The Steady-States of Splitter Networks

Basile Couëtoux
Aix-Marseille Université, CNRS, LIS, Marseille, France

Bastien Gastaldi
Télécom SudParis, Institut Polytechnique de Paris, Evry, France

Guyslain Naves
Aix-Marseille Université, CNRS, LIS, Marseille, France

Abstract

We introduce splitter networks, which abstract the behavior of conveyor belts found in the video game Factorio. Based on this definition, we show how to compute the steady-state of a splitter network. Then, leveraging insights from the players community, we provide multiple designs of splitter networks capable of load-balancing among several conveyor belts, and prove that any load-balancing network on n belts must have $\Omega(n \log n)$ nodes. Incidentally, we establish connections between splitter networks and various concepts including flow algorithms, flows with equality constraints, Markov chains and the Knuth-Yao theorem about sampling over rational distributions using a fair coin.

2012 ACM Subject Classification Theory of computation → Network flows; Mathematics of computing → Network flows; Mathematics of computing → Graph algorithms; Theory of computation → Random walks and Markov chains

Keywords and phrases Factorio, splitter networks, flow, balancer, steady-state

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.9

Related Version *Extended Version*: https://arxiv.org/abs/2404.05472

1 Introduction

The transportation of materials or data within various networks represents an inexhaustible source of mathematical problems, which has lead to almost as many solutions, theories and algorithms. These advancements have brought about significant improvements across diverse fields including supply chain management, logistics, network optimization. Transportation also serves as a central component in numerous games, as evidenced by the transportation category on BoardGameGeek which lists almost two thousand games [3]. In Factorio [24], a video game published in 2020 by Wube Software, players must mine natural resources to feed a rocket-building factory on an hostile planet. A major part of the gameplay involves the movement of resources within the factory, employing various mechanism: robotic arms, conveyor belts, drones or trains.

In this work, we study the conveyor belts of Factorio. An item placed on a belt will move at a constant speed toward the end of the belt, until it reaches that end, or is blocked by an item preceding it. Belts in Factorio can be combined using a splitter, connecting one or two incoming belts to one or two outgoing belts. A splitter takes items from the incoming belts and places them on the outgoing belts, trying to split the flow as fairly as possible between the incident belts, while maximizing the throughput. Given the scale of a typical Factorio game, players frequently encounter the need to balance the loads across multiple belts, and the community has devised numerous efficient networks to address this load-balancing problem.

An intriguing aspect of Factorio is its encouragement for players to construct vast systems of automation, requiring intensive planning and optimization. Ultimately, the limiting factor arises from the CPU load generated by game state updates. Consequently, players are

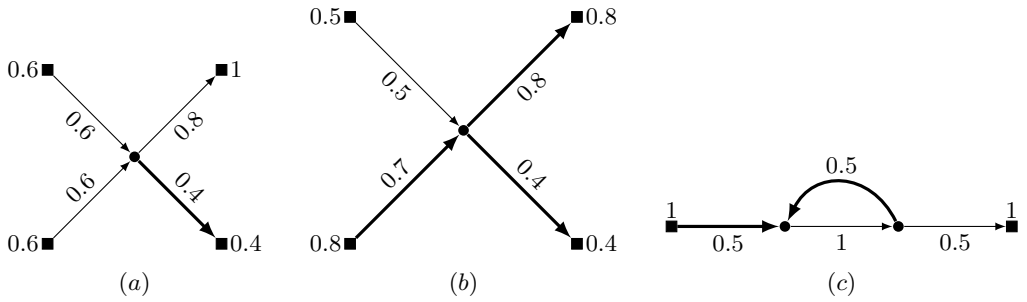

Figure 1 Three splitter networks given with capacities and associated steady-states. Splitter will be represented by circle vertices, terminals by square vertices. Each terminal is tagged by its capacity, and each arc by its throughput. Saturated arcs are bolder than fluid arcs.

incentivized to prioritize resource efficiency, particularly concerning gameplay elements that entail frequent computations such as splitters. This motivates the minimization of the number of splitters in load-balancing networks.

Our goal is two-fold: first we model the steady-state of a network of splitters. The network of conveyor belts is abstracted as a directed graph, with nodes corresponding to splitters and arcs to belts. A steady-state is a throughput function on the arcs; a circulation with additional constraints to capture the fact that splitters are fair and locally optimizing. We present two polynomial-time algorithms for computing a steady-state in a splitter network. An analogy is made with two classical maximum-flow algorithms: the blocking-flow algorithm [8] and the push-relabel algorithm [10]. In contrast to maximum flows, the primary challenge arises when a belt reaches full capacity, as its supplying splitter may no longer stay both fair and maximizing. In that case, the splitter is allowed to become unfair, but that decision changes the constraints applied to the flow, making the problem fundamentally non-convex. In a second part, we showcase various load-balancing network designs sourced from the Internet, formalizing concepts defined by the players community. Furthermore, we prove that those designs approach optimality. Specifically, we prove that any balancing network on n belts must have $\Omega(n \log n)$ splitters, by exhibiting a relation with the problem of sampling the uniform distribution over a set of n elements using only a fair coin. The core design is the Beneš network, a circuit-switching network well-known in the field of telecommunication [1, 2].

The blocking-flow-like algorithm relies on finding circulations with equality constraints. A *circulation* on a directed graph is a flow without any excess at any vertex. Given a directed graph (G, A), we denote $\delta^+(v)$ and $\delta^-(v)$ the sets of outgoing and incoming arcs incident to a vertex v. Let $\mathcal{C}^=$ be a partition of A such that for each part $C \in \mathcal{C}^=$, there is some vertex v with $C \subseteq \delta^+(v)$. The $\mathcal{C}^=$-circulation problem is to decide whether there is a non-zero circulation f that is constant within each part. While this problem can easily be solved using linear programming, we require a good characterization of graphs admitting a $\mathcal{C}^=$-circulation, Additionaly a polynomial-time algorithm is needed to either construct a $\mathcal{C}^=$-circulation or identify an obstacle that prevents its existence. The algorithm relies on the computation of a stationary distribution of an auxiliary graph. In contrast, solving maximum integral flow problems with additional equality constraints is known to be NP-hard [17], even when the partition is exactly the sets of leaving arcs of each vertex [23, 18].

Sorting networks [12] and Beneš networks have topologies similar to splitter networks, with nodes of in-degree and out-degree 2. In microfluidics, mixing graphs are used to produce droplets of specific concentration, using devices that produces two identical droplets from two droplets of any concentration [7]. The concentration values on the arcs are subject to equality

constraints similar to those of splitter networks, but without a maximizing constraint. The topology of splitter networks is nonetheless more general than these examples, as splitter networks may have directed cycles, those being necessary in particular to achieve load-balancing with an arbitrary number of outputs.

In an answer to a question on the mathematics section of `stackexchange`, David Ketcheson attempted to model and compute the throughputs of splitter networks [11]. Rather than binary categorizing each belt as full or not, each arc is assigned a density and a velocity. The density will be monotonically increasing, and the velocity monotonically decreasing during the run of the algorithm, until a steady-state is reached. In fact the velocity increases only after the density reaches its maximum at one. Therefore this description is equivalent to our solution, which involves a throughput function and a set of full belts. Unfortunately his algorithm does not always terminate, and its solutions do not satisfy that splitters use their incoming belts fairly. Ketcheson also gave a procedure, albeit non-polynomial, to determine whether a network (not necessarily load-balancing) may limit throughput. Hovewer, this procedure is applicable only to networks without directed cycle. In [15], Leue modeled splitter networks using Petri nets, and uses model checking to check the load-balancing properties of some small networks.

The Factorio community is very active and creative. Players have designed load-balancing networks of various sizes, with efficient embeddings into the grid while respecting the constraints of the game. Additionally, they have developed general methods for constructing arbitrary large load-balancing networks. They introduced the concept of balancing networks, along with the more robust properties of being throughput unlimited or universal, and subsequently designed networks that exhibit these characterics. A notable example is the universal balancer presented by *pocarski* [20], although it uses non-fair splitters too; our universal balancer only uses fair splitters. They also discovered the relationship with Beneš networks. Factorio-SAT [21] is a project that uses a SAT-solver to find optimal embeddings of splitter networks in the grid. The project *VeriFactory* uses a SAT-solver to check various load-balancing properties of splitter networks [14]. Factorio belts are actually sufficiently complex to be Turing-complete [16]. There are many implementations of various devices inside Factorio, ranging from raytracers to programming language interpreters, using the diverse set of available gameplay mechanisms. Factorio has been the inspiration for several other academic works [22, 19, 4, 6, 9].

The rest of this paper presents an overview of the main concepts and results of this work. An extended version [5] will contain more details, proofs and additional results. Splitter networks and their steady states are defined in Section 2. Section 3 describes two algorithms to compute the steady-state of a splitter network. The concept of balancer is defined in Section 4, which also contains a presentation of some balancer designs. Section 5 describes how to derive a lower bound on the number of splitters in a balancer network. Finally in Section 6 we will present some perspectives.

2 Splitter networks and their steady-states

We start by modeling networks of conveyor belts and splitters by directed graphs, where each single belt is an arc, and each splitter is a node (thus abstracting the length of the belts).

▶ **Definition 1.** *A* splitter network *is a directed graph G (with possible loops or parallel arcs) whose vertex set can be partitioned into three sets $V(G) = I \uplus S \uplus O$ where*

- **(i)** *I is the set of* inputs*, and $d^+(i) = 1$, $d^-(i) = 0$ for any input i;*
- **(ii)** *O is the set of* outputs*, and $d^-(o) = 1$, $d^+(o) = 0$ for any output o;*
- **(iii)** *S is the set of* splitters*, and $d^-(s) = d^+(s) = 2$ for any splitter s.*

We will use the word *flow* to informally describe the material transported by the network, and *throughput* for the amount of flow going through the arcs. Our work aims to understand the throughputs inside a splitter network at steady state, when some maximum throughputs are forced on its inputs and its outputs, which are respectively the sources and sinks of the flow passing through the network. To this end we will consider capacity functions on the input and output. A capacity c on an input means that the input has an incoming flow of throughput c. The input will try to push that much into the network, but no more. A capacity c on an output means that the output will accept a maximum throughput of c. We consider that the maximum throughput of any arc is 1, with all belts being identical.

A splitter can be described using two operational rules. The first rule, which takes precedence, is to maximize the amount of flow that goes through it. The secondary rule is to be fair. A splitter is fair relatively to its outgoing arcs: it tries to push as much flow onto each of them. It is also fair relatively to its incoming arcs: it tries to pull as much flow from each of them. As the maximization rule takes precedence, it will not be fair when being unfair leads to higher throughput. For instance, consider the network in Figure 1 (a), depicting a network with a single splitter. As one of the output has a lower capacity, it pushes more flow toward the other output, thereby maximizing the total throughput, while still being as fair as possible as it minimizes the difference of throughputs on its outgoing arcs.

The throughput of an arc may reach a limit when its head is an output with a low capacity. For example in Figure 1 (a), an output of capacity 0.4 acts as a bottleneck. In other cases the head of an arc is a splitter, which itself is limited by what its outgoing arcs can accept. For example in Figure 1 (b), as all the outputs have reached their capacities, the splitter cannot accept more flow, even if the bottom input could provide even more flow. In terms of conveyor belts, some belts will initially receive more items that they can deliver, causing them to fill up. Once full, they can only accept from upstream as much as they deliver downstream, which may in turn limit throughputs upstream. We say that such belts are *saturated*.

The output capacities are not the only factor that limit the total throughput and create bottlenecks. This can be observed in Figure 1 (c). There, the rightmost splitter tries to be fair and send some of the flow back to the left. The leftmost splitter also tries to be fair, thus accept the flow coming from the right. This results in the stabilization into the given throughputs. This example illustrates that the throughput is not globally maximized, contrary to the expectation of a total throughput of 1 for this network. Instead, it is only 0.5.

The following definition formalizes the notions of capacity, throughput and saturations, as well as the behaviour of splitters related to the flow going through the network in a steady state.

▶ **Definition 2.** *Let $G = (I \uplus S \uplus O, E)$ be a splitter network, and let $c : I \cup O \to [0, 1]$ be the maximal* capacities *of each input and output node. A* steady-state *for (G, c) is a pair (t, F) where*

R1 *$t : E \to [0, 1]$ is the* throughput *function;*
R2 *$F \subseteq E$ is the set of* fluid arcs*, $E \setminus F$ is the set of* saturated *arcs;*
R3 *for each $i \in I$ with $\delta^+(i) = \{e\}$, $t(e) \leq c(i)$ and moreover if $e \in F$ then $t(e) = c(i)$;*
R4 *for each $o \in O$ with $\delta^-(o) = \{e\}$, $t(e) \leq c(o)$ and moreover if $e \notin F$ then $t(e) = c(o)$;*
R5 *for each $s \in S$, with $\delta^-(s) = \{e_1, e_2\}$ and $\delta^+(s) = \{e_3, e_4\}$, $t(e_1) + t(e_2) = t(e_3) + t(e_4)$;*
R6 *for any $e_1, e_2 \in E$ with $\{e_1, e_2\} = \delta^-(s)$ and $e_1 \notin F$, $t(e_1) \geq t(e_2)$;*
R7 *for any $e_1, e_2 \in E$ with $\{e_1, e_2\} = \delta^+(s)$ and $e_1 \in F$, $t(e_1) \geq t(e_2)$;*
R8 *for any $uv \in E \setminus F$ and $vw \in F$, $t(uv) = 1$ or $t(vw) = 1$.*

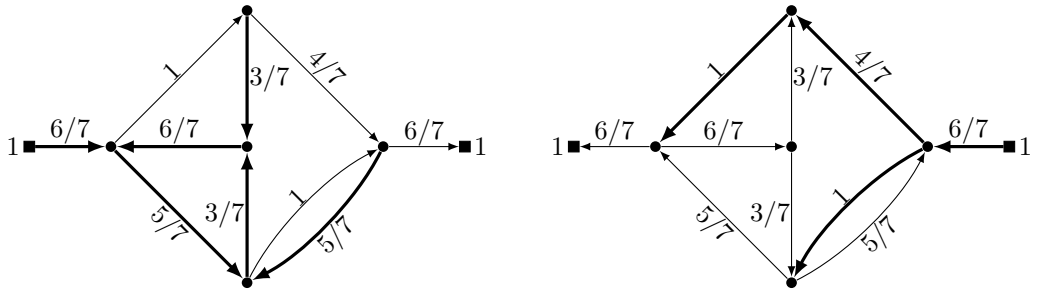

Figure 2 An example of steady-state in a moderately small network, and the reverse network with its steady-state obtained by reversal. Notice that the reversed steady-state satisfies rule 8 but not rule 9.

Rules 3 and 4 say that the throughputs are limited at each input and each ouput, and moreover, an input pushes as much flow as allowed by its capacity on a fluid arc. Similarly an output absorbs as much flow as allowed by its capacity from a saturated arc. Rule 5 imposes the conservation of flow. Rules 6 and 7 enforce the fairness constraints: a splitter consumes no less flow from a saturated arc than from another incoming arc. A saturated arc represents a belt that is full. Therefore, the splitter is not limited in how much flow it can pull from that arc, and thus cannot pull less than from the other incoming arc. Similarly it produces no less flow in a fluid outgoing arc than in another outgoing arc. In particular, if both incoming arcs are saturated, or if both outgoing arcs are fluid, they must have equal throughput, suggesting the following definition.

▶ **Definition 3.** *Given a splitter network $G = (I \uplus S \uplus O, E)$, and a set $F \subseteq E$ of fluid arcs, we say that two arcs $e, e' \in E$ are*

- in-coupled *if $e, e' \notin F$ and there is a splitter vertex $v \in S$ with $\delta^-(v) = \{e, e'\}$,*
- out-coupled *if $e, e' \in F$ and there is a splitter vertex $v \in S$ with $\delta^+(v) = \{e, e'\}$,*
- coupled *if they are in-coupled or out-coupled.*

Finally rule 8 imposes the maximization of the throughput by each splitter. Indeed, a saturated arc can provide more flow, while a fluid arc can absorb more flow. Thus, a steady-state cannot contain a saturated arc followed by a fluid arc. The only exception is when one of them already has a throughput of 1. We will sometimes replace rule 8 by a stronger maximization rule :

R9 for any arcs $uv \in E \setminus F$ and $vw \in F$, $t(vw) = 1$.

Rule 9 implies rule 8, and although the converse is not true, any steady-state can be modified into a steady-state for which rule 9 also holds.

The definitions of splitter networks and steady-states exhibit a remarkable symmetry. By reversing each arc, exchanging the role of inputs and outputs, and complementing the set of fluid arcs, a steady-state is transformed into a steady-state of the reverse graph, as seen in Figure 2.

For convenience, when defining or representing splitter networks, we will allow splitters with in-degree one or out-degree one (see Figure 2 for instance). This is justified by the fact that if a splitter s has in-degree one, we can add a dummy input node i with capacity $c(i) = 0$. An arc from i to s can then be added, that will always remain fluid. Similarly if s has out-degree one, we can add a dummy output node o with capacity $c(o) = 0$, and an always-saturated arc from s to o. The throughputs on those arcs are forced to be 0. Therefore it does not induce any new constraint on the non-dummy arcs as rules 6 and 7 are clearly true for those arcs.

Additionally, for convenience, for any input $i \in I$ with outgoing arc e, we note $t(i) := t(e)$, and similarly for any output $o \in O$ with incoming arc e, $t(o) := t(e)$. We also extend the capacities to arcs by setting $c(e)$ to be either $c(i)$ if $e \in \delta^+(i), i \in I$, or $c(o)$ is $e \in \delta^-(o), o \in O$, or 1 otherwise.

3 Existence and computation of steady-states

Let F be a fixed set of fluid arcs. Then the set of possible throughput functions t of a steady-state (t, F) can be described as a polyhedron. Indeed, each of the rules 1, 3, 4, 5, 6, 7 can be encoded by linear inequations. Rule 8 is non-convex, but we will later introduce its slight strengthening, rule 9. That stronger rule admits an encoding as a family of linear inequations. Thanks to linear programming, finding a steady-state thus reduces to finding a set of fluid arcs that admits a steady-state. Nevertheless, we still need to find F. We propose two algorithms to compute a steady-state, which relates to two families of maximum flow algorithm:

- a push-relabel-like algorithm, where we relax the conservation rule 5, thus defining a *pre-steady-state* by analogy with *pre-flows*. Given a set F, we use a linear program to compute an optimal pre-steady-state (t, F) (for some well-chosen objective), and prove that either (t, F) is a steady-state, or there is an arc $e \in F$ such that $(t, F \setminus e)$ is also a (non-optimal) pre-steady-state. Then after at most $|E|$ steps we get a steady-state;
- a blocking-flow-like algorithm, where we relax the rule 3 on input capacities, removing the requirement that an input whose throughput is less than its capacity must have a saturated outgoing arc. This defines the notion of *sub-steady-state*. Given a set F, we solve a linear system to find a sub-steady-state t, and prove once again that either (t, F) is a steady-state or there is an arc $e \in F$ such that $(t, F \setminus e)$ is a sub-steady-state.

The pre-steady-state algorithm is technically simpler but requires an LP-solver. The sub-steady-state only requires an algorithm to compute stationary distributions in directed graphs. We defer a complete presentation and proof of these algorithms to the extended version of this paper, and focus here on explaining the sub-steady-state algorithm.

▶ **Definition 4.** *Given $G = (I \uplus S \uplus O, E)$ a splitter network with capacities $c : I \uplus O \to [0, 1]$, a* sub-steady-state *for (G, c) is a pair (t, F) satisfying rules 1, 2, 4, 5, 6, 7 and the strong maximization rule 9, and for any $i \in I$ and $e \in \delta^+(i)$, $t(e) \leq c(i)$.*

The algorithm starts with the trivial sub-steady-state $(t : e \to 0, E)$, and will improve it iteratively until reaching a steady-state. At each iteration of the algorithm, we will be trying to increase the throughputs of the arcs without violating any rule. Unlike in maximum flows, we do not have the choice of which leaving arc to increase the flow on. Furthermore, rule 8 forces each splitter to send as much flow forward as possible. A non-obvious consequence is that, when increasing the input capacities, throughputs can only increase on fluid arcs, and can only decrease on saturated arcs. This suggests a definition of the residual graph for the sub-steady-state (t, F). Its vertex set is $\{z\} \cup S$, where z is obtained by identifying all the inputs and outputs into a single node. Its edge set contains some fluid arcs and the reverses of some saturated arcs.

Consider the splitters in Figure 3. We examine what happens when we increase the throughput on edge e_1 by $+\varepsilon$, or in case (d) when we decrease $t(e_3)$ by ε. In case (a), by rule 7, the throughputs on the two leaving arcs must stay equal, hence both increases by $\varepsilon/2$. In case (b), only the throughput of the fluid leaving arc e_4 can increase. In case (c), both leaving arc are saturated, the splitter cannot push more flow downward, hence it is

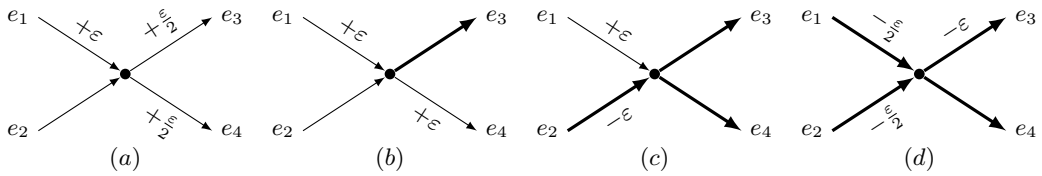

Figure 3 Four examples of throughput changes at a single splitter, depending on which arcs are fluid.

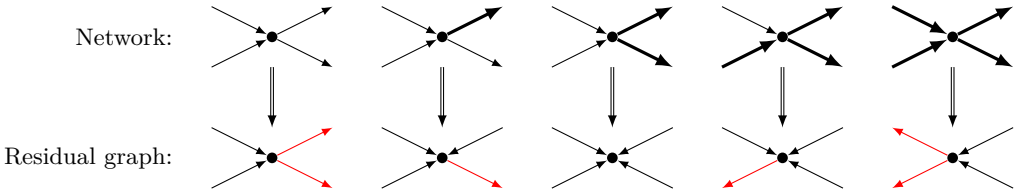

Figure 4 Configurations of splitters and the corresponding vertex in the residual graph. The outgoing arcs from a vertex of the residual graphs are highlighted in red: notice that in a sub-steady-state, the throughputs on these arcs must be equal.

forced to push back flow through its incoming saturated arcs. Thus $t(e_2)$ decreases while $t(e_1)$ increases, by no more than $(t(e_2) - t(e_1))/2$ because of rule 6. Finally in case (d), if we decrease $t(e_3)$, then $t(e_1)$ and $t(e_2)$ must decrease by half as much.

Case (c) presents a challenge due to rule 6, which imposes $t(e_1) \leq t(e_2)$. When $t(e_1) = t(e_2)$, the throughput of e_1 cannot increase, and the throughput of e_2 cannot decrease. We say that e_1 and e_2 are *tight*. In such a case, removing e_1 from F is allowed by rule 6. Fluid arcs e with $t(e) = c(e)$ or saturated arc with $t(e) = 0$ are also *tight*, since we cannot modify their throughput further. Then we define the edge-set of the residual graph to only contain non-tight fluid arcs and reverses of non-tight saturated arcs.

Due to the conservation rule 5, any iterative change to the throughputs of the network must be in accordance with a circulation of the residual graph. Because of rules 6 and 7, some arcs are constrained to have the same throughput. Therefore the chosen circulation itself has similar constraints. This is illustrated in Figure 4, where the arcs that have equal throughput are highlighted in the residual graph. As may be readily checked, those constraints are exactly set on the leaving arcs in the residual graph of each vertex corresponding to a splitter. As for the special vertex z, obtained from the identification of the inputs and the outputs, we may non-deterministically select one of its leaving arc. Then we force all other arcs leaving z to have zero flow, by removing those arcs from the residual graph. From the residual graph, we compute a circulation satisfying each equality constraint. First compute a stationary distribution of a random walk on the residual graph. Then assign to each arc the probability of being the next arc in a random walk from that distribution. This results in a so-called *stationary circulation* (see Figure 5). One must be careful if the residual graph is not strongly connected. Then either we can find a strongly connected subgraph induced by the leaving arcs of some subset of vertices, or the residual graph contains a sink (as in Figure 6). In the former case we can still find a circulation, while in the latter case, we will be able to remove some arc from F.

Once a circulation is found, we increase the throughput as much as possible. This process will result in the creation of at least one sink in the updated residual graph. We show that when the residual graph contains a sink, some arc can be safely removed from F and becomes

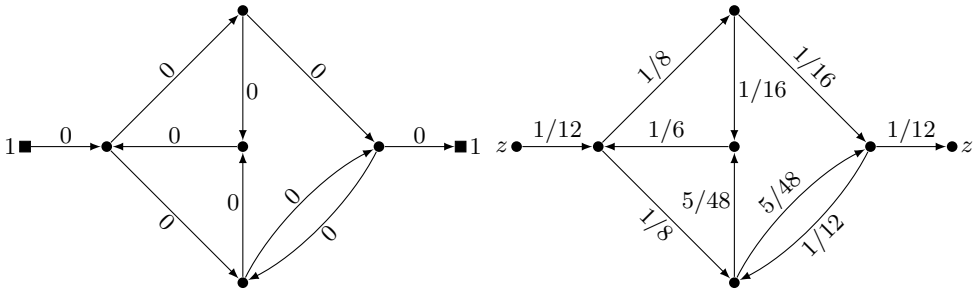

Figure 5 Starting from a trivial sub-steady-state, we compute a residual graph and a stationary circulation in this graph (the two vertices marked z should be identified). Then we increase the throughputs accordingly, as much as possible without violating a sub-steady-state rule, by adding $\lambda = 6$ times the circulation at which point some edge reaches its capacity (see Figure 6).

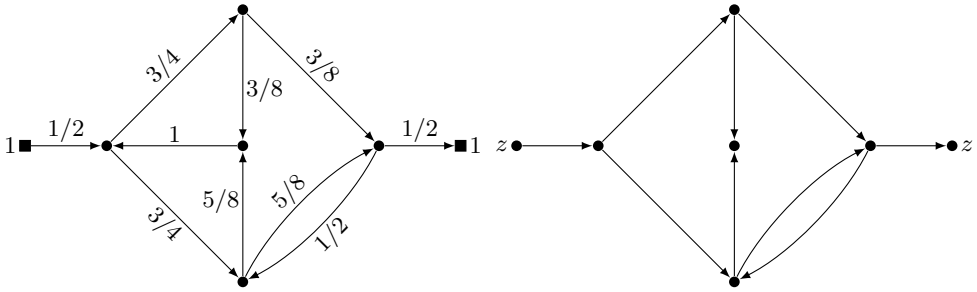

Figure 6 We compute a new residual graph, which does not contain the arc with throughput 1, since this arc cannot increase. Then the existence of a sink prevents us to find a stationary circulation in this residual graph (the two vertices marked z should be identified). We remove from F the incoming arc to the sink with highest throughput, and go to the next iteration.

saturated. This bounds the number of steps until the algorithm stops, when z itself becomes a sink. At this point, any arc leaving an input node is either at full capacity or is saturated. Hence rule 3 is satisfied, (t, F) is a steady-state. Summarizing the discussion, we get:

▶ **Theorem 5.** *There is an algorithm that given a splitter network $G = (I \uplus S \uplus O, E)$ with capacities $c : I \uplus O \to [0, 1]$, finds a steady-state (t, F) in time $O(|S|^2 + |S| \operatorname{sd}(G_z))$, where G_z is the graph obtained by identifying $I \cup O$ into a single vertex z, and $\operatorname{sd}(G_z)$ denotes the time to compute a stationary distribution on any orientation of a subgraph of G_z.*

Steady-states are not unique: a directed cycle with no input or output can have any constant throughput on all its arcs. Figure 7 showcases a more interesting network, having one input, one output, and many possible steady-states. However, in this example, all steady-states have the same throughputs on the inputs and outputs. Is there a network with two steady-states having different throughputs on their inputs and outputs? We conjecture that this cannot happen: steady-states are unique up to minor modifications, as in Figure 7. Those modifications would be adding or removing some arcs from F, and adding or subtracting a circulation from the residual graph that leaves the inputs and outputs unchanged.

4 Balancers

We now define load-balancing networks and their properties. The goal of a load-balancing network is to divide some input flow evenly between several output belts. In the simplest case, the output belts can receive an arbitrarily large flow (up to the capacity of the belt). In

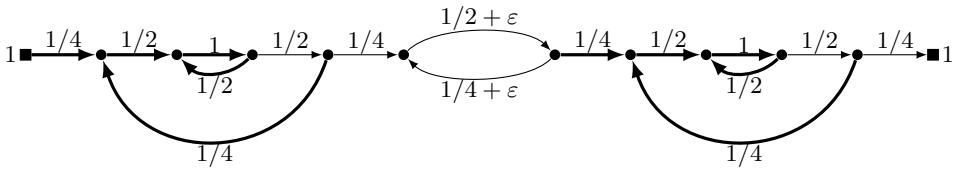

Figure 7 A network having several steady-states. Any value for ε between 0 and $\frac{1}{2}$ gives a steady-state.

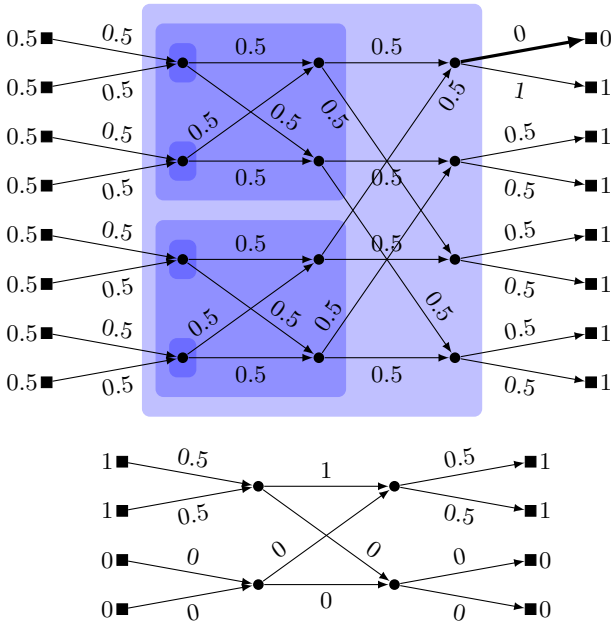

Figure 8 On top, the simple balancer of order 3, with a steady-state that is not balanced when some output capacity is not 1. The capacity of each input (resp. output) is given at their left (resp. right). Below, a simple balancer of order 2, with a steady-state with total throughput less than both the total input capacity and the total output capacity.

more general cases, some outputs may be restricted but we still want the flow to be divided as evenly as possible, without limiting the total throughput available. We distinguish three properties of load-balancing networks. The first of these properties considers networks where the output capacities are not constrained.

▶ **Definition 6.** *A splitter network* $G = (I \uplus S \uplus O, E)$ *is a* balancer *if for any* $c : I \uplus O \to [0, 1]$ *such that for each output* $o \in O$, $c(o) = 1$, *there is a steady-state* (t, F) *for* (G, c) *with* t *constant on* $\delta^-(O)$. *An* (n, p)*-balancer is defined as a balancer with* $|I| = n$ *inputs and* $|O| = p$ *outputs.*

When $|I| = |O| = 2^k$, the *simple balancer of order* k is a balancer network. It can be defined recursively: a simple balancer of order $k + 1$ is made from two simple balancers of order k in parallel. We identify each pair of outputs with equal index from the two balancers, creating a new splitter whose leaving arcs go to new output nodes. The recursive process is highlighted by blue boxes in Figure 8. A drawback of the simple balancer occurs when the output capacities are not uniformly 1. Then the balancing property is lost, as can be seen on the network in the top half of Figure 8.

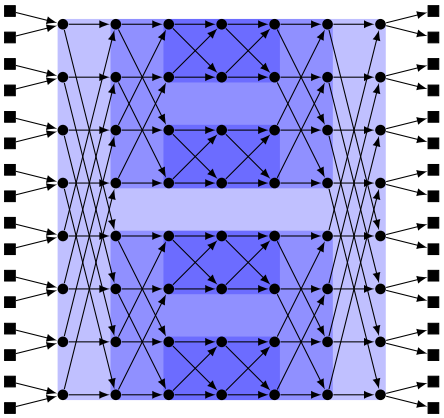

Figure 9 A Beneš network of order 4 with the recursive structure being made explicit.

Another limitation of simple balancers is that the total throughput at steady-state is not as much as we could expect. A simple upper bound on the total throughput is $\min\{c(I), c(O)\}$. It is reasonable to expect from a load-balancing network to always reach that bound. However, simple balancers do not have this property, as shown by the example on the bottom half of Figure 8. Improving over the definition of simple balancer, the concept of throughput-unlimited balancer imposes a maximized global throughput.

▶ **Definition 7.** *A balancer* $G = (I \uplus S \uplus O, E)$ *is* throughput-unlimited *if for any* $c : I \uplus O \to [0, 1]$*, there is a steady-state* (t, F) *for* (G, c) *such that total throughput* $t(I) = t(O)$ *is maximized at* $\min\{c(I), c(O)\}$*.*

Notice that it has to be balancing only when the output capacities are uniformly 1. Beneš networks are throughput-unlimited networks with $|O| = |I| = 2^k$. They can be described as gluing two simple balancers, where the second balancer is reversed, see Figure 9. Observe that Beneš networks are their own reverses.

On the negative side, Beneš network are still not balancing when output capacities are not uniformly 1, for instance one could extend the steady-state in the network on the left side of Figure 8 to a steady-state in a Beneš network with the same throughputs. This calls for a stronger property, that a network should be load-balancing and throughput-unlimited for any capacity function. This is the notion of universal balancer.

▶ **Definition 8.** *A splitter network* $G = (I \uplus S \uplus O, E)$ *is* universally balancing *if for each capacity* $c : I \uplus O \to [0, 1]$*, there is a steady-state* (t, F) *and* $\alpha, \beta \in \mathbb{R}_{\geq 0}$ *such that*

(i) *for each input* i*,* $t(\delta^+(i)) = \min\{c(i), \alpha\}$*,*
(ii) *for each output* o*,* $t(\delta^-(o)) = \min\{c(o), \beta\}$*.*
(iii) *the total throughput* $T := t(\delta^+(I))$ *equals* $\min\{c(I), c(O)\}$*.*

In the extended version of this work, we will show how to build a universal balancer with $|I| = |O| = 2^k$. From such a universal balancer, by ignoring any set of inputs and outputs (setting their capacities to 0), we can make balancers with arbitrary numbers of inputs and outputs. We will also prove that every balancer presented here contains $\Theta(n \log n)$ splitters where n is the number of inputs and outputs.

▶ **Proposition 9.** *The number of splitters in the simple balancer, Beneš network and universal network of order* k *are respectively* $S(k) = k \cdot 2^{k-1}$*,* $B(k) = (2k - 1) \cdot 2^{k-1}$*, and* $U(k) = (k + 1)2^{k+2}$*.*

5 Lower bounds on the number of splitters

Our next goal is to provide an $\Omega((n+p)\log(n+p))$ lower bound on the number of splitters in a (n,p)-balancer. We begin with what may seem as an unrelated problem: sampling in a discrete probability distribution. Given a fair coin that can be tossed arbitrarily often, how to choose an outcome in $\{1,\dots,d\}$, with probabilities given by a distribution $\pi \in [0,1]^d$? First, consider the case when $\pi(i)$ is a rational for each $i \in \{1,\dots,d\}$, say $\pi(i) = p_i/q$ where q is a common denominator. Then a sequence of coin tossing can be described as a (possibly infinite) binary decision tree, with each leaf labeled with a sampled value. Here we present a construction of such a tree. Start from a single vertex, which serves as the root. Grow the tree in repeated iterations. At each iteration, add two children to every unlabelled leaf. As soon as the deepest level of the tree contains at least q leaves, label p_i of these leaves with i, for each $i \in \{1,\dots,d\}$. Once labeled, each leaf becomes definitive and will not grow anymore. The process goes on by once again growing the unlabelled leaves, as long (possibly infinitely) as some unlabelled leaf exists. After the tree is completed, the tree can be optimized using a simple trick repeated multiple times. If at any depth d, two leaves share a common label, move them under a common parent, then replace these two leaves with a single leaf at depth $d-1$ bearing the same label. This process can be generalized to irrational probabilities, and gives a sampling algorithm that minimizes the number of coins tossed:

▶ **Theorem 10** ([13]). *Let $\pi \in [0,1]^d$ a discrete probability distribution (so $\mathbb{1}\pi = 1$). Then the minimum expected number of coin tosses necessary to sample an element with probability distribution π is $\sum_{i=1}^{d}\sum_{k\in\mathbb{N}} \frac{k}{2^k}\mathrm{binary}_k(\pi_i)$. This minimum is achieved by a binary decision tree where at each depth k and for each $i \in [\![1,d]\!]$, the number of leaves with label i is $\mathrm{binary}_k(\pi_i)$ (the value of the bit of weight 2^k in the binary expansion of π_i).*

Consider a splitter network, and think of the flow as discrete, arbitrarily small items. An item enters the network from some input, then meets splitters repeatedly until reaching an output. When an item arrives at a splitter with both outgoing arcs being fluid, it will continue on any of the two outgoing arcs, without preference for one over the other because the splitter is fair. It implies that, from the perspective of this single item, the splitter network behaves like a coin-tossing network, with each splitter corresponding to a coin toss. If the network is a balancer, the sampled distribution is the uniform distribution on O.

Formally, when all the arcs remains fluid, increasing a single input capacity from 0 to 1 results in a non-decreasing throughput on each arc. Because all arcs are still fluid, the sub-steady-state algorithm performs a single iteration. Therefore the increase in throughputs follows a single stationary circulation. As illustrated on Figure 10, it is obtained from the embedding of a binary decision tree T onto the splitter network. The increase in throughput on an arc e is the sum of probabilities of the edges mapped to e. Furthermore, in a balancer network, the increase of throughput is the same on every output. This implies that, as we progressively increase each input capacity from 0 to 1, each binary decision tree must uniformly sample from O.

In each binary decision tree, label each edge e with the probability of its usage during sampling. The sum of these labels represents the expected number of tosses, and can be bounded as shown in Theorem 10. When mapped into the splitter network, for an arc e, the sum of these labels on each edge of the tree mapped to e is the additional throughput on e. By summing over all the binary decision tree, we get that the sum of all labels is at most the number of outgoing arcs of all splitters, that is $2|S|$. Applied on balancers, it yields:

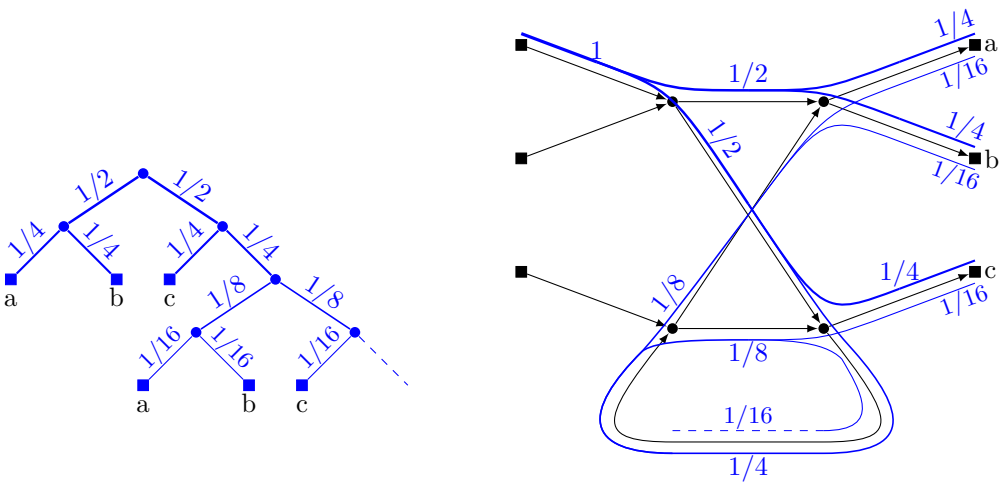

Figure 10 The infinite decision tree (in blue) used to sample uniformly over a three-element set $\{a, b, c\}$ can be embedded from any input into a $(3, 3)$-balancer. Moreover, the sum of the probabilities of the 3 trees, one from each input, will be at most one on any arc, which shows that this network is indeed a simple balancer.

▶ **Theorem 11.** *Let $G = (I \uplus S \uplus O, E)$ be an (n, p)-balancer, such that when all input capacities are 1, the steady-state has no saturated arc. Then*

$$|S| \geq \frac{1}{2}|I||O| \sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right)$$

For a balancer with $|I| = |O| = 2^k$, since $\sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right) = \frac{k}{2^k}$, we get a lower bound of $k2^{k-1}$ splitters, matching the value of $S(k)$. Therefore the simple balancer of order k is optimal among all balancer networks without any saturated arcs in their steady-states. By extending this argument to steady-states with saturated arcs, we can remove that restriction, albeit at the cost of halving the lower bound.

Consider the various configurations of fluid and saturated arcs incident to a splitter, illustrated in Figure 3. If a splitter has two fluid outgoing arcs, any additional flow is evenly distributed between the two outputs, akin to the probabilities of a coin toss. If a splitter has two incoming saturated arcs, by rule 8, its outgoing arcs are saturated or at full capacity. In an augmenting circulation, the throughput on those arcs may only decrease by the same quantity by rule 6: the splitter still acts as a coin toss, but on the flow that is pushed back. Otherwise, a positive change in throughput on an incoming arc will be followed by an increase on a single outgoing fluid arc or a decrease on a single incoming saturated arc. Similarly a negative change of throughput on an outgoing saturated arc will impact only one other arc. Any additional unit of flow entering the splitter would be routed deterministically. Therefore, in the embedding of a binary decision tree into the splitter network, a node cannot be mapped to such a vertex, and no coin toss occurs here. Thus any splitter, depending on which of its incident arcs are fluid, acts as either a coin toss or a deterministic router. Thus, even in the presence of saturated arcs, we can embed a binary decision tree, by mapping each edge to a directed path in the residual graph. The inner nodes of any such path are *deterministic* splitters, while its extremities are *tossing* splitters. As a consequence of the sub-steady-stat algorithm, the throughput on each arc increases until it becomes saturated,

then decreases. Therefore its throughput varies by at most 2 during the whole algorithm. This limits the extent to which an arc can be utilized by the embeddings of binary decision trees, leading us to the following conclusion:

▶ **Theorem 12.** *Let $G = (I \uplus S \uplus O, E)$ be an (n, p)-balancer. Then*

$$|S| \geq \frac{1}{4}|I||O| \sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right)$$

6 Perspectives

We formalized splitter networks and their steady-states, and presented various load-balancing designs. The ability to design universal balancers enables the simulation of networks with integral capacities: each arc is replicated according to its capacity, and each splitter is replaced by a universal balancer. A universal balancer is fair by the balancing property, and maximizing by the unlimited-throughput property, effectively generalizing splitters. Our definition of splitter network can also be extended to support arc capacities natively, with most of the proofs requiring only minor modifications. In an extended version of this paper [5], we will demonstrate how to simulate any rational capacity. Given an arbitrary rational value between 0 and 1, we will design a splitter network, with a single input and a single output, and achieving this value as maximal throughput. However, simulating irrational capacities is not feasible, as the steady-state throughput is a solution to a linear system of inequations.

Although our continuous model is convenient for modeling the expected throughput of splitter networks, Factorio's belt systems operates discretely. Therefore, the observed throughputs in Factorio's splitter networks are only approximations of those theorized by our model. Further investigation into the disparities between the discrete and continuous splitter networks is necessary to accurately apply our findings to Factorio.

We have let several questions unanswered. The most fundamental remains regarding the uniqueness of the steady-state throughput. While it is possible for a single splitter network to admit multiple steady-states, we have yet to encounter a network with two steady-states that yield different throughputs on their outputs.

Our lower bounds for the number of splitters in balancers have a constant multiplicative gap across all designs, indicating they are not tight. For simple balancers of order k, this gap is closed when we forbid saturated arcs in the steady-state of the balancer. Consequently, leveraging saturation is necessary to further reduce the number of splitters in load-balancing networks. Furthermore, it is worth investigating stronger lower bounds in the context of universal balancers.

Factorio allows splitters to be configured to prioritize either an outgoing arc, or an incoming arc. Utilizing this feature, the universal network described in [20] achieves a significantly smaller size compared to our design. Our technique still establishes a lower bound on the number of fair splitters. In general, what is the minimum size achievable for networks utilizing these more general splitters? It is straightforward to extend the definition of steady-state to accomodate unfair splitters. Additionally, in the extended version of this paper, we will provide complexity results for the problem of global throughput maximization, when we can choose which arcs to prioritize in each splitter or a subset of those splitters.

As a last series of questions, consider a network whose steady-state, when all inputs and outputs have capacity 1, has no saturated arcs. If the augmenting flow from any single input is uniformly distributed across the outputs, then the network is a balancer. This provides a polynomial-time procedure for determining whether a network is a balancer, subject to the absence of saturation. Is it feasible to devise a general procedure to decide whether a splitter network is balancing, throughput unlimited or universal?

References

1 Václav E Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, 1962.

2 Václav E Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964.

3 BoardGameGeek. Boardgame category: transportation. `https://boardgamegeek.com/boardgamecategory/1011/transportation`.

4 Bonnie S. Boardman and Caroline C. Krejci. Simulation of production and inventory control using the computer game factorio. In *ASEE 2021 Gulf-Southwest Annual Conference*, 2021.

5 Basile Couëtoux, Bastien Gastaldi, and Guyslain Naves. The steady-states of splitter networks, 2024. URL: `https://arxiv.org/abs/2404.05472`.

6 Chase Covello, Hyunjang Jung, and Bryan C. Watson. Using graph theory to investigate the role of expertise on infrastructure evolution: A case study examining the game factorio. In *Conference on Systems Engineering Research*, pages 297–311. Springer, 2023.

7 Miguel Coviello Gonzalez and Marek Chrobak. Towards a theory of mixing graphs: A characterization of perfect mixability. *Theoretical Computer Science*, 845:98–121, 2020.

8 Yefim A. Dinits. The method of scaling and transportation problems. *Studies in Discrete Mathematics, Moscow*, pages 46–57, 1973.

9 Shivam Duhan, Chengming Zhang, Wenyu Jing, and Mingqi Li. Factory optimization using deep reinforcement learning ai. *Purdue Undergraduate Research Conference*, 57, 2019.

10 Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

11 Ketcheson, David. Mathematics Stackexchange: Belt Balancer problem (Factorio). `https://math.stackexchange.com/questions/1775378/belt-balancer-problem-factorio`.

12 Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3*. Pearson Education, 1998.

13 Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In JF Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Addison-Wesley, 1976.

14 Legnagi, Alessandro and Montini, Axel. VeriFactory. `https://github.com/alegnagi/verifactory/`.

15 Andre Leue. *Verification of Factorio Belt Balancers using Petri Nets*. PhD thesis, Bachelorarbeit, Darmstadt, Technische Universität Darmstadt, 2021.

16 MatthaeusHarris. Factorio belts are Turing-complete. `https://www.reddit.com/r/factorio/comments/lc25cx/factorio_belts_are_turing_complete/`.

17 Carol A. Meyers and Andreas S. Schulz. Integer equal flows. *Operations Research Letters*, 37(4):245–249, 2009.

18 Amandeep Parmar. *Integer programming approaches for equal-split network flow problems*. PhD thesis, Georgia Institute of Technology, 2007.

19 Sean Patterson, Joan Espasa, Mun See Chang, and Ruth Hoffmann. Towards automatic design of factorio blueprints. *arXiv preprint arXiv:2310.01505*, 2023.

20 pocarski. Universal 8-8: Perfectly Balanced, as All Things Should Be. `https://web.archive.org/web/20230922022806/https://alt-f4.blog/ALTF4-27/`.

21 R-O-C-K-E-T. factorio-SAT: Enhancing the Factorio experience with SAT solvers. `https://github.com/R-O-C-K-E-T/Factorio-SAT`.

22 Kenneth N. Reid, Iliya Miralavy, Stephen Kelly, Wolfgang Banzhaf, and Cedric Gondro. The factory must grow: automation in factorio. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 243–244, 2021.

23 K. Srinathan, Pranava R. Goundan, MVN Ashwin Kumar, R. Nandakumar, and C. Pandu Rangan. Theory of equal-flows in networks. In *Computing and Combinatorics: 8th Annual International Conference, COCOON 2002 Singapore, August 15–17, 2002 Proceedings 8*, pages 514–524. Springer, 2002.

24 Wube Software. Factorio. `https://www.factorio.com/`.