# Locally Optimal Load Balancing

**Laurent Feuilloley**
laurent.feuilloley@ens-cachan.fr

École Normale Supérieure de Cachan, France

Helsinki Institute for Information Technology HIIT,
Department of Computer Science, Aalto University, Finland

**Juho Hirvonen**
juho.hirvonen@aalto.fi

Helsinki Institute for Information Technology HIIT,
Department of Computer Science, Aalto University, Finland

**Jukka Suomela**
jukka.suomela@aalto.fi

Helsinki Institute for Information Technology HIIT,
Department of Computer Science, Aalto University, Finland



**Abstract.** This work studies distributed algorithms for *locally optimal load-balancing*: We are given a graph of maximum degree $\Delta$, and each node has up to $L$ units of load. The task is to distribute the load more evenly so that the loads of adjacent nodes differ by at most 1.

If the graph is a path ($\Delta = 2$), it is easy to solve the *fractional* version of the problem in $O(L)$ communication rounds, independently of the number of nodes. We show that this is tight, and we show that it is possible to solve also the *discrete* version of the problem in $O(L)$ rounds in paths.

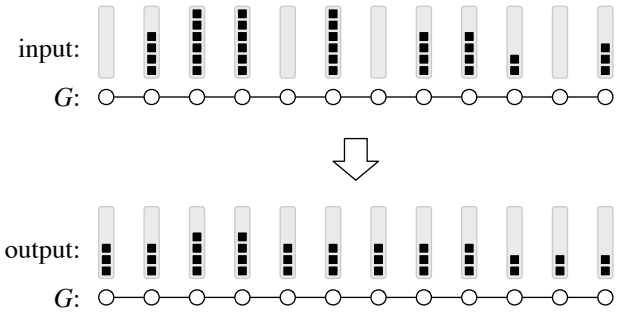
For the general case ($\Delta > 2$), we show that fractional load balancing can be solved in $\mathrm{poly}(L, \Delta)$ rounds and discrete load balancing in $f(L, \Delta)$ rounds for some function $f$, independently of the number of nodes.

# 1 Introduction

In this work, we introduce the problem of *locally optimal load balancing*, and study it from the perspective of distributed algorithms.

## 1.1 Locally optimal load balancing

In this problem, we are given a graph $G = (V, E)$, and each node has up to $L$ units of load. The task is to distribute load more evenly so that the loads of adjacent nodes differ by at most 1:




That is, we want to *smooth out* the load distribution, and find an *equilibrium* in which no edge can improve its load distribution by selfishly moving load between its endpoints.

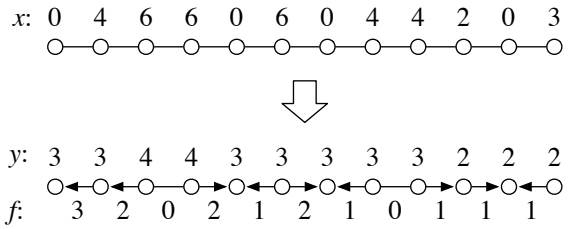
A bit more formally, in the load balancing problem we are given an input vector $x\colon V \to \{0, 1, \ldots, L\}$, and the task is to find an output vector $y\colon V \to [0, L]$ and a flow $f\colon E \to \mathbb{R}$ so that for each node $v \in V$ we have

$$y(v) = x(v) + \sum_{(u,v)\in E} f(u, v), \tag{1}$$

and for each edge $(u, v) \in E$ we have

$$|y(u) - y(v)| \leq 1. \tag{2}$$

Here is an illustration of the input and a feasible solution in the special case that $G$ is a path:




The problem comes in two natural flavours:

- *Discrete load balancing*: $y(v) \in \{0, 1, \ldots, L\}$, i.e., load units are indivisible.
- *Fractional load balancing*: $y(v) \in [0, L]$, i.e., load units can be divided.

## 1.2 Centralised algorithms

Both discrete and fractional load balancing can be solved easily with the following algorithm: Start with $y \leftarrow x$ and $f \leftarrow 0$. Then repeatedly pick an *unhappy edge* $(u, v) \in E$ with $y(u) \geq y(v) + 2$, and move one unit of load from $u$ to $v$. This algorithm clearly converges, as the potential function $\sum_v y(v)^2$ decreases by at least 2 in each step.

### 1.3 Local solutions and local algorithms

In the above centralised algorithm, we can think that each node $v$ has a pile of $y(v)$ tokens and we always move the topmost token. Then the height of a token decreases by at least one every time we move it; hence no individual token is moved more than $L$ times. This argument shows that there always exists a *local solution* in which the final position of a token is always within distance $L$ from its origin; that is, each token can stay in its radius-$L$ neighbourhood.

In this work we are interested if the problem can be solved with a *local algorithm*: is it possible to solve the problem so that we can compute the flow $f(u, v)$ for each edge $(u, v) \in E$ based on only the information that is available within distance $T$ from $(u, v)$ in graph $G$, for some $T$. Equivalently, we want to know if there is a (deterministic) distributed algorithm in the usual LOCAL model that solves the load balancing problem in $T$ *communication rounds*, or more succinctly, in *time* $T$.

We will assume that the input graph has maximum degree $\Delta$. We are interested in local algorithms with a running time of $T = T(L, \Delta)$ that may depend on the maximum load $L$ and maximum degree $\Delta$, but is independent on the number of nodes $n = |V|$. Such an algorithm could be used to solve load balancing even in infinitely large graphs, and it would be very easy to e.g. parallelise such algorithms, as each part of the output can be determined based on its local neighbourhood.

### 1.4 Smoothing with moving average

There is a special case that can be easily solved with a local algorithm in time $T = O(L)$: fractional load balancing in 2-regular graphs (cycles and infinite paths). We can simply calculate the moving average of the input loads with a window of size $\Theta(L)$. More concretely, each node gives a fraction $1/(2L + 1)$ of its input load to every node (including itself) in its radius-$L$ neighbourhood. This way the final loads of adjacent nodes differ by at most $L/(2L + 1) < 1/2$ units. The same strategy can be applied easily in, e.g., $d$-dimensional grids.

Among others, the present work seeks to answer the following questions:

- Is the running time of $O(L)$ optimal here, or could we solve it in time $o(L)$?
- Can we generalise this kind of smoothing algorithms to arbitrary graphs, and if so, what is the running time?
- Can we generalise this kind of smoothing algorithms to discrete load balancing?

### 1.5 Contributions

The contributions of this work are as follows. We start with a simple lower bound:

**Theorem 1.** *Load balancing requires $\Omega(L)$ rounds, even in the case of paths and cycles.*

Then we prove negative results for various algorithm families that have been used widely in the prior work. To this end, we define the following algorithm families:

- *Match-and-balance algorithms*: In each step, the algorithm finds a matching $M$ and balances the load (fully or partially) for each edge in $M$. More precisely, for each edge $(u, v) \in M$ with $y(u) > y(v)$, the algorithm increases the flow $f(u, v)$ by at most $(y(u) - y(v))/2$. For example, many natural distributed versions of the centralised algorithm from Section 1.2 are of match-and-balance type.
- *Careful algorithms*: In each round, for each edge $(u, v) \in E$, the algorithm increases or decreases $f(u, v)$ by at most $\text{poly}(L)$. All match-and-balance algorithms are also careful algorithms.

- *Oblivious algorithms*: The total amount of load moved from node $u$ to $v$ only depends on the initial load of $u$ and the distance between $u$ and $v$. For example, the moving average algorithm from Section 1.4 is oblivious.

We show that algorithms of any of these types cannot find a locally optimal load balancing efficiently (or at all):

**Theorem 2.** *Any match-and-balance algorithm takes $\Omega(L^2)$ rounds in the worst case, even in paths and cycles.*

**Theorem 3.** *Any careful algorithm takes $\Delta^{\Omega(L)}$ rounds in the worst case.*

**Theorem 4.** *There are no oblivious algorithms for infinite $d$-regular trees with $d \geq 3$.*

We then present the main contributions—local algorithms for load balancing. First, we show that we can circumvent the barrier of Theorem 2:

**Theorem 5.** *Discrete load balancing can be solved in time $O(L)$ in paths and cycles, with a deterministic local algorithm.*

**Corollary 6.** *The time complexity of both fractional and discrete load balancing in paths and cycles is $\Theta(L)$.*

Next we show that we can also circumvent the barriers of Theorem 3 and 4 for fractional load balancing—naturally, we have to design an algorithm that is neither oblivious nor careful:

**Theorem 7.** *Fractional load balancing can be solved in time* $\mathrm{poly}(L, \Delta)$ *in bounded-degree graphs with a deterministic local algorithm.*

Finally, we show that discrete load balancing can be solved locally, i.e., in time that is independent of $n$:

**Theorem 8.** *Discrete load balancing can be solved in time $T(L, \Delta)$, for some function $T$, in bounded-degree graphs with a deterministic local algorithm.*

Whether there is an *efficient* algorithm for discrete load balancing in the general case remains an open question.

## 2 Related work

There is a vast body of literature related to problems that are superficially similar to locally optimal load balancing. However, in many cases the primary goal is something else—for example, achieving a near-optimal global solution—and the algorithms just happen to also find a locally optimal solution.

Most of the previous solutions are inefficient. In particular, we are not aware of any solution that comes close to $O(L)$ for discrete load balancing on paths, or close to $\mathrm{poly}(L, \Delta)$ for fractional load balancing in general graphs. In prior work, the inefficiency typically stems from at least one of the following factors:

1. *Inherently global problems*: A lot of prior work focuses on problems that are inherently global—for example, the task is to find a solution such that the difference between the minimum load and the maximum load is at most 1. It is easy to see that any algorithm for solving such problems takes $\Omega(n)$ rounds in the worst case.

2. *Natural but inefficient algorithms*: Many papers study various natural processes for doing load balancing. Many of these are of match-and-balance type, and virtually all of these are careful. Typically, the negative results of Theorems 2 and 3 apply.

In contrast, we study a problem that can be solved efficiently, and our algorithms demonstrate that it is indeed possible to break the barriers of Theorems 2 and 3. In what follows, we will discuss related work in more detail.

**Reducing a global potential with local rules.** There is a lot of literature on load balancing when the goal is to reduce a global potential function by iterating a local balancing rule. Examples of such potential functions are the difference between the maximum and the minimum load (*discrepancy*), the maximum load (*makespan*), and the quadratic difference to the average load.

Various models are considered: two classic models are the *diffusion model*, where vertices distribute their load to all their neighbours, and the *matching model*, where the load is exchanged only along the edges of a matching—for example a random matching or an edge colouring.

In the *continuous case*, where the loads are assumed to be infinitely divisible, the speed of convergence was analysed for simple schemes both in the diffusion model [21, 23] and the matching model [6, 11]. In both the speed of convergence is essentially captured by the spectral properties of the graph in question.

In the context of indivisible loads, known as the *discrete case*, similar problems were first studied for networks designed to balance the load quickly [20]. Different schemes for reducing the discrepancy in the discrete case were analysed, the question of whether the speed of convergence in the continuous case could be matched, remained open [1, 11, 12, 19]. Recently Sauerwald and Sun [22] were able to prove convergence as fast as in the continous case, up to constant factors. Reducing discrepancy is a global problem and can take linear time in the worst case.

**Semi-matching problem.** In the *semi-matching problem* the nodes of a graph are divided into clients and servers [14]. Each client has to be assigned to an adjacent server. The goal is to optimise the total waiting time of the clients.

Czygrinow et al. [8] presented a distributed algorithm for finding a locally optimal semi-matching in time $\mathrm{poly}(\Delta)$; this also implies a factor-2 approximation of globally optimal semi-matchings.

The semi-matching problem is very similar to the locally optimal load balancing problem, especially when limited to the case of degree 2 clients, with the tokens being more "localised". Indeed, our linear lower bound can be adapted to prove an $\Omega(\Delta)$ lower bound for locally optimal semi-matchings.

**Balls into bins.** In the *d-choice process* each of $n$ balls goes in the least loaded of $d$ random bins. Dependency of the maximum load on the parameter $d$ is well known [3, 16, 24]. The choice of the bins can be modelled by a graph [17]; in one variant the bins are connected by edges and each ball does a local search until it finds a local minimum [5, 7]. This process produces a locally optimal load balancing.

**Sandpile models and chip-firing games.** Our stability condition is similar to what is used in *sandpile models* [4, 9, 15] and *chip-firing games* [2]. However, in these problems the goal is usually to describe final configurations for fixed, very simple algorithms that simulate a natural phenomenon.

**Filtering.** *Sliding window algorithms* for computing the *running average* or for *image filtering* are natural local algorithms. Averaging type algorithms, however, cannot guarantee an integral solution to load balancing problems. *Median filtering* does guarantee integral solutions for integral inputs; however, it does not preserve the total load.

**Games and equilibriums.** The locally optimal load balancing problem can be seen as a problem of finding an equilibrium state, where no single load token can gain advantage by moving. We show that such an equilibrium can be found locally, that is, the decisions made in one part of the graph do not propagate too far. This is in contrast with problems such as finding *stable matchings*, where there is a local algorithm only for finding almost-stable matchings [10].

**Matchings.** Locally optimal load balancing is closely relate to *bipartite maximal matching*: if the initial loads are $x(v) \in \{0, 2\}$, then it is easy to see that a solution can be found using a bipartite maximal matching algorithm. This is a problem that can be solved in time $O(\Delta)$ [13]. Showing a matching lower bounds is a major open question, and we do not expect that one can prove tight lower bounds for locally optimal load balancing as a function of $\Delta$ before we resolve the distributed time complexity of bipartite maximal matching.

In our algorithms for discrete load balancing, we will use the bipartite maximal matching [13] algorithm as a subroutine. For fractional load balancing, we use the *almost-maximal fractional matching algorithm* due to Khuller et al. [18] as a subroutine.

# 3 Negative results

We will now prove the negative results of Theorems 1–4. For simplicity, we prove the statements for deterministic distributed algorithms; it is fairly straightforward to extend the results to randomised algorithms (e.g., consider the expected values of the outputs).

Recall that in Section 1.1 we defined the problem so that the output is bounded by $L$. However, we will not exploit this restriction in any of the lower-bound proofs. The negative results hold verbatim for a relaxed version of the problem in which the outputs can be any nonnegative real numbers. We only assume that the inputs are bounded by $L$.

## 3.1 Load balancing on paths and cycles

We start with the unconditional lower bound that holds for any algorithm, for both fractional and discrete load balancing, and in the simplest possible case of paths or cycles.

**Theorem 1.** *Load balancing requires $\Omega(L)$ rounds, even in the case of paths and cycles.*

*Proof.* We will give the proof for the case of paths; the case of cycles is very similar. Consider a path $P$ with $n$ nodes, labelled with the numbers $1, 2, \ldots, n$ from left to right, for a sufficiently large $n$. Let $A$ be a load-balancing algorithm. For an input $x\colon v \to L$, we write $A(x)$ for the output of $A$ on input $x$. Let $h = \lfloor L/2 \rfloor - 1$.

Consider the following constant inputs: $x_0\colon v \to 0$ and $x_L\colon v \to L$. Let $y_0 = A(x_0)$ and $y_L = A(x_L)$. Clearly $y_0(v) = 0$ for all $v$ and $y_L(v) \geq L$ for at least one $v$. Hence we can find two nodes, $\ell$ and $r$, such that

$$y_0(\ell) = 0, \quad y_L(r) \geq L, \quad |r - \ell| = L - 1.$$

See Figure 1 for an illustration.

W.l.o.g., assume that $\ell < r$. Let $m = (r + \ell)/2$ be the midpoint between $\ell$ and $r$. Now define an input $x$ such that $x(i) = 0$ for $i \leq m$ and $x(i) = L$ otherwise. Note that the radius-$h$ neighbourhoods of $\ell$ are identical in $x_0$ and $x$. Similarly, the radius-$h$ neighbourhoods of $r$ are identical in $x_L$ and $x$.

Let $y = A(x)$. If $y(\ell) = y_0(\ell)$ and $y(r) = y_L(r)$, we have a contradiction: the distance between $\ell$ and $r$ is smaller than their load difference, and hence there has to be an unhappy edge between them. Therefore $y(\ell) \neq y_0(\ell)$ or $y(r) \neq y_L(r)$. In both cases, there is a node $v$ that changed its output between two instances, even though the inputs were identical up to distance $h$. Hence the running time of $A$ has to be at least $h + 1 = \Theta(L)$. □

## 3.2 Match-and-balance algorithms

Recall that in each round, a match-and-balance algorithm finds some matching $M$, and then for each edge $(u, v) \in M$ with $y(u) > y(v)$, the algorithm increases the flow $f(u, v)$ by at most $(y(u) - y(v))/2$. Note that $M$ does not need to be a maximal matching, a maximum matching,

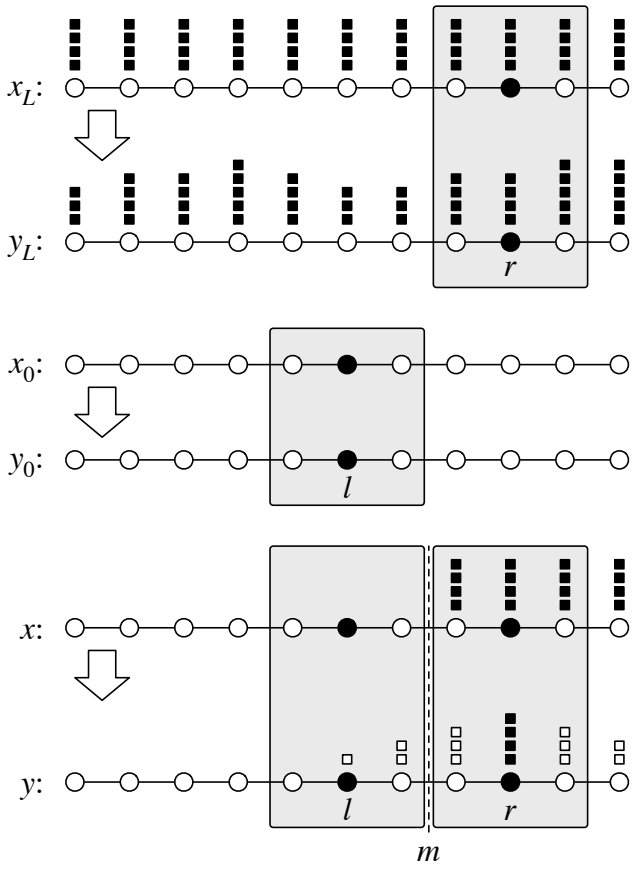



Figure 1: The proof of Theorem 1 in Section 3.1. In this example, $L = 4$ and $h = 1$. We can find a node $r$ with output at least $L$ in $y_L$, and a node $\ell$ with output 0 in $y_0$ so that the distance between $\ell$ and $r$ is $L-1$. Then we construct instance $x$ that looks like $x_L$ in the $h$-neighbourhood of $r$ and it looks like $x_0$ in the $h$-neighbourhood of $\ell$. If node $r$ does not change its output between $y_L$ and $y$, then node $\ell$ has to change its output between $y_0$ and $y$. Hence the running time is at least $h + 1$.

or a random matching—the following lower bound holds regardless of how clever the algorithm tries to be in its selection of the matching $M$, and even if it gets the matchings in zero time from an oracle.

**Theorem 2.** *Any match-and-balance algorithm takes $\Omega(L^2)$ rounds in the worst case, even in paths and cycles.*

The basic idea of the proof is simple. Let $A$ be a match-and-balance algorithm.

1. We construct an instance in which $A$ has to move $\Omega(L^3)$ units of load in total.
2. We prove that $A$ can move only $O(L)$ units of load per round.

Hence we have a lower bound of $\Omega(L^2)$ for the running time of $A$.

We will again study the case of paths; the case of cycles is very similar. Let $P$ be a path with $2n + 1$ nodes, labelled with $-n, -n + 1, \ldots, n$ from left to right. We say that a load vector is *monotone* if $y(i) \geq y(j)$ for all $i \leq j$; see Figure 2. The key feature of match-and-balance algorithms is that a monotone load vector remains monotone after each step.

**Lemma 9.** *Match-and-balance algorithms maintain a monotone load configuration on $P$.*

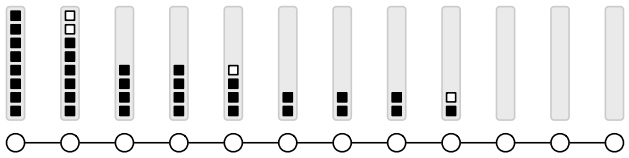


Figure 2: The proof of Lemma 9 in Section 3.2. In this example, $L = 8$ and the load distribution is monotone. A match-and-balance algorithm can only move at most $L/2 = 4$ units of load (the highlighted tokens).

*Proof.* Assume that the current load configuration $y$ is monotone. Let $M$ be a matching and let $y'$ be the load configuration after balancing over $M$. Consider nodes $i$ and $i+1$. Initially $y(i) \geq y(i+1)$; we will prove by a case analysis that $y'(i) \geq y'(i+1)$:

1. $\{i, i+1\} \in M$: we will have $y'(i) \geq y'(i+1)$.
2. $\{i, i+1\} \notin M$:
   - $\{i, i-1\} \notin M$: we will have $y'(i) = y(i)$.
   - $\{i, i-1\} \in M$: we will have $y'(i) \geq y(i)$.
   - $\{i+1, i+2\} \notin M$: we will have $y'(i+1) = y(i+1)$.
   - $\{i+1, i+2\} \in M$: we will have $y'(i+1) \leq y(i+1)$.

In each case $y'(i) \geq y'(i+1)$. □

In a monotone configuration, we can only move $O(L)$ units of load per round—see Figure 2.

**Lemma 10.** *Any match-and-balance algorithm $A$ can move at most $L/2$ units of load in a single round on path $P$ with a monotone load configuration.*

*Proof.* Since $A$ maintains a monotone load configuration, the sum of the load differences over all edges is at most $L$. Therefore even if $M$ contains all edges with a non-zero load difference, the algorithm can move only at most $L/2$ units of load per round in total. □

*Proof of Theorem 2.* We will consider the input vector $x$ where $x(i) = L$ for $i \leq 0$ and $x(i) = 0$ otherwise. The vector is monotone and hence it remains monotone throughout the execution of $A$. Consider the output of node 0. There are two cases; see Figure 3:

(a) The output of node 0 is at most $h = L/2$. Now for each $i = 0, 1, \ldots, h-1$, we can observe that the load of node $-i$ has decreased by at least $h-i$ units, and by monotonicity, all of this load has been moved to the right. In particular, for each $i$ we have moved $h-i$ units of load from node $-i$ over at least $i+1$ edges. The total amount of work done by the nonpositive nodes is at least the tetrahedral number $1 \cdot h + 2 \cdot (h-1) + \ldots + h \cdot 1 = \Theta(h^3) = \Theta(L^3)$.

(b) The output of node 0 is at least $h = L/2$. Now for each $i = 0, 1, \ldots, h-1$, we can observe that the load of node $i$ has increased by at least $h-i$ units, and by monotonicity, all of this load has been moved from the left. The total amount of work done by the nonnegative nodes is at least $\Theta(L^3)$.

By Lemma 10, moving $\Theta(L^3)$ units of load takes $\Omega(L^2)$ rounds. □

### 3.3 Careful algorithms

Recall that careful algorithms move $O(L)$ units of load per round—this includes, for example, all match-and-balance algorithms, as well as many other natural algorithms that simulate the physical process of collapsing piles of tokens.

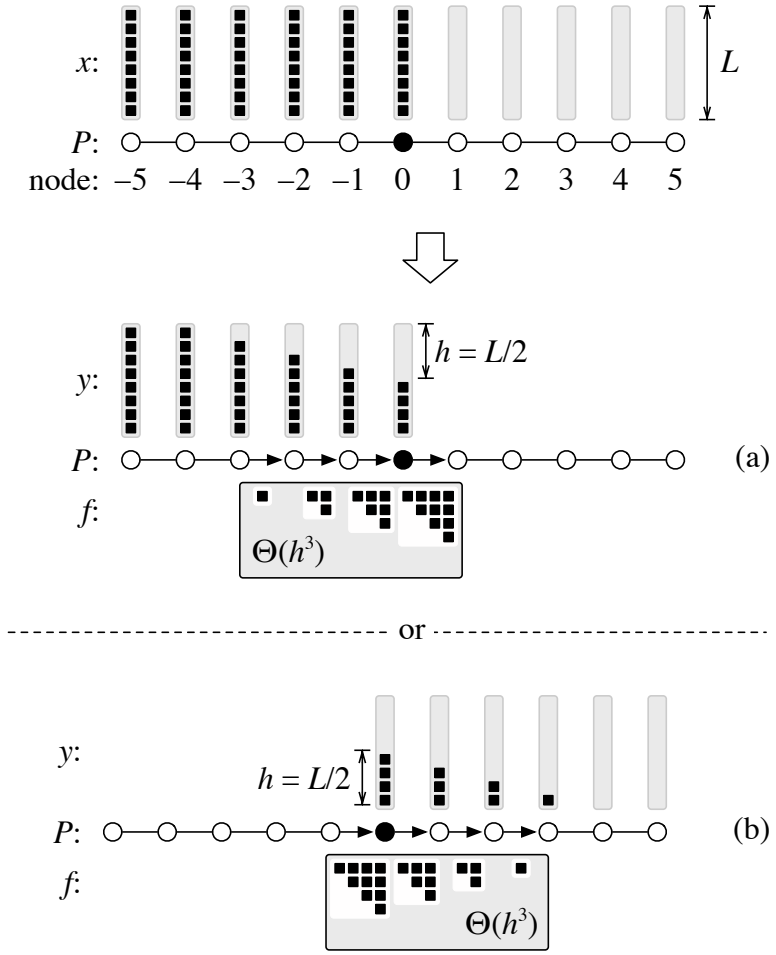



Figure 3: The proof of Theorem 2 in Section 3.2. We construct input $x$, run any match-and-balance algorithm, and have a case analysis based on the output of node 0: (a) The load of node 0 decreases by at least $h$, and the nonpositive nodes do $\Omega(h^3)$ units of work in total to push load to the right. (b) The load of node 0 is still at least $h$, and the nonnegative nodes do $\Omega(h^3)$ units of work in total to pull load from the left.

**Theorem 3.** *Any careful algorithm takes $\Delta^{\Omega(L)}$ rounds in the worst case.*

*Proof.* Construct the input $(G, x)$ as shown in Figure 4: We have a tree $G_u$ rooted at $u$, a tree $G_v$ rooted at $v$, plus an edge $\{u, v\}$. Both trees are of depth $L/4$; each non-leaf node has $d-1$ children. All nodes of $G_u$ have an input load of 0, and all nodes of $G_v$ have an input load of $L$.

Now consider any solution $(y, f)$. If $y(u) \geq L/4$, then all nodes of $G_u$ have a load of at least 1, and there are $d^{\Omega(L)}$ nodes in $G_u$. All of the load has been moved across the edge $\{u, v\}$, and hence $f(v, u) = d^{\Omega(L)}$. Otherwise $y(u) < L/4$, and $y(v) < L/4 + 1$. In this case all nodes of $G_v$ have a load of at most $L - 1$, and again we can conclude that $f(v, u) = d^{\Omega(L)}$.

A careful algorithm starts with $y \leftarrow x$ and $f \leftarrow 0$ and changes each element of $f$ by at most $\text{poly}(L)$ in each round. Hence any careful algorithm has to spend $d^{\Omega(L)}$ for this instance. □

### 3.4 Oblivious algorithms

Recall that in an oblivious algorithm, the total amount of load moved from node $u$ to $v$ only depends on the initial load of $u$ and the distance between $u$ and $v$. For example, the algorithm that computes the moving average in an infinite path is an oblivious algorithm. We show that such algorithms do not exist for infinite regular trees of a degree larger than 2.

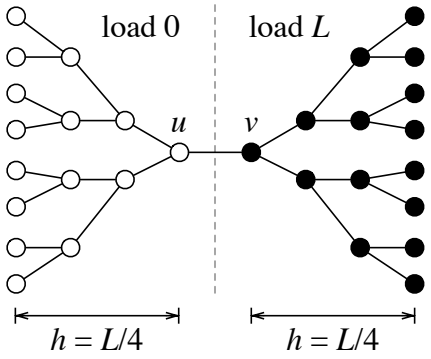



Figure 4: The proof of Theorem 3 in Section 3.3. In this example, $d = 3$ and $L = 12$.

**Theorem 4.** *There are no oblivious algorithms for infinite $d$-regular trees with $d \geq 3$.*

*Proof.* We say that a node is *full* if it has a load of $L$, and *empty* if it has a load of 0. We say that a subtree is full if all nodes in it are full, and a subtree is empty if all nodes in it are empty.

Construct the input $(G, x)$ as shown in Figure 5a:

- $G$ is the $d$-regular infinite tree,
- $\{u, v\}$ is an edge of $G$,
- each node $w$ that is closer to $u$ than $v$ is empty,
- each node $w$ that is closer to $v$ than $u$ is full.

We will consider the infinite tree $G$ rooted at either $u$ or $v$. If we root it at $u$, then $u$ is adjacent to 1 full subtree and $d-1$ empty subtrees. If we root it at $v$, then $v$ is adjacent to $d-1$ full subtrees and 1 empty subtree. See Figure 5a for illustrations.

Let $g(r)$ be the amount of load that the oblivious algorithm moves from a full node $w$ to any node that is at distance $r$ from $w$. Define the shorthand notation

$$\alpha = \sum_{r=0}^{\infty} (d-1)^r g(r+1),$$

which has two equivalent interpretations in rooted infinite $d$-regular trees:

- a full root node sends in total $\alpha$ units of load to each subtree,
- the root node receives $\alpha$ units from each full subtree.

See Figure 5b. In total, a full node gives $d\alpha$ units of load to other nodes, so it leaves

$$\beta = L - d\alpha$$

units of load for itself. It is easy to verify that we must have $\beta \geq 0$ and hence $\alpha \leq L/d$; otherwise there would be inputs with negative outputs.

Now we are ready to put the pieces together. Node $u$ receives $\alpha$ units of load from its only full subtree, while $v$ receives $(d-1)\alpha$ units of load from its $d-1$ full subtrees; moreover, $v$ leaves $\beta$ units of load for itself. The load difference between $u$ and $v$ is therefore at least

$$(d-1)\alpha + \beta - \alpha = L - 2\alpha \geq \frac{d-2}{d} L.$$

Hence for any $d \geq 3$ and for a sufficiently large $L$, edge $\{u, v\}$ will be unhappy in $G$, no matter which oblivious algorithm we apply. □

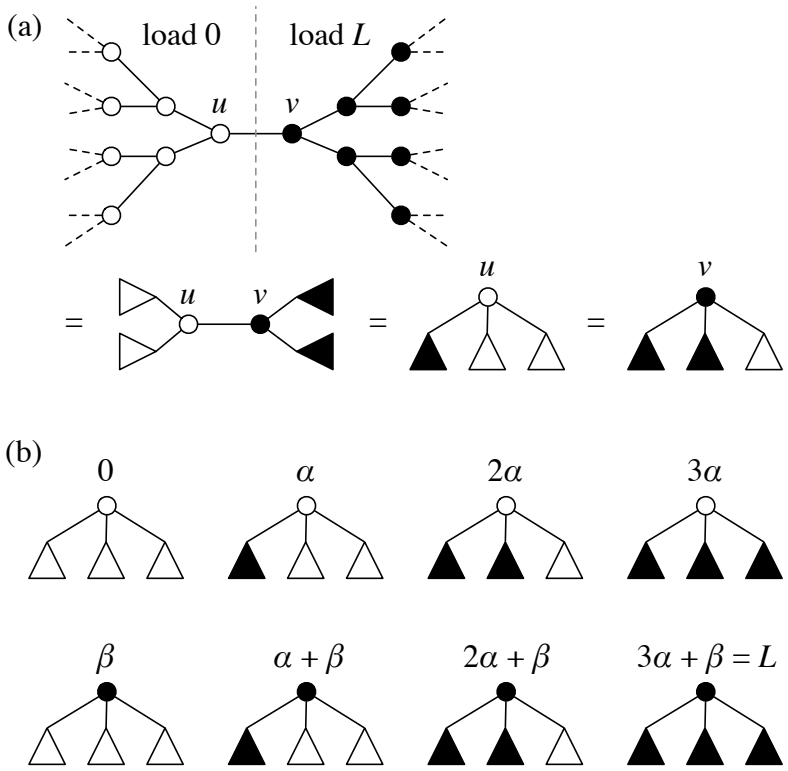



Figure 5: The proof of Theorem 4 in Section 3.4. In this example, $d = 3$. (a) The input graph $G$ is an infinite $d$-regular tree rooted at $\{u, v\}$; one half has an input load of 0, and the other half has an input load of $L$. (b) In the output, each full subtree contributes $\alpha$ units of load, and the node itself contributes $\beta$ units of load.

# 4 Positive results

We will now prove the positive results of Theorems 5, 7, and 8.

## 4.1 Discrete load balancing in paths and cycles

We first give an algorithm that exactly matches the lower bound of Theorem 1.

**Theorem 5.** *Discrete load balancing can be solved in time $O(L)$ in paths and cycles, with a deterministic local algorithm.*

**Infinite directed paths.** We will first show how to do load balancing in an *infinite path with a consistent orientation*. That is, each node $v$ has a degree of 2, and it can refer to its *left neighbour* $v - 1$ and *right neighbour* $v + 1$ in a globally consistent manner.

We will interpret the path with tokens as a 2-dimensional grid, indexed by $(v, i)$, where $v \in V$ is a node and $i \in \{1, \ldots, L\}$ is a possible location for a token. We say that $(v, i)$ is a *slot*. Initially, slot $(v, i)$ holds a *token* if $x(v) \geq i$. Our plan is to move the tokens around in the grid so that we maintain the following stability conditions—see Figure 6 for an illustration.

**Definition 1.** A token in slot $(v, i)$ is $k$-stable if $i = 1$ or there is a token in slot $(v + k, i - 1)$. A configuration is $k$-stable if all tokens are $k$-stable. For a set $K$, a configuration is $K$-stable if it is $k$-stable for all $k \in K$.

We write $[\![a, b]\!] = \{a, a + 1, \ldots, b\}$. Initially, the configuration is 0-stable. If we can find a $[\![-1, 1]\!]$-stable configuration, we can construct a feasible solution to the load balancing problem by simply setting $y(v)$ to be equal to the number of tokens in slots $(v, \cdot)$.

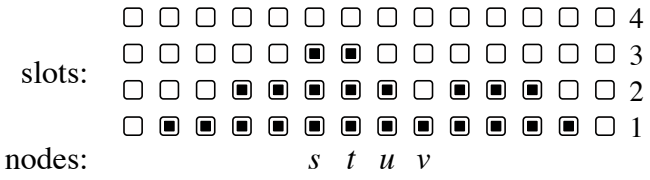



Figure 6: Stability. Token $(s, 3)$ is 2-stable, as there exists a token in slot $(u, 2)$, where $u = s + 2$, i.e., the node 2 steps right from $s$. Also $(s, 2)$ and $(s, 1)$ are 2-stable. However, this configuration is not 2-stable: token $(t, 3)$ is not 2-stable, as there is an empty slot $(v, 2)$. It can be verified that the configuration is 0-stable, 1-stable, and $(-1)$-stable.

However, we will now design an $O(L)$-time algorithm with a *stronger* stability condition: it will compute a $[\![-3, 3]\!]$-stable configuration. Informally, we smooth out the load distribution so that the slope of the load curve is at most $1/3$. This extra slack will be helpful when we eventually want to solve the problem in paths without consistent orientations.

This algorithm is based on the concept of *pushes*. For a node $v$ and integer $\ell$, define the $\ell$-diagonal of $v$ as the following list of slots (see Figure 7):

$$S(v, \ell) = \big((v - \ell, 1), (v - 2\ell, 2), \ldots, (v - L\ell, L)\big)$$

In an $\ell$-push we redistribute the tokens in each $S(v, \ell)$: if there are $k$ tokens in $S(v, \ell)$, then we redistribute the tokens so that the first $k$ elements of $S(v, \ell)$ are occupied and the remaining $L - k$ elements are empty (see Figure 7). In essence, we let the tokens slide along each diagonal so that they are piled on the bottom of each diagonal.

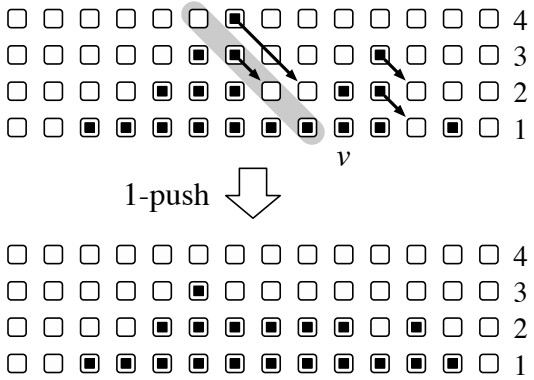



Figure 7: Pushing. The 1-diagonal of $v$ is highlighted. In a 1-push, we redistribute the tokens in each 1-diagonal so the end result will be a 1-stable configuration. Note that this configuration was already 0-stable and $(-1)$-stable, and it remained 0-stable and $(-1)$-stable after a 1-push. In general, whatever stability we have already achieved by pushing is never lost in subsequent pushes.

An $\ell$-push can be efficiently implemented in time $O(\ell L)$ with a distributed algorithm: for example, node $v$ is responsible for redistributing the tokens in slots $S(v, \ell)$, and we first use $O(\ell L)$ rounds so that each node $v$ can discover everything related to $S(v, \ell)$, and then another $O(\ell L)$ rounds so that node $v$ can inform the relevant nodes regarding how to move tokens in $S(v, \ell)$.

Clearly, after an $\ell$-push we will have an $\ell$-stable configuration. The non-trivial part is that $\ell$-pushes do not interfere with any stability that we have previously achieved.

**Lemma 11.** *For every choice of integers $\ell$ and $k$, if a configuration is $k$-stable, then it is still $k$-stable after an $\ell$-push.*

*Proof.* The case $k = \ell$ is trivial; hence we assume that $k \neq \ell$. Consider slots $a = (v, i)$ and $b = (v + k, i - 1)$; see Figure 8. We need to argue that if $a$ holds a token after an $\ell$-push, then $b$ will also hold a token after an $\ell$-push. To this end, let $A$ be the $\ell$-diagonal that contains $a$, and let $B$ be the $\ell$-diagonal that contains $b$. Now by definition, after an $\ell$-push, slot $a$ is occupied if and only if there were at least $i$ tokens in $A$.

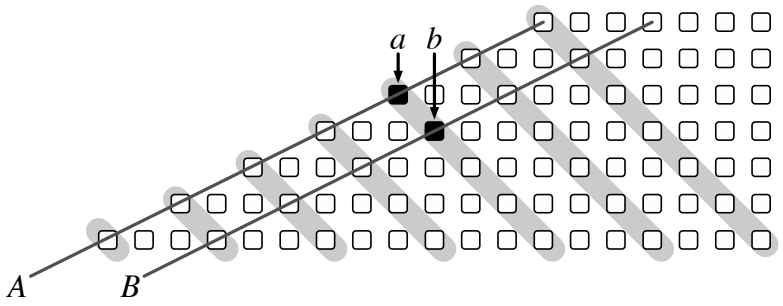



Figure 8: The proof of Lemma 11, for $\ell = -2$ and $k = 1$. The configuration was 1-stable, i.e., the gray 1-diagonals filled starting from the bottom. Now we do a $(-2)$-push, and want to argue that the configuration will be still 1-stable. Consider slot $a$. It will be filled iff there are at least 5 tokens in the $(-2)$-diagonal $A$. But this implies that there are at least 4 tokens in the diagonal $B$, and hence the slot $b$ will be filled, too.

The key observation is that $k$-stability implies that for every token in $A$ there is a token in $B$, with the exception of the first token—if $(u, j) \in A$ holds a token and $j > 1$ then $(u+k, j-1) \in B$ holds a token as well. In particular, if there were at least $i$ tokens in $A$, there were at least $i - 1$ tokens in $B$, and hence $b$ will also hold a token. □

Now we can easily find a $[\![-3, 3]\!]$-stable configuration in time $O(L)$: the algorithm simply does an $\ell$-push for each $\ell \in [\![-3, 3]\!]$, sequentially, in an arbitrary order. We will call this algorithm $A_1$.

**Finite directed paths and cycles.** Algorithm $A_1$ finds a $[\![-3, 3]\!]$-stable configuration in infinite directed paths in time $O(L)$. To handle *finite* directed paths we could extend the algorithm and its analysis so that it takes into account the boundary effects. However, this would be a bit boring—instead, we will show that we can simply take $A_1$ and use it as a black box.

Let us first adjust the stability condition so that it makes sense on finite paths: a token $(v, i)$ is considered $k$-stable also if node $v + k$ does not exist.

Let $T_1 = \Theta(L)$ be the worst-case running time of $A_1$. We use it to construct an algorithm $A_2$ that finds a $[\![-3, 3]\!]$-stable configuration for a finite path $G$, as follows:

1. Check if the path is of length at most $4T_1$; if so, we solve the problem by brute force in time $O(L)$, and stop.
2. Each endpoint $u$ gathers all tokens up to distance $2T_1$ and redistributes them so that all nodes within distance at most $T_1$ from $u$ have the same constant load; let us denote this constant $c(u)$.
3. Construct a virtual graph $G'$ as follows: each endpoint $u$ pretends that the path continues with infinitely many additional dummy nodes, each with the same constant load $c(u)$.
4. Simulate algorithm $A_1$ in the virtual graph $G'$.
5. Discard the dummy nodes.

It is easy to verify that the $A_1$ will never move any tokens across an endpoint, as its neighbourhood was already well-balanced. Therefore if we remove the dummy nodes, we have a feasible solution for $G$. Moreover, the running time of $A_2$ is still $O(L)$.

It is also easy to see that $A_2$ works correctly in directed *cycles*; the first three steps simply do nothing as there are no endpoints.

**Undirected paths and cycles.** So far we have designed an algorithm $A_2$ that finds a $[\![-3, 3]\!]$-stable configuration in paths and cycles with a globally consistent orientation. Now we show how to use it to design an algorithm $A_3$ that finds a $[\![-1, 1]\!]$-stable configuration in paths and cycles without an orientation.

It can be shown that *some* form of local symmetry-breaking is needed. We will use the familiar *port-numbering model*: Each node $v$ has up to two communication ports, labelled with $(v, 1)$ and $(v, 2)$. The ports are identified with the endpoints of the edges; each edge joins a pair of ports. The port numbers at the endpoints of an edge do not need to match—for example, an edge $\{u, v\}$ may join $(u, 1)$ to $(v, 1)$ or $(u, 1)$ to $(v, 2)$.

In algorithm $A_2$, we construct a *virtual graph* $G'$ as shown in Figure 9: Each node $v$ splits itself in two virtual nodes, $v_1$ and $v_2$. The virtual nodes also have two ports. For each edge $e = \{u, v\}$, depending on the type of $e$ we connect the virtual nodes of $u$ and $v$ as follows:

- $e$ joins $(u, 1)$ to $(v, 1)$: connect $(u_1, 1)$ to $(v_2, 2)$ and $(u_2, 2)$ to $(v_1, 1)$,
- $e$ joins $(u, 1)$ to $(v, 2)$: connect $(u_1, 1)$ to $(v_1, 2)$ and $(u_2, 2)$ to $(v_2, 1)$,
- $e$ joins $(u, 2)$ to $(v, 1)$: connect $(u_1, 2)$ to $(v_2, 1)$ and $(u_2, 1)$ to $(v_1, 2)$,
- $e$ joins $(u, 2)$ to $(v, 2)$: connect $(u_1, 2)$ to $(v_1, 1)$ and $(u_2, 1)$ to $(v_2, 2)$.

If $G$ was a path with $n$ nodes, then $G'$ consists of two disjoint paths with $n$ nodes each. If $G$ was an $n$-cycle, then $G'$ consists of either one cycle with $2n$ nodes or two cycles with $n$ nodes each.

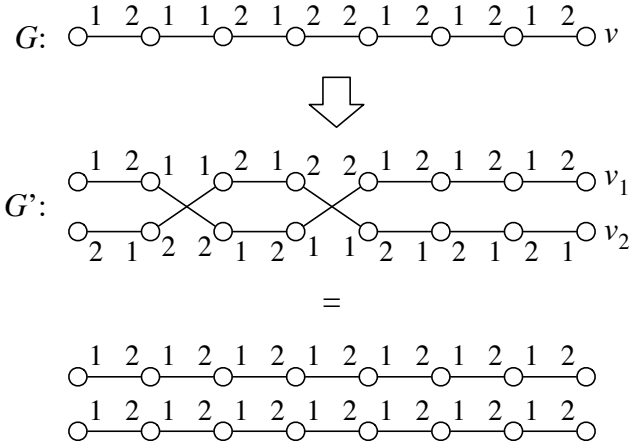



Figure 9: Given any path $G$ with some port numbering, we can construct a virtual graph $G'$ that consists of two paths, both of which have a *consistent* port numbering.

The key observation is that there is a *consistent* port numbering in $G$: port 1 of a virtual node is always connected to port 2 of an adjacent virtual node. We can now interpret the ports so that in each virtual node port 1 points "left" and port 2 points "right".

Each node first splits its input load arbitrarily between its virtual copies. Then we run algorithm $A_2$ to find a $[\![-3, 3]\!]$-stable configuration in the virtual graph, and then map all tokens back to the original graph: the new load of $v$ is the sum of the new loads of $v_1$ and $v_2$; see Figure 10.

Now we have a configuration where the maximum load difference between a pair of adjacent nodes is 2. However, the load is *approximately well-balanced*: a load difference of more than 2

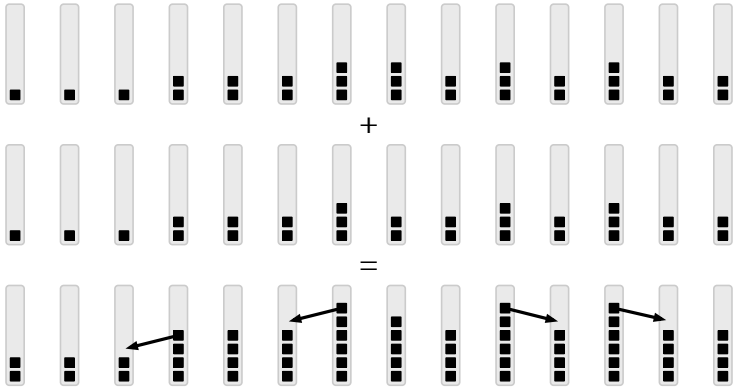


Figure 10: The sum of two $[\![-3,3]\!]$-stable configurations can be easily turned into a $[\![-1,1]\!]$-stable configuration with local modifications.

implies a distance of at least 4. Therefore we can easily find a $[\![-1,1]\!]$-stable configuration in $O(1)$ time with local operations (see Figure 10). For example, we can apply a match-and-balance algorithm: find a maximal matching $M$ of unhappy edges and move a token over each edge. Conveniently, all edges become happy, including those that were not in $M$. It is easy to find a maximal matching $M$ in $O(1)$ time, as this is in essence maximal matching in a bipartite graph of maximum degree 2: on one side we have the nodes that are "too low" and on the other side we have the nodes that are "too high" in comparison with their neighbours.

In summary, we can find a $[\![-1,1]\!]$-stable configuration in any path or cycle in time $O(L)$, and therefore we can do discrete load balancing in any path or cycle in time $O(L)$.

### 4.2 Discrete load balancing in general graphs

We will now show how to do discrete load balancing in graphs of maximum degree $\Delta$.

**Theorem 8.** *Discrete load balancing can be solved in time $T(L,\Delta)$, for some function $T$, in bounded-degree graphs with a deterministic local algorithm.*

Again, we will imagine that each node $v$ has $L$ *slots*, labelled $(v,\cdot)$, and each token is placed in one of the slots. Initially slots $(v,1),(v,2),\ldots,(v,x(v))$ are occupied with tokens.

We define the *(downward) cone* $C(v,i)$ of slot $(v,i)$ as the set of slots $(u,j) \neq (v,i)$ such that $i-j \geq \operatorname{dist}(v,u)$; see Figure 11. In the algorithm, if there is a token in $(v,i)$ and all slots of the cone $C(v,i)$ are full, then we say that the token is *stable*, and we *freeze* it, i.e. it will never be moved again.

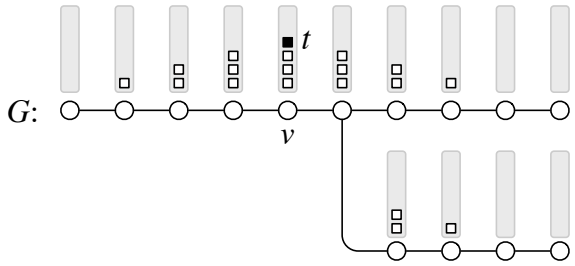



Figure 11: The downward cone $C(v,4)$ of the token $t=(v,4)$ consists of the slots denoted with white boxes.

In the algorithm we try to match the highest unfrozen tokens with the free slots in their cones. If they succeed then they move to these slots; otherwise they can be frozen.

We now give the pseudo-code of the algorithm in a centralised way, prove the correctness of the algorithm, and then show that it is actually a local algorithm. The algorithm proceeds as follows:

1. All stable tokens of the initial configuration are frozen.
2. For each $h = L, L-1, \ldots, 1$:
   (a) Construct the virtual bipartite graph $F_h = (T \cup S, E)$, where $T$ consists of unfrozen tokens at level $h$, $S$ consists of all empty slots at levels below $h$, and there is an edge $\{t, s\}$ if $s \in S$ is an empty slot in the cone of token $t \in T$.
   (b) In $F_h$, find a maximal matching $M$.
   (c) For every unfrozen token $t$ at level $h$: if the token is matched with a slot $s$ in $M$, move the token to slot $s$, otherwise freeze it.
   (d) Collapse the tokens so that for each node $v$ that holds $k$ tokens, the tokens are in the slots $(v, 1), (v, 2), \ldots, (v, k)$.

First, remark that we maintain the invariant that at round $h$, all load in slots at height $h$ either moves down or is safely frozen. Indeed, if a token is not matched, then all slots in its cone will be full at the end of the loop, and if it is matched, it moves to a strictly lower level, thereafter the invariant is true for level $h$ and maintained for the levels above. At the end of the algorithm all the tokens are frozen, thus the configuration is stable.

We stated the algorithm in a centralised manner, but it is actually local: The vertices only need the knowledge of their radius-$L$ neighbourhood to find their neighbours in graph $F_h$. Graph $F_h$ has a maximum degree of $O(L\Delta^L)$. Therefore we can find a maximal matching in $F_h$ by simulating $O(L\Delta^L)$ rounds of the proposal algorithm [13] in the virtual graph $F_h$. The simulation has a multiplicative $O(L)$ overhead—adjacent nodes in $F_h$ are at distance $O(L)$ in graph $G$. Finally, we have $O(L)$ iterations, giving the overall complexity of $O(L^3\Delta^L)$.

### 4.3 Fractional load balancing in general graphs

In fractional load balancing, we can use the same basic idea as what we had in the discrete case, but much faster:

**Theorem 7.** *Fractional load balancing can be solved in time* $\mathrm{poly}(L, \Delta)$ *in bounded-degree graphs with a deterministic local algorithm.*

Our algorithm follows the same basic structure as the discrete algorithm of Section 4.2. However, in each bipartite virtual graph $F_h$, we compute an $\varepsilon$-maximal fractional matching. With the algorithm by Khuller et al. [18], this can be done in $O(\log \frac{1}{\varepsilon} + \log \Delta)$ rounds, which gives us an exponential speedup over the $O(\Delta)$-round algorithm for maximal bipartite matching.

**Almost maximal fractional matchings.** Let the bipartite graph be $G = (V, E)$ with $V = T \cup S$ and $\Delta$ be the maximum degree. Each node has a *capacity* $c\colon V \to [0, 1]$. A fractional matching is a function $y\colon E \to [0, 1]$ such that for each node, the sum $y[v] = \sum_{e \ni v} y(e)$ is at most $c(v)$. A fractional matching is *$\varepsilon$-maximal* if

$$\max_{e \in E} \min\{c(v) - y[v] : v \in e\} \leq \varepsilon,$$

that is, there is no edge $e$ with a value $y(e)$ that could be increased by more than $\varepsilon$.

**Algorithm for fractional load balancing.** Next we describe the algorithm for finding a fractional load balancing. As before, we have *slots* labelled with $(v, i)$; here $v$ is a node and $i$ is the *level* of the slot. However, now each slot may contain fractional units of load. We adapt the definition of stability to fractional load balancing in a natural manner: we say that $\alpha$ units of the load in slot $(v, i)$ is stable if each $(u, j)$ with $i - j = d(v, u)$ has at least $\alpha$ units of load, and each $(u, j)$ with $i - j > d(v, u)$ is full. In the algorithm we will *freeze* some parts of the load. We use $\ell_i(v)$ to denote the total amount of load in slot $(v, i)$, and $f_i(v)$ to denote the amount of frozen load in slot $(v, i)$.

The algorithm would be simpler to analyse if we had a maximal fractional matching algorithm, but we only have an efficient $\varepsilon$-maximal one. Our strategy is to round the load, and by doing it, we accumulate *surplus* and *deficit*. This way we can analyse easily the iterations of the algorithm. We keep track of surplus and deficit, and at the end of the algorithm we readjust the loads.

The algorithm for fractional load balancing works as follows. First, double all load; this way we have $2L$ slots per node. Based on the new input, freeze all stable load. Then, for each $h = 2L, 2L - 1, \ldots, 1$ perform the following steps:

1. Construct the virtual bipartite graph $F_h = (T \cup S, E)$, where $T$ consists of slots with unfrozen load at level $h$, $S$ consists of all slots at levels below $h$, and there is an edge $\{t, s\}$ if $s \in S$ is a non-full slot in the cone of slot $t \in T$.
2. Define the capacities of $F_h$ as follows: the capacity of a node $(t, h) \in T$ is its unfrozen load $c(t, h) = \ell_h(s) - f_h(t)$, and the capacity of a node $(s, i) \in S$ is its free space $c(s, i) = 1 - \ell_i(s)$.
3. Find an $\varepsilon$-maximal fractional matching $y$ in $F_h$.
4. For each edge $e = \{t, s\}$ of $F_h$, move $y(e)$ units of load from slot $t$ to slot $s$.
5. For each $t$ consider the two cases:
   - If $t$ has load at most $\varepsilon$, round it to 0 and freeze the load in $t$. We create at most $\varepsilon$ units of deficit in slot $t$ at the moment we freeze it.
   - If $t$ has strictly more than $\varepsilon$ load, then each unfrozen slot in the cone of $t$ has at least $1 - \varepsilon$ load. We round them to 1 and freeze all load at $t$ and in $C(t)$. We create at most $\varepsilon$ units of surplus in each slot of $C(t)$ at the moment we freeze it.
6. Collapse all unfrozen load as low as possible.

Finally, undo the rounding—remove the surplus and put back the deficit. Then normalise the output by dividing all load by two.

**Analysis.** Thanks to the rounding, the configuration after each iteration satisfies the same invariant as the discrete algorithm: at round $h$, all load in slots at height $h$ either moves down or is safely frozen. Then the configuration at the end of the iteration phase is stable and for each edge $\{u, v\}$ we have $|y(u) - y(v)| \leq 1$.

Each slot is involved at most once in the rounding, precisely at the moment we freeze it. Hence before normalisation, the difference between the real load and the rounded load is in $[-2L\varepsilon, 2L\varepsilon]$ for each node. Therefore we have $|y(u) - y(v)| \leq 1 + 4L\varepsilon$ for each edge $\{u, v\}$ before normalisation and $|y(u) - y(v)| \leq 1/2 + 2L\varepsilon$ after normalisation. We guarantee a feasible solution by choosing $\varepsilon \leq 1/(4L)$.

Khuller et al. [18] show how to find an $\varepsilon$-maximal fractional matching in time $O(\log \varepsilon^{-1} + \log d)$ in graphs of maximum degree $d$. The virtual graph $F_h$ has a maximum degree of $d = O(L\Delta^L)$, and there is an $O(L)$ overhead in the simulation of $F_h$ in $G$. Finally, we have $L$ iterations in the algorithm; in total, the running time can be bounded by

$$O(L^2(\log \varepsilon^{-1} + \log d)) = O(L^3 \log \Delta).$$

This completes the proof of Theorem 7.

## 5 Conclusions

In this work, we have introduced the problem of finding a *locally optimal load balancing*, and studied its distributed time complexity. We have shown that the problem can be solved in a strictly local fashion, but to do it, one has to resort to algorithms that are very different from typical load-balancing strategies that are used in the literature. Among the key findings are:

- an $O(L)$-time algorithms for discrete load balancing in paths and cycles,
- a $\mathrm{poly}(L, \Delta)$-time algorithm for fractional load balancing in graphs of maximum degree $\Delta$.

The main open question is the distributed time complexity of the discrete load balancing problem. Our algorithm is local, but it has a running time exponential in $L$; the key question is whether $\mathrm{poly}(L, \Delta)$-time algorithms exist. We suspect that it is related to another long-standing open question—the distributed time complexity of bipartite maximal matching. Indeed, a $\mathrm{polylog}(\Delta)$-time algorithm for bipartite maximal matching would imply a $\mathrm{poly}(L, \Delta)$-time algorithm for discrete load balancing. We conjecture that such algorithms do not exist, but proving such lower bounds seems to be still beyond the reach of current techniques.

Another open question is the generalisation of the results from the LOCAL model to the CONGEST model. In particular, the polynomial-time algorithm for fractional load balancing heavily abuses the unlimited bandwidth of the LOCAL model, but it seems that there are no major obstacles for designing an analogous algorithm that works efficiently in the CONGEST model.

## Acknowledgements

We have discussed this problem and its variants over the years with numerous people, including, at least, Sebastian Brandt, Pierre Fraigniaud, Mika Göös, Petteri Kaski, Barbara Keller, Janne H. Korhonen, Juhana Laurinharju, Tuomo Lempiäinen, Christoph Lenzen, Joseph S. B. Mitchell, Pekka Orponen, Joel Rybicki, Thomas Sauerwald, Stefan Schmid, and Jara Uitto. Many thanks to all of you for your comments! Computer resources were provided by the Aalto University School of Science "Science-IT" project.

## References


[1] William Aiello, Baruch Awerbuch, Bruce Maggs, and Satish Rao. Approximate load balancing on dynamic and asynchronous networks. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 632–641. ACM Press, 1993. doi:10.1145/167088.167250.

[2] Richard Anderson, László Lovász, Peter Shor, Joel Spencer, Eva Tardos, and Shmuel Winograd. Disks, balls, and walls: analysis of a combinatorial game. *The American Mathematical Monthly*, 96(6):481–493, 1989. http://www.jstor.org/stable/2323970.

[3] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999. doi:10.1137/S0097539795288490.

[4] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality: An explanation of the 1/f noise. *Physical Review Letters*, 59(4):381–384, 1987. doi:10.1103/PhysRevLett.59.381.

[5] Paul Bogdan, Thomas Sauerwald, Alexandre Stauffer, and Sun He. Balls into bins via local search. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 16–34. Society for Industrial and Applied Mathematics, 2013. doi:10.1137/1.9781611973105.

[6] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006. doi:10.1109/TIT.2006.874516.

[7] Karl Bringmann, Thomas Sauerwald, Alexandre Stauffer, and He Sun. Balls into bins via local search: cover time and maximum loads. In *Proc. 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, pages 187–198, 2014. doi:10.4230/LIPIcs.STACS.2014.187.

[8] Andrzej Czygrinow, Michał Hańćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. Distributed 2-approximation algorithm for the semi-matching problem. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, volume 7611 of *Lecture Notes in Computer Science*, pages 210–222. Springer, 2012. doi:10.1007/978-3-642-33651-5_15.

[9] Deepak Dhar. Theoretical studies of self-organized criticality. *Physica A*, 369(1):29–70, 2006. doi:10.1016/j.physa.2006.04.004.

[10] Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost stable matchings by truncating the Gale–Shapley algorithm. *Algorithmica*, 58(1):102–118, 2010. doi:10.1007/s00453-009-9353-9. arXiv:0812.4893.

[11] Bhaskar Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *Journal of Computer and System Sciences*, 53(3):357–370, 1996. doi:10.1006/jcss.1996.0075.

[12] Bhaskar Ghosh, F. T. Leighton, Bruce Maggs, S. Muthukrishnan, C. Greg Plaxton, R. Rajaraman, Andréa W. Richa, Robert E. Tarjan, and David Zuckerman. Tight analyses of two local load balancing algorithms. *SIAM Journal on Computing*, 29(1):29–64, 1999.

[13] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 219–225. Society for Industrial and Applied Mathematics, 1998.

[14] Nicholas J. A. Harvey, Richard E. Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal of Algorithms*, 59(1):53–78, 2006. doi:10.1016/j.jalgor.2005.01.003.

[15] Leo P. Kadanoff, Sidney R. Nagel, Lei Wu, and Su-min Zhou. Scaling and universality in avalanches. *Physical Review A*, 39(12):6524–6537, 1989. doi:10.1103/PhysRevA.39.6524.

[16] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, 16(4–5):517–542, 1996. doi:10.1007/s004539900063.

[17] Krisnaram Kenthapadi and Rina Panigrahy. Balanced allocation on graphs. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA 2006)*, pages 434–443. Society for Industrial and Applied Mathematics, 2006. doi:10.1145/1109557.1109606. arXiv:cs/0510086.

[18] Samir Khuller, Uzi Vishkin, and Neal Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *Journal of Algorithms*, 17(2):280–289, 1994. doi:10.1006/jagm.1994.1036.

[19] S. Muthukrishnan, Bhaskar Ghosh, and Martin H. Schultz. First- and second-order diffusive methods for rapid, coarse, distributed load balancing. *Theory of Computing Systems*, 31(4):331–354, 1998. doi:10.1007/s002240000092.

[20] David Peleg and Eli Upfal. The token distribution problem. *SIAM Journal on Computing*, 18(2):229–243, 1989. doi:10.1137/0218015.

[21] Yuval Rabani, Alistair Sinclair, and Rolf Wanka. Local divergence of Markov chains and the analysis of iterative load-balancing schemes. In *Proc. 39th Annual Symposium on Foundations of Computer Science (FOCS 1998)*, page 694. IEEE, 1998. doi:10.1109/SFCS.1998.743520.

[22] Thomas Sauerwald and He Sun. Tight bounds for randomized load balancing on arbitrary network topologies. In *Proc. 53rd Annual Symposium on Foundations of Computer Science (FOCS 2012)*, pages 341–350. IEEE, October 2012. doi:10.1109/FOCS.2012.86.

[23] Alistair Sinclair and Mark Jerrum. Approximate counting, uniform generation and rapidly mixing Markov chains. *Information and Computation*, 82(1):93–133, 1989. doi:10.1016/0890-5401(89)90067-9.

[24] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4):568–589, 2003. doi:10.1145/792538.792546.