v) with positive flow where u and v are on the same level (other than level 1), we can remove the flow and instead add equal flow to (u, w) and (w, v), where w is the unique node on the previous level joined

to both u and v. If we replicate the resulting flow on T', we have a feasible solution to MinNodeMaxFlow on T'.

Since each node is joined to at most two nodes on the same level, there are at most k' pairs of adjacent active nodes on the same level. In the worst case, we add k' active nodes to the solution, resulting in $2k'$ active nodes. The optimal solution to MinNodeMaxFlow on T' must have at most as many active nodes as this feasible solution, so it also has at most $2k'$ active nodes. Therefore, the number of nodes is at most twice the number of active nodes in an optimal solution on G''. ◀

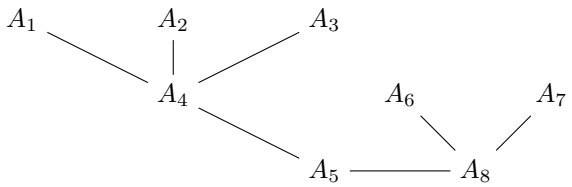
▶ **Lemma 10.** *MinNodeMaxFlow can be solved in $O(k'' \log k'' + k')$ time on T'.*

Proof. The edges from the source s and to the sink t must all be saturated to produce a max flow, so we can ignore both nodes. The remaining subgraph has structure identical to T, and is thus a tree. We sort the sources/sinks by their level in descending order and initialize a linked list of pointers to them in that order. Consider the set of nodes with excess (potentially negative) flow on the highest level (other than level 1). At first, these are all sources/sinks. Each of these nodes has a unique edge leading to the level below. Any valid flow must have flow on these edges. We place the appropriate amount of flow on each of these edges to satisfy the demands of the sources/sinks. Then we consider the new set of nodes with excess flow on the highest level (other than level 1) and repeat the process, updating the linked list accordingly. We terminate once all nodes not on level 1 have no excess flow. In each step, the highest level with excess flow decreases, so this process must terminate.

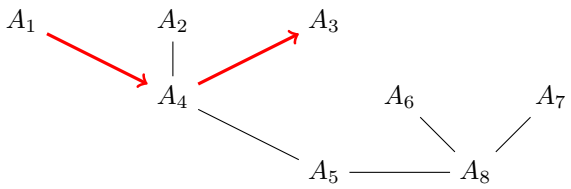
If any nodes with excess flow remain, they are on level 1. Consider the leftmost such node. We put flow on each edge to the right until the current node has no excess flow. Then we find the new leftmost node with excess flow and repeat the process. In each step, the number of nodes with excess flow decreases, so eventually no nodes will have excess flow, at which point we terminate.

Sorting the initial list takes $O(k'' \log k'')$ time. In the first phase, we process only the nodes in the solution, and each in constant time, giving a running time of $O(k')$ for the first phase. In the second phase, assuming the relative position of each node on level 1 is stored when T is constructed, we can find the leftmost node in $O(k')$ time, and after that, we only process nodes in the solution, each in constant time. This gives us a running time of $O(k')$ for the second phase. Overall, we have a running time of $O(k'' \log k'' + k')$ for solving MinNodeMaxFlow on T'. ◀

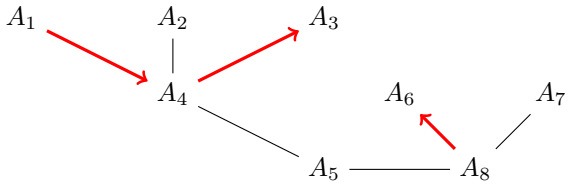
▶ **Example 11.** Consider the following neighborhood tree.



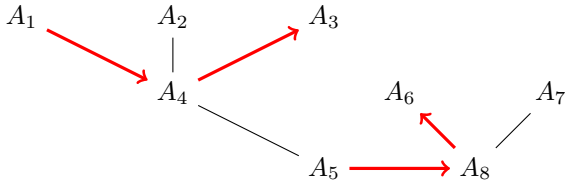
Suppose A_1 and A_5 are sources and A_3 and A_6 are sinks, all with demands of 1. At first, A_1 and A_3 are the highest nodes with excess flow. We put flow on the edges (A_1, A_4) and (A_4, A_3).



Now the highest node with excess flow is A_6, so we put flow on (A_8, A_6).



Continuing, we put flow on (A_5, A_8).



All nodes are now satisfied, so the algorithm terminates. ⌟

After MinNodeMaxFlow has been solved on T', we can return to G to perform a heuristic optimization. For each active node u that is neither a source/sink nor sending flow to the level below, consider the subgraph consisting of its active neighbors on the level above. If that subgraph is connected, then we can remove u from the solution. Finding all such subgraphs takes $O(k)$ time since we consider each edge a constant number of times. Checking if the subgraphs are connected takes $O(k)$ time in total. Overall, this heuristic pruning process adds $O(k)$ additional time.

Finally, we rerun the offline necklace splitting algorithm on the substring belonging to the agents corresponding to the active nodes. We will henceforth refer to this procedure as BatchPath. We have an overall running time of $O\left(\frac{1}{m'}\left(k'' \log k'' + k' + \frac{k'm}{k}\right)\right) = O\left(\frac{k'' \log k''}{m'} + \frac{k'm}{km'}\right)$ per bead for BatchPath.[7] Since $k'' \leq m'$, we also have the looser but more intuitive running time of $O\left(\log k + \frac{k'm}{km'}\right)$. Thus, there is only an $O(\log k)$ overhead when using BatchPath compared to Path, without accounting for the $m'\times$ speedup from batching updates.

▶ **Theorem 12.** *BatchPath produces a set of cuts of size at most* $2(k-1)$ *and has time complexity* $O\left(\log k + \frac{k'm}{km'}\right)$.

5 Relocation with Multiple Colors and High Cut Density

Next, we consider the case of relocation for general $n \geq 2$ with the additional restriction that $m = nk$. Note that using the corresponding offline algorithm of Alon and Graur [2,

[7] Alternatively, $O\left(\frac{k'' \log k''}{m'} + \frac{k'm}{km'} + \frac{k}{m'}\right)$ per bead if pruning is used.

Proposition 1], the initial solution is guaranteed to have exactly $n(k-1)$ cuts (some may be redundant). This initial solution is produced in $O(m)$ time. To enable efficient implementations for our dynamic algorithms, we maintain a two-dimensional array indexed by agent and color that points to the corresponding bead in S.

5.1 Adjacent Indices

Suppose we need to swap a red bead (on the left) and blue bead (on the right) between A_1 and A_2, respectively. There are three possible (nontrivial) cases (Table 3).

Table 3 The possible cases for the DenseSwap algorithm.

Case	Necklace
1	| R | B | → | B | R |
2	| R | B G → | B | R G
3a	B R | B → B B | R → B | B R
3b	G R | B → G B | R → G | B R

In the first case, there is a cut to the left of the red bead and to the right of the blue bead (or it is at the end of the necklace). This is the simplest case – we swap the beads and let the agents retain ownership of their original beads. No cuts need to be adjusted.

In the second case, there is a cut on the left, but not the right. We swap the beads and do not adjust any cuts. A_2 takes possession of the left interval, and A_1 takes possession of the right interval. As part of the right interval, A_1 gains at least one unneeded extra bead (the green one). Each extra bead lacks a cut to its left, so they must be the first of their colors. Thus, the original beads of A_1 are somewhere to the right in the necklace. Furthermore, since each of the original beads is not the first of its color, they each start their own intervals. A_2 takes ownership of each of these intervals and repeats the process with each of them if they have multiple beads, continuing to attempt to re-satisfy the fairness constraint. Since the necklace has exactly $n-1$ pairs of adjacent beads without a cut, this results in at most n ownership exchanges throughout the procedure.

In the third case, there is no cut on the left. We swap the beads and move the cut left by one bead (to ensure that the set of cuts maintains the structure of one generated by the offline algorithm). A_1 and A_2 retain the left and right intervals, respectively. Similarly to the second case, to maintain the fairness constraint, at most $n-1$ intervals will have to be exchanged between the two agents.

We will henceforth refer to this procedure as DenseSwap. The two-dimensional array allows us to perform each ownership exchange in constant time, giving a running time of $O(n)$ for DenseSwap.

▶ **Theorem 13.** *DenseSwap produces a set of cuts of size exactly $n(k-1)$ and has time complexity $O(n)$.*

5.2 Nonadjacent Indices

We can generalize the ideas from the previous section to design an algorithm for arbitrary relocation. In addition to the two-dimensional array, we maintain for each color a red–black

tree tracking the indices of the beads of that color in S. Suppose we need to relocate a red bead. There are four possible cases (Table 4).

Table 4 The possible cases for the DenseJump algorithm.

Case	First of its color?
1	✓ → ✓
2	✗ → ✗
3	✓ → ✗
4	✗ → ✓

In the first case, the bead to be relocated begins as the first of its color and remains so. We move the bead to the desired position, and if the interval is owned by a different agent, we re-satisfy the fairness constraint by exchanging at most n intervals between the two agents, as in the previous section.

In the second case, the bead to be relocated begins not as the first of its color and remains so. We move the bead and its corresponding cut to an arbitrary boundary to make it a singleton interval, and if the two affected intervals are not owned by the same agent, we re-satisfy the fairness constraint by exchanging at most $n-1$ intervals between the two agents. Then, we move the bead and its corresponding cut to the desired position, and if the destination interval does not belong to the owner of the bead, we re-satisfy the fairness constraint by exchanging at most n intervals between the two agents.

In the third case, the bead begins as the first of its color but does not remain so. We relocate the bead directly to the left of the second red bead via Case 1, and then we relocate the second red bead to the desired position via Case 2. At most $3n-1$ intervals are exchanged throughout this case.

In the fourth case, the bead begins not as the first of its color but becomes the first. We relocate the bead and its corresponding cut directly to the right of the first red bead via Case 2, and then we relocate the first red bead to the desired position via Case 1. At most $3n-1$ intervals are exchanged throughout this case.

We will henceforth refer to this procedure as DenseJump. Updating the red–black tree in Case 2 takes $O(k)$ time if the rank of the bead changes. Otherwise, all updates to the tree take constant time. This gives us a total running time of $O(k+n)$ for DenseJump.

▶ **Theorem 14.** *DenseJump produces a set of cuts of size exactly $n(k-1)$ and has time complexity $O(k+n)$.*

6 Insertion and Deletion

In this section, we study the complementary dynamic updates of insertion and deletion with two colors. For insertion, we simply give α beads to each agent and then relocate the αk beads to the desired indices using BatchPath. For deletion, we delete the desired beads and then relocate beads from agents with surpluses to those with deficits, again using BatchPath. Both insertion and deletion take $O\left(\frac{k''\log k''}{\alpha k}+\frac{k'm}{\alpha k^2}\right)=O\left(\frac{\log k}{\alpha}+\frac{k'm}{\alpha k^2}\right)$ time per bead using this approach.

▶ **Theorem 15.** *There is an algorithm for insertion and deletion that produces a set of cuts of size at most $2(k-1)$ and has time complexity $O\left(\frac{\log k}{\alpha}+\frac{k'm}{\alpha k^2}\right)$.*

We now show how the running time can be more precisely understood in the special case of $\alpha = 1$ by bounding the value of k'' as k grows. Suppose each of the beads is associated with a scalar value, such that the position of a bead in the necklace is determined by the rank of its value. This is the case, for example, when designing fair hash maps [19]. If the bead values are drawn independently from the same distribution $\mathcal{D}$, then the distribution of the rank of a new sample among the existing beads in the necklace is simply the uniform distribution. As such, it is well motivated to consider beads as being inserted into the necklace uniformly at random. Equivalently, agents to receive beads are chosen uniformly at random. In the absence of any further information, it is justified to assume that beads to be deleted are chosen uniformly at random as well.

Ideally (to reduce running time), we want to have as many agents receive exactly one bead as possible, since k'' is the number of agents that do not receive exactly one bead. We will show that this is likely to be the case. Assume $k > 101$. For each $i \in [k]$, let X_i be a Bernoulli random variable indicating if the number i is not chosen exactly once when sampling k values from the discrete uniform distribution on $[k]$. Let $X = \sum_i X_i$. We have $k(1 - 1.005/e) < \mathbb{E}[X] < k(1 - 1/e)$. For the variance, we have the following.

$$\begin{aligned}
\operatorname{Var}(X) &= \operatorname{Var}(k - X) \\
&= \sum_i \operatorname{Var}(1 - X_i) + \sum_{i,j=1,i\neq j}^{k} \operatorname{Cov}(1 - X_i, 1 - X_j) \\
&= k \operatorname{Var}(1 - X_1) + k(k-1) \operatorname{Cov}(1 - X_1, 1 - X_2) \\
&< \frac{1.005k}{e} - \frac{k}{e^2} + k(k-1)\left(\frac{(k-1)(k-2)^{k-2}}{k^{k-1}} - \frac{(k-1)^{2k-2}}{k^{2k-2}}\right) \\
&= \frac{1.005e-1}{e^2}k + \frac{k^{k-1}(k-1)^2(k-2)^{k-2} - (k-1)^{2k-1}}{k^{2k-1}} \\
&< \frac{1.005e-1}{e^2}k + \frac{k^{2k-1} - (k-1)^{2k-1}}{k^{2k-1}} \\
&= \frac{1.005e-1}{e^2}k + 1 - \left(1 - \frac{1}{k}\right)^{2k-1} \\
&< \frac{1.005e-1}{e^2}k + 1 - \frac{0.995}{e^2}
\end{aligned}$$

Then, by the Chebyshev–Cantelli inequality, we have

$$\Pr(X \geq \lambda k) < \frac{\frac{1.005e-1}{e^2}k + 1 - \frac{0.995}{e^2}}{\frac{1.005e-1}{e^2}k + 1 - \frac{0.995}{e^2} + \left(\lambda k + \frac{1}{e} - 1\right)^2}.$$

If we set $\lambda = 1 - 1/e$, then we have $k'' < (1 - 1/e)k$ with probability greater than 99.3%.

7 Approximate Necklace Splitting

In this section, we design an algorithm for approximate necklace splitting with two colors that works in both the static and dynamic settings. The algorithm is efficient when the number of agents k is small, for example, $k = o(m)$.

7.1 Static Setting

We construct a red–black tree $\mathcal{T}$ over the necklace S with respect to the order of the beads (i.e., we assume that $S[j_1] < S[j_2]$ if $j_1 < j_2$). Let S_1 be the subsequence consisting of red

beads and S_2 the subsequence consisting of blue beads with $|S_1| = m_1$ and $|S_2| = m_2$. Let $\varepsilon \in (0, 1)$ be a parameter specified by the user. Let $\mathcal{I}^0 = \varnothing$. We run the following subroutine for k iterations.

In the jth iteration, for each color $i \in \{1, 2\}$, we use $\mathcal{T}$ to obtain a uniform random sample $\mathcal{S}_i^j \subseteq S_i \setminus \mathcal{I}^{j-1}$ of cardinality $O\big((k-j+1)^2 2^{2k} \varepsilon^{-2} \log(2km)\big)$. Let $\mathcal{S}^j = \mathcal{S}_1^j \cup \mathcal{S}_2^j$. We execute one iteration of the exact offline algorithm with $k - j + 1$ agents on $S \setminus \mathcal{I}^{j-1}$. Let I_j be the interval returned by the algorithm. We add the (at most) two corresponding cuts. Let $\mathcal{I}^j = \mathcal{I}^{j-1} \cup \{I_j\}$.

After the kth iteration, the resulting set of cuts is returned. We will henceforth refer to this procedure as ApproxStatic.

▶ **Theorem 16.** *ApproxStatic produces an approximate solution with at most* $2(k-1)$ *cuts, such that each agent gets at least* $(1-\varepsilon)\frac{m_i}{k}$ *and at most* $(1+\varepsilon)\frac{m_i}{k}$ *beads of color* i *with probability at least* $1 - \frac{1}{m}$*. It has time complexity* $O\big(m + k^3 2^{2k} \varepsilon^{-2} (\log m)^2\big)$.[8]

7.2 Dynamic Setting

We now show how to adapt ApproxStatic to the dynamic setting. Our algorithm can handle any type of dynamization, including relocation, insertion, and deletion.

Notice that the crux of ApproxStatic relies on a small set of samples (given sufficiently small k). Hence, if we ensure that sampling is performed efficiently under updates, we can have a dynamic algorithm for approximate necklace splitting with an update time that only depends logarithmically on m. Indeed, assume that the tree $\mathcal{T}$ is constructed in the preprocessing phase. It is known that a red–black tree can be updated in $O(\log m)$ time under insertion/deletion of an element. Thus, our algorithm for the dynamic necklace splitting problem consists of maintaining $\mathcal{T}$ under relocations/insertions/deletions (a relocation can be handled as a deletion followed by an insertion). When the user requests the set of cuts, we execute ApproxStatic as a subroutine using the updated $\mathcal{T}$. We will henceforth refer to this procedure as Approx.

When a batch of k updates is encountered, $\mathcal{T}$ is updated in $O(k \log m)$ time, and the new set of cuts can be constructed in $O\big(k^3 2^{2k} \varepsilon^{-2} (\log m)^2\big)$ time. The overall running time of Approx is thus $O\big(k^2 2^{2k} \varepsilon^{-2} (\log m)^2 + \log m\big)$ per bead. If we do not often need to produce the actual set of cuts, we only need $O(\log m)$ time per bead to maintain $\mathcal{T}$.

▶ **Theorem 17.** *Approx produces an approximate solution with at most* $2(k-1)$ *cuts, such that each agent gets at least* $(1-\varepsilon)\frac{m_i}{k}$ *and at most* $(1+\varepsilon)\frac{m_i}{k}$ *beads of color* i *with probability at least* $1 - \frac{1}{m}$*. It has time complexity* $O\big(k^2 2^{2k} \varepsilon^{-2} (\log m)^2 + \log m\big)$.

References

1 Noga Alon. Splitting necklaces. *Advances in Mathematics*, 63(3):247–253, March 1987. doi:10.1016/0001-8708(87)90055-7.

2 Noga Alon and Andrei Graur. Efficient splitting of necklaces. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *ICALP '21*, pages 14:1–14:17, Virtual (Glasgow, Scotland), July 2021. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.14.

[8] See Appendix D for proof.

3 Noga Alon and Douglas B. West. The Borsuk-Ulam theorem and bisection of necklaces. *Proceedings of the American Mathematical Society*, 98(4):623–628, 1986. doi:10.1090/s0002-9939-1986-0861764-9.

4 Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, November 1999. doi:10.1017/cbo9780511624216.

5 Gerdus Benade, Aleksandr M. Kazachkov, Ariel D. Procaccia, and Christos-Alexandros Psomas. How to make envy vanish over time. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, EC '18, pages 593–610, Ithaca, New York, USA, June 2018. ACM. doi:10.1145/3219166.3219179.

6 Sandeep N. Bhatt and Charles E. Leiserson. How to assemble tree machines (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 77–84, San Francisco, California, USA, May 1982. ACM. doi:10.1145/800070.802179.

7 Karol Borsuk. Drei Sätze über die n-dimensionale euklidische Sphäre. *Fundamenta Mathematicae*, 20(1):177–190, 1933. doi:10.4064/fm-20-1-177-190.

8 Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, November 1969. doi:10.1145/363269.363610.

9 Charles H. Goldberg and Douglas B. West. Bisection of circle colorings. *SIAM Journal on Algebraic Discrete Methods*, 6(1):93–106, January 1985. doi:10.1137/0606010.

10 Sariel Har-Peled. *Geometric Approximation Algorithms*, volume 173 of *Mathematical Surveys and Monographs*. AMS, June 2011.

11 Jiafan He, Ariel D. Procaccia, Alexandros Psomas, and David Zeng. Achieving a fairer future by changing the past. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, IJCAI-2019, pages 343–349, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2019/49.

12 Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, June 2017. doi:10.1016/j.jnca.2017.04.007.

13 Ian Kash, Ariel D. Procaccia, and Nisarg Shah. No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research*, 51:579–603, November 2014. doi:10.1613/jair.4405.

14 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504. ACM, May 2018. doi:10.1145/3183713.3196909.

15 Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM Computing Surveys*, 54(6):1–35, July 2021. doi:10.1145/3457607.

16 Dana Pessach and Erez Shmueli. A review on fairness in machine learning. *ACM Computing Surveys*, 55(3):1–44, February 2022. doi:10.1145/3494672.

17 Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. Can learned models replace hash functions? *Proceedings of the VLDB Endowment*, 16(3):532–545, November 2022. doi:10.14778/3570690.3570702.

18 Nima Shahbazi, Yin Lin, Abolfazl Asudeh, and V. Jagadish, H.˙Representation bias in data: A survey on identification and resolution techniques. *ACM Computing Surveys*, 55(13s):1–39, July 2023. doi:10.1145/3588433.

19 Nima Shahbazi, Stavros Sintos, and Abolfazl Asudeh. FairHash: A fair and memory/time-efficient hashmap. *Proceedings of the ACM on Management of Data*, 2(3):1–29, May 2024. doi:10.1145/3654939.

A Offline Algorithm is Optimal

▶ **Proposition 18.** *For every k, there exists a necklace that requires exactly $2(k-1)$ cuts.*

Proof. The $k = 1$ case is trivial, so we assume $k > 1$. Consider the necklace with $\frac{m}{2}$ red beads followed by $\frac{m}{2}$ blue beads. Suppose we have an optimal set of cuts of the necklace. The first interval in the resulting allocation consists of at most $\frac{m}{2k}$ red beads. Without loss of generality, assume this interval belongs to A_1. If it has exactly $\frac{m}{2k}$ beads, we move to the next interval (without loss of generality, owned by A_2). Else, we check if the first two intervals combined have more than $\frac{m}{2k}$ beads. If so, we can move the first cut to the right until the first interval has exactly $\frac{m}{2k}$ beads. To maintain the fairness constraint, we reassign all the red beads of A_1 in the rest of the necklace to A_2 (without adding any cuts). We then move to the next interval. Instead, if the first two intervals combined have at most $\frac{m}{2k}$ beads, we remove the cut between them and add one cut within another interval with red beads belonging to A_1, maintaining the overall number of cuts.

We repeat this process from both the left (red) and right (blue) ends of the necklace until we reach the center interval, which consists of $\frac{m}{2k}$ red beads and $\frac{m}{2k}$ blue beads. We have exactly $2(k-1)$ cuts, and since the number of cuts remained constant throughout this procedure, the original optimal set of cuts was of size $2(k-1)$. ◀

B Lengths of Paths in the Neighborhood Graph

In this section, we analyze the lengths of paths in G under loose conditions on the dynamic updates. We show that if relocation positions are drawn uniformly at random, we can bound the length of the path with high likelihood.

▶ **Proposition 19.** *The distance between two distinct nodes selected uniformly at random is at most $\lceil k/2 \rceil$ with probability at least 75%.*

We first state and prove a lemma that will be useful in the proof.

▶ **Lemma 20.** *The average distance between a node u and each other node in a connected graph with k nodes is at most $k/2$.*

Proof. Let d be the greatest distance between u and any other node. The existence of such a node implies the existence of nodes at distances $d-1$, $d-2$, etc. from u as well. The remaining $k-1-d$ nodes are at distance at most d from u by assumption. The average distance between u and each other node can be bounded as follows.

$$\frac{\sum_{\kappa=1}^{d} \kappa + (k-1-d)d}{k-1} \leq \frac{\sum_{\kappa=1}^{d} \kappa + \sum_{\kappa=d+1}^{k-1} \kappa}{k-1} = \frac{\sum_{\kappa=1}^{k-1} \kappa}{k-1} = \frac{k(k-1)}{2(k-1)} = \frac{k}{2}$$

◀

Proof of Proposition 19. First, we show that the connected graph with maximum average path length is the linear graph. Note that the average path length of any graph is at most that of one of its spanning trees, so we can restrict ourselves to trees. We prove the claim by induction.

For the base case, our claim trivially holds for $k < 3$. Assume the claim holds for some $k = \mu$. We show that it holds for $k = \mu + 1$. Consider the linear graph with $\mu + 1$ nodes. The subgraph induced by the first μ nodes has maximum average path length by assumption. The average distance between the last node and each other node is $k/2$, so by Lemma 20, it cannot be increased. Therefore, the claim holds for $k = \mu + 1$, and by induction, it holds for all k.

Next, we verify that it is possible for the neighborhood graph to be linear. One possible necklace that gives rise to a linear neighborhood graph is the following.

Finally, we analyze the lengths of shortest paths in a linear graph. Suppose G is a linear graph and two (distinct) nodes are selected uniformly at random. The distance between them follows a triangular distribution. The length is at most $\lceil k/2 \rceil$ with probability at least

$$1 - \frac{\sum_{\kappa=1}^{k/2-1} \kappa}{\sum_{\kappa=1}^{k-1} \kappa} = 1 - \frac{k(k-2)}{4k(k-1)} = \frac{3k-2}{4k-4} > \frac{3k-3}{4k-4} = 75\% .$$

◀

C NP-completeness of MinNodeMaxFlow

Proof of Proposition 7. First, we show that MinNodeMaxFlow is in NP. We are given a flow network and a limit κ on the number of active nodes as input and a proposed flow as a certificate. We can compute the value of a max flow on the graph (in polynomial time). Then we can easily verify whether the proposed flow has that value and satisfies the flow constraints and whether the number of active nodes is at most κ. MinNodeMaxFlow is in NP.

Next, we show that MinNodeMaxFlow is NP-hard, via reduction from Vertex Cover. Given an undirected graph $\Gamma = (V_\Gamma, E_\Gamma)$ and parameter κ, we want to determine whether Γ has a vertex cover of size at most κ. We construct an auxiliary, directed graph $\Gamma' = (V_{\Gamma'}, E_{\Gamma'})$ containing the same vertices in addition to a source node σ, a sink node τ, and an extra node $\overline{uv}$ for every edge $(u, v) \in E_\Gamma$. We replace each edge $(u, v) \in E_\Gamma$ with two infinite-capacity edges: $(\overline{uv}, u)$ and $(\overline{uv}, v)$. We add a unit-capacity edge from σ to each extra node $\overline{uv}$ and an infinite-capacity edge from each original node u to τ.

If there is a vertex cover of size κ in Γ, then there is a max flow on Γ' with $|E_\Gamma| + \kappa$ active nodes. Conversely, suppose there is a max flow on Γ' with κ active nodes. If any node $\overline{uv}$ is sending fractional flow to each of its neighbors, we can reroute its flow to go to just one neighbor without changing the number of active nodes. We would need to do this at most $|E_\Gamma|$ times, so we can always efficiently find an integral solution, given the initial solution. Once we obtain an integral solution, we have a corresponding vertex cover of size κ. Therefore, MinNodeMaxFlow is NP-hard. ◀

D Correctness and Time Complexity of ApproxStatic

▶ **Definition 21** (ε-sample [10, Chapter 5]). *Let $(X, \mathcal{R})$ be a set system. For any $\varepsilon \in [0, 1]$, a subset $C \subseteq X$ is an* ε-sample *for $(X, \mathcal{R})$ if for every $\rho \in \mathcal{R}$, we have*

$$\left| \frac{|\rho|}{|X|} - \frac{|C \cap \rho|}{|C|} \right| \leq \varepsilon .$$

Proof of Theorem 16. We can construct $\mathcal{T}$ in $O(m)$ time. We can retrieve an element from $\mathcal{T}$ in $O(\log m)$ time, so we can construct the set $\mathcal{S}^j$ in $O(|\mathcal{S}^j| \log m) = O\big(k^2 2^{2k} \varepsilon^{-2} (\log m)^2\big)$ time. Executing the exact offline algorithm for one iteration takes $O(|\mathcal{S}^j|) = O\big(k^2 2^{2k} \varepsilon^{-2} \log m\big)$ time. We run the subroutine for k iterations, so the total running time of ApproxStatic is $O\big(m + k^3 2^{2k} \varepsilon^{-2} (\log m)^2\big)$. It is clear that, by construction, the algorithm produces at most $2(k-1)$ cuts. What remains is to show that the resulting assignment of beads is approximately fair.

For any subset of beads X in the necklace, consider the set system $(X, \mathcal{R}_S)$ of VC dimension 2, where $\mathcal{R}_S$ is the family of all possible intervals (sets of contiguous beads) in

S. Then, with probability at least $1-\varphi$, a uniform random subset $C \subseteq X$ of cardinality $O\big(\varepsilon^{-2}\log\big(\varphi^{-1}\big)\big)$ is an ε-sample for $(X, \mathcal{R}_S)$ [4].

Let $\bar{\varepsilon} = \frac{\varepsilon}{2^k}$. By definition, the probability that $\mathcal{S}_i^j$ is a $\frac{\bar{\varepsilon}}{k-j+1}$-sample is at least $1 - \frac{1}{2km}$. We have that in the jth iteration, both $\mathcal{S}_1^j$ and $\mathcal{S}_2^j$ are $\frac{\bar{\varepsilon}}{k-j+1}$-samples for $(S_1 \setminus \mathcal{I}^{j-1}, \mathcal{R}_S)$ and $(S_2 \setminus \mathcal{I}^{j-1}, \mathcal{R}_S)$, respectively, with probability at least $1 - \frac{1}{km}$. Via a union bound over the iterations, with probability at least $1 - \frac{1}{m}$, we have that $\mathcal{S}_i^j$ is a $\frac{\bar{\varepsilon}}{k-j+1}$-sample for $(S_i \setminus \mathcal{I}^{j-1}, \mathcal{R}_S)$ for all $j \in [k]$ and $i \in \{1,2\}$.

We now prove by induction that for each iteration $j \in [k]$ and color $i \in \{1,2\}$, we have

$$\frac{1-(2^j-1)\bar{\varepsilon}}{k} \leq \frac{|S_i \cap I_j|}{|S_i|} \leq \frac{1+(2^j-1)\bar{\varepsilon}}{k}$$

if $\mathcal{S}_i^j$ is an $\frac{\bar{\varepsilon}}{k-j+1}$-sample for $S_i \setminus \mathcal{I}^{j-1}$. First we consider the base case: $j = 1$. By definition, in the first iteration, $\mathcal{S}_i^1$ is a $\frac{\bar{\varepsilon}}{k-j+1}$-sample for S_i. By the correctness of the exact offline algorithm on $\mathcal{S}^1$, for each $i \in \{1,2\}$, we have that $|\mathcal{S}_i^1 \cap I_1|/|\mathcal{S}_i^1| = 1/k$. Thus, we have

$$\frac{1-\bar{\varepsilon}}{k} \leq \frac{|S_i \cap I_1|}{|S_i|} \leq \frac{1+\bar{\varepsilon}}{k}\,.$$

For the inductive step, assume that the claim holds for all $j < \zeta$. We will show that it holds for $j = \zeta$. By the correctness of the exact offline algorithm on $\mathcal{S}^1$ we have that $|\mathcal{S}_i^\zeta \cap I_\zeta|/|\mathcal{S}_i^\zeta| = 1/(k-\zeta+1)$. Since $\mathcal{S}_i^\zeta$ is a $\frac{\bar{\varepsilon}}{k-\zeta+1}$-sample for $S_i \setminus \mathcal{I}^{\zeta-1}$, we have

$$\frac{1-\bar{\varepsilon}}{k-\zeta+1} \leq \frac{|(S_i \setminus \mathcal{I}^{\zeta-1}) \cap I_\zeta|}{|S_i \setminus \mathcal{I}^{\zeta-1}|} \leq \frac{1+\bar{\varepsilon}}{k-\zeta+1}\,.$$

Notice that $(S_i \setminus \mathcal{I}^{\zeta-1}) \cap I_\zeta = S_i \cap I_\zeta$, so we can rewrite this as

$$\frac{1-\bar{\varepsilon}}{k-\zeta+1}|S_i \setminus \mathcal{I}^{\zeta-1}| \leq |S_i \cap I_\zeta| \leq \frac{1+\bar{\varepsilon}}{k-\zeta+1}|S_i \setminus \mathcal{I}^{\zeta-1}|\,.$$

We first prove the left inequality of the claim.

$$\begin{aligned}
|S_i \cap I_\zeta| &\geq \frac{1-\bar{\varepsilon}}{k-\zeta+1}\big|S_i \setminus \mathcal{I}^{\zeta-1}\big| \\
&= \frac{1-\bar{\varepsilon}}{k-\zeta+1}\left(m_i - \sum_{j=1}^{\zeta-1}|S_i \cap I_j|\right) \\
&\geq \frac{1-\bar{\varepsilon}}{k-\zeta+1}\left(1 - \sum_{j=1}^{\zeta-1}\frac{1+\big(2^j-1\big)\bar{\varepsilon}}{k}\right)m_i \\
&= (1-\bar{\varepsilon})\left(1 - \frac{\bar{\varepsilon}}{k-\zeta+1}\sum_{j=1}^{\zeta-1}\big(2^j-1\big)\right)\frac{m_i}{k} \\
&= (1-\bar{\varepsilon})\left(1 - \frac{\big(-\zeta+2^\zeta-1\big)\bar{\varepsilon}}{k-\zeta+1}\right)\frac{m_i}{k} \\
&\geq (1-\bar{\varepsilon})\big(1-\big(-\zeta+2^\zeta-1\big)\bar{\varepsilon}\big)\frac{m_i}{k} \\
&\geq \big(1-\bar{\varepsilon}+\zeta\bar{\varepsilon}-2^\zeta\bar{\varepsilon}+\bar{\varepsilon}\big)\frac{m_i}{k} \\
&\geq \big(1-\big(2^\zeta-1\big)\bar{\varepsilon}\big)\frac{|S_i|}{k}
\end{aligned}$$

Similarly, we prove the right inequality.

$$\begin{aligned}
|S_i \cap I_\zeta| &\le \frac{1+\bar\varepsilon}{k-\zeta+1}\big|S_i \setminus \mathcal{I}^{\zeta-1}\big| \\
&= \frac{1+\bar\varepsilon}{k-\zeta+1}\left(m_i - \sum_{j=1}^{\zeta-1}|S_i \cap I_j|\right) \\
&\le \frac{1+\bar\varepsilon}{k-\zeta+1}\left(1 - \sum_{j=1}^{\zeta-1}\frac{1-\big(2^j-1\big)\bar\varepsilon}{k}\right)m_i \\
&= (1+\bar\varepsilon)\left(1 + \frac{\big(-\zeta+2^\zeta-1\big)\bar\varepsilon}{k-\zeta+1}\right)\frac{m_i}{k} \\
&\le (1+\bar\varepsilon)\big(1+\big(-\zeta+2^\zeta-1\big)\bar\varepsilon\big)\frac{m_i}{k} \\
&= \big(1+2^\zeta\bar\varepsilon-\zeta\bar\varepsilon+\big(2^\zeta-\zeta-1\big)\bar\varepsilon^2\big)\frac{m_i}{k} \\
&< \big(1+2^\zeta\bar\varepsilon-\zeta\bar\varepsilon+\bar\varepsilon\big)\frac{m_i}{k} \\
&\le \big(1+\big(2^\zeta-1\big)\bar\varepsilon\big)\frac{|S_i|}{k}
\end{aligned}$$

By induction, the claim holds. Finally, substituting $\bar\varepsilon = \frac{\varepsilon}{2^k}$, we have

$$\frac{(1-\varepsilon)|S_i|}{k} < |S_i \cap I_j| < \frac{(1+\varepsilon)|S_i|}{k}\,.$$

◀