An $O(n \log n)$ Algorithm for a Load Balancing Problem on Paths

Nikhil R Devanur[1] and Uriel Feige[2⋆]

[1] Microsoft Research, Redmond, WA. nikdev@microsoft.com
[2] Weizmann Institute of Science, Rehovot, Israel. uriel.feige@weizmann.ac.il

Abstract. We study the following load balancing problem on paths (PB). There is a path containing n vertices. Every vertex i has an initial load h_i, and every edge $(j, j+1)$ has an initial load w_j that it needs to distribute among the two vertices that are its endpoints. The goal is to distribute the load of the edges over the vertices in a way that will make the loads of the vertices as balanced as possible (formally, minimizing the sum of squares of loads of the vertices). This problem can be solved in polynomial time, e.g, by dynamic programming. We present an algorithm that solves this problem in time $O(n \log n)$.
As a mental aide in the design of our algorithm, we first design a hydraulic apparatus composed of bins (representing vertices), tubes (representing edges) that are connected between bins, cylinders within the tubes that constrain the flow of water, and valves that can close the connections between bins and tubes. Water may be poured into the various bins, to levels that correspond to the initial loads in the input to the PB problem. When all valves are opened, the water flows between bins (to the extent that is feasible due to the cylinders) and stabilizes at levels that are the correct output to the respective PB problem. Our algorithm is based on a fast simulation of the behavior of this hydraulic apparatus, when valves are opened one by one.

1 Introduction

We describe a problem that we shall call *Path Balancing* (PB).

An instance of PB is a path on n vertices. Every vertex v_i has an initial height $0 \leq h_i \leq 1$. Every edge $e_j = (v_j, v_j + 1)$ has weight $0 \leq w_j \leq 1$. A feasible solution splits the weight of every edge in an arbitrary way between its endpoints, thus contributing to the heights of its endpoints. The goal is to make the vector of heights as balanced as possible. (Here and elsewhere, heights, in contrast to initial heights, will refer to the heights of vertices in a solution and not in the input.) In a perfectly balanced solution all heights are identical. When there is no perfectly balanced solution, the notion of balance that we use is that of minimizing the sum of squares of the heights.

⋆ The author holds the Lawrence G. Horowitz Professorial Chair at the Weizmann Institute. Work supported in part by The Israel Science Foundation (grant No. 873/08).

The problem above can be formulated as a convex program as follows. For $1 \le i \le n-1$, let x_i denote the amount of weight that edge e_i gives to vertex v_i. The rest of the weight of e_i which is $w_i - x_i$ is given to vertex v_{i+1}. Then there are $n-1$ constraints of the form $0 \le x_i \le w_i$, and the objective function is to minimize $(h_1+x_1)^2+\sum_{i=2}^{n-1}(h_i+(w_{i-1}-x_{i-1})+x_i)^2+(h_n+(w_{n-1}-x_{n-1}))^2$. For simplicity of notation, we shall introduce fictitious $x_n = 0$, $w_0 = 0$ and $x_0 = 0$, and let $h(v_i)$ denote the value of $h_i + (w_{i-1} - x_{i-1}) + x_i$. Hence the objective function can be written as $\sum_{i=1}^{n}(h(v_i))^2$. (Actually, the optimal solution with the objective being any convex function of the $h(v_i)$'s will turn out to be the same, we just choose $h(v_i)^2$ for convenience.)

We are interested in efficient algorithms for PB. Since it can be formulated as a convex program, it follows that it can be solved in polynomial time. In fact, a natural dynamic programming approach gives a running time of $O(n^3)$ (Appendix B) and with some effort, one can obtain an algorithm that runs in time $O(n^2)$ (Appendix C). In this paper we show that this problem can be solved in $O(n \log n)$ time. In measuring the running time of algorithms we shall count the number of basic operations that they perform, without worrying too much about the cost of each operation (e.g., the cost of basic arithmetic operations as a function of the precision needed), or about the data structures that are needed in order to implement the algorithms efficiently. Our assumption is that these issues can be addressed while imposing only acceptable overhead on the algorithms.

Besides being a natural problem, the hope is that such an algorithm for this problem might be useful in finding fast algorithms for computing maximum cardinality matchings in bipartite graphs. The fastest algorithms known for this problem are by Hopcroft and Karp [HK71,HK73] (time $O(m\sqrt{n})$), by Ibarra and Moran [IM81] (time $O(n^{\omega})$)[3] and by Feder and Motwani [FM95] (time $O(m\sqrt{n}\log_n(n^2/m))$). Recently, Goel et. al. [GKK09] gave an $O(n \log n)$ algorithm to find a perfect matching in regular bipartite graphs. (The case of regular bipartite graphs is easier; $O(m)$ time algorithm was known earlier [COS01]). One approach to solve the matching problem is an interior point approach, which searches through fractional matchings, updating them in each step. Our problem can be thought of as an analog of updating along an augmenting path for fractional matchings. A vertex i in our problem corresponds to a vertex i on one side (say L) of the bipartite graph. The edge i in our problem corresponds to a vertex i' on the other side (say R). Each vertex $i' \in R$ is adjacent to the vertices i and $i+1 \in L$. h_i corresponds to the total amount of edges matched to i from vertices other than i' and $(i-1)' \in R$. w_i corresponds to 1 minus the total amount of edges matched to i' from vertices other than i and $i+1 \in L$.

The PB problem is also a special case of a *power-minimizing scheduling problem* that has been well studied [YDS95,LY05,LYY06]: suppose that there are n jobs to be scheduled on a single machine. Each job has an arrival time, a deadline, and needs some amount of CPU cycles. The machine can be run at different speeds, if it is run at speed s then it can supply s CPU cycles per unit time, and

[3] ω is the exponent in the matrix multiplication algorithm.

consumes a power of s^2 per unit time. (Again, it could be any convex function of s.) The goal is to schedule the jobs on the machine and determine the speeds so as to minimize the total power consumed. Li, Yao and Yao [LYY06] gave the fastest known algorithm for this problem that runs in time $O(n^2 \log n)$. Designing an $O(n \log n)$ time algorithm is an open problem. The PB problem is the following special case: there are n jobs with [arrival time, deadline] = $[i, i+1]$ which require h_i CPU cycles, for $i = 1$ to n. There are $n-1$ jobs with [arrival time, deadline] = $[i, i+2]$ which require w_i CPU cycles, for $i = 1$ to $n-1$.

The design of our algorithm is aided by physical intuition. We first design a hydraulic apparatus that may serve as an analog (rather than digital) computing device that solves the PB problem. Thereafter, we design an efficient algorithm that quickly simulates the operation of the hydraulic apparatus.

2 Preliminaries

Proposition 1 *The optimal solution is unique.*

The proof of the proposition, and all others in this section are in the appendix.

Proposition 2 *There is a linear time algorithm for checking if there is a perfectly balanced solution.*

When there is no perfectly balanced solution, we provide a structural characterization of the unique optimal solution.

Definition 1. *A solution is said to have a* block structure *(BS) if it can be partitioned into blocks in which each block is a consecutive set of vertices that have the same height, and every edge between two adjacent blocks allocates all its weight to the vertex of lower height (and hence is said to be* oriented *towards that node).*

Lemma 1. *For any PB problem, there is a unique solution with a block structure.*

Lemma 2. *A solution is optimal if and only if it has a block structure.*

It follows that to solve the PB problem it suffices to find a BS solution.

2.1 Hydraulic Apparatus

Our goal is to design more efficient algorithms for the PB problem. But before that, we describe a hydraulic apparatus that solves the PB problem (See Figure 1). The apparatus is constructed from a row of n identical bins arranged from left to right, where each bin has base area 1 square unit and height 4 units. Every two adjacent bins are connected by a horizontal cylindrical tube of base area 1 square unit and length one unit. Inside the tube there is a solid cylinder that exactly fits the width of the tube (no water can flow around it) and has width

$(1 - w_j)$ for tube e_j. (It would be desirable to have solid cylinders whose width can be varied so as to encode different instances of the PB problem, but the physical design of such cylinders is beyond the scope of this manuscript). The openings between the tube and each of the adjacent bins have smaller diameter than the tube, and hence the cylinder cannot extend out of the tube. There is a valve between every tube and the bin to the left of it.

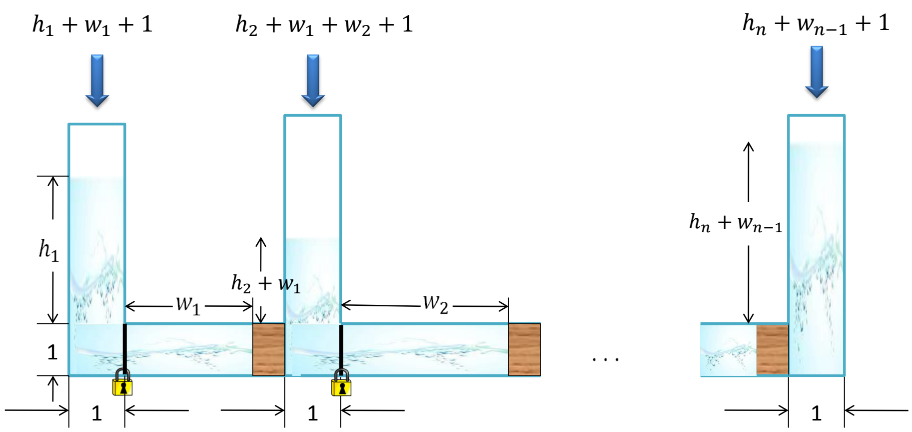

Fig. 1. Illustration of the Hydraulic Apparatus.

To input the initial conditions of the PB problem, first one shuts all valves. Then, iteratively for i from 1 up to $n-1$, one opens valve i, pours $(h_i+w_{i-1}+w_i+1)$ cube units of water into bin i (that now fill the tube to the right (ensuring this is the reason for the $+1$ term in the volume of water) pushing the cylinder all the way to the right), and closes valve i (closing valve i is not strictly necessary, but helps understand the algorithms that will follow). For bin n there is no valve to open, and one simply pours into it $(h_n + w_{n-1} + 1)$ cubic units of water. Observe that the initial condition corresponds to the case that the vertex to the right of an edge gets all the weight of the edge (the bin to the left of a tube also gets a volume of water corresponding to the weight of the corresponding edge, but this volume is spent on filling the tube). Now one opens all the valves. As a result, some of the cylinders may drift towards the left in their tubes (to an extent that depends on the relative water levels of adjacent bins). This corresponds to the situation where the corresponding edge allocates part (or all) of its weight to the left. The water levels when the system stabilizes (minus 1) are the solution to the PB problem.

Our algorithm is obtained by simulating (quickly) the action of the hydraulic apparatus. Our algorithm will be *monotone* in the sense that in the mathematical program, the variables x_i are initially all set to 0, and in every step of the algorithm can only be raised. (This corresponds to cylinders only drifting to the left and never to the right.)

Every edge will be in one of three states:

- *closed.* This corresponds to the situation when the valve of the corresponding tube is closed. All the edge weight has to be allocated to the right. Equivalently, the corresponding variable x_i is set to 0. At this point, the PB problem is broken into two independent subproblems, one to the left and one to the right of the edge.
- *open.* This corresponds to the situation when the valve is open. The weight of an edge may be distributed in an arbitrary way (including still allocating all the weight to the right). Once an edge is open, it is never closed again. Also, an edge is open unless it is blocked, which is the next state.
- *blocked.* This corresponds to the situation that all the weight of the edge is allocated to the left. Equivalently, the corresponding variable x_i is set to w_i. For the hydraulic apparatus, this means that the cylinder drifted all the way to the left of the tube. Since our algorithms will be monotone, once an edge is blocked it will never become unblocked again. Hence again, the PB problem is broken into two independent subproblems, one to the left and one to the right of the edge. However, since the edge will never reopen, the subproblems remain independent until the algorithm ends.

Having introduced the notion of closed edges, we extend the notion of block structure to that of *constrained block structure* (CBS). Here, some edges may be designated as being closed, and the PB problem is broken into independent subproblems separated by the closed edges, and one seeks BS solution for every subproblem. In particular, the initial state of the hydraulic apparatus corresponds to a CBS with all edges closed, and the final solution is a CBS with no edge closed. Given the set of closed edges, there is a unique corresponding CBS.

3 An $O(n \log n)$ Algorithm

Our algorithm for PB will go through a sequence of CBS's. Initially, all edges are closed. At every step one more edge is opened, and the corresponding CBS is found. Eventually, all edges become open and the final BS is found. We now focus on the opening of one edge.

Opening one edge at a time. When a valve is opened, the water in the hydraulic apparatus re-adjusts itself to get to a stable point. We refer to this process as one *round*. Suppose valve i is opened to begin a round. If at this point $h(v_i) \geq h(v_{i+1})$ then the system is already in a stable situation, and the old CBS is also the new CBS. Otherwise, cylinder i moves to the left until it comes to rest because either the heights of the vertices v_i and v_{i+1} become the same, or edge e_i becomes blocked. During this process we track the instantaneous block structure (IBS) of the system: this is the block structure defined by the instantaneous heights of the bins where consecutive bins with the same height belong to the same block. An IBS satisfies all the conditions of a CBS, except at the newly opened edge. As the cylinder i moves farther to the left, the IBS goes through a sequence of changes, and the IBS when the cylinder i comes to rest is the new CBS. We now identify the (only) two types of events that change the IBS.

Type 1 Events: Consider an edge e_j which has been opened prior to the round, but remained oriented to the right. That is, all its weight is allocated to the vertex v_{j+1} and x_j is set to 0. This is because prior to the round, $h(v_j) > h(v_{j+1})$. If at some point during the round the heights become the same, then cylinder j starts to move to the left, and the edge e_j is no longer being oriented. At this point the IBS changes: the blocks on either side of e_j merge to become a single block. For such an event we also say that an edge *starts to move.*

Type 2 Events: The other type of event is when an edge becomes blocked. Again the IBS changes: the block containing the edge is split into two blocks on either side of it.

Opening the rightmost edge We now consider a special case, suppose that we have the CBS where all edges except e_{n-1} are open. We then open edge e_{n-1} and find the new CBS (which will be the BS solution). We use an algorithm for this special case as a subroutine to design an algorithm for the PB problem. For now we prove the following theorem which guarantees a fast algorithm for this case.

Theorem 1 *Given the CBS solution with all edges but e_{n-1} open, the BS can be found in $O(n)$ time.*

First, we present two lemmas that describe how the IBS changes when we open the edge e_{n-1}. For the discussion that follows, we introduce a notion of time t. t is set to 0 when the round begins. We assume that the cylinder $n-1$ moves to the left at unit speed and calculate all other values as a function of t. We denote the speed at which cylinder i moves by $\frac{dx_i}{dt}$. We will also be interested in the height of the block containing vertex v_{n-1} and denote it by h. We denote the speed with which h increases by $\frac{dh}{dt}$.

Lemma 3. *Let the block containing v_{n-1} be $[j, n-1]$.*

1. $\frac{dh}{dt} = \frac{1}{n-j}$. *The time at which the edge e_{j-1} starts to move, if no other event happens earlier, is $(n-j)(h(v_{j-1}) - h(v_j))$.*
2. $\frac{dx_i}{dt} = \frac{i-j+1}{n-j}$. *The time at which edge e_i becomes blocked, if no other event happens earlier, is $(w_i - x_i)(n-j)/(i-j+1)$.*

Proof. 1. Water that flows into v_{n-1} at unit rate is distributed equally among all the $n-j$ vertices in the block $[j, n-1]$. Edge e_{j-1} starts to move when $\Delta h = h(v_{j-1}) - h(v_j)$, that is, when $\Delta t = (n-j)(h(v_{j-1}) - h(v_j))$.

2. There are $i - j + 1$ vertices in the block to the left of e_i, each of which accumulates water at rate $\frac{1}{n-j}$. The time at which edge i becomes blocked is precisely when $\Delta x_i = w_i - x_i$.

Lemma 4. *The events happen in the following order: Type 1 events happen from right to left (decreasing order), and after all such events, Type 2 events happen from left to right (increasing order).*

Proof. Let the current block containing v_{n-1} be $[j, n-1]$. Clearly, all the edges that started to move in this round lie in the current block, and the only edge that can start to move next is e_{j-1}. Also, if the last event was an edge blocking event, then it must have been the edge e_{j-1}. In this case any subsequent event does not effect the vertices in $[1, j-1]$. Therefore the only subsequent events that can happen are edges becoming blocked in $[j, n-1]$. (Or n joins the block $[j, n-1]$ and the system stabilizes.) Thus, if the events upto some time follow the given order, then the next event also follows the same order. The proof follows by induction on the sequence of events.

The algorithm computes the sequence of events that happen and other relevant information such as the heights of the blocks when these events happen, and then the eventual BS. The block structure is represented using an array. The ith element of the array contains information about the vertex i, whether it is the left end of a block, the right end (or both), or in the middle of the block. If it is the left end, then the position of the right end of the block is stored, and vice versa if it is the right end. The height of the block is stored at both the ends. Finally, for a vertex that is at an end of the block, we also store whether the adjacent edge is closed.

Given a CBS, we compute the solution (x_i values) that respects the CBS. It is easy to see that this can be done in $O(n)$ time.

Our algorithm proving Theorem 1 is composed of three procedures, where each procedure makes gradual progress towards the solution. Procedure 1 assumes a simplified version of the problem in which Type 2 events (blocking events) are assumed not to happen. Hence only Type 1 events (edges starting to move) happen, and the order of them happening is from right to left. The output of Procedure 1 is a tentative sequence $O_1, O_2, \ldots O_{n_1}$ of Type 1 events in the order in which they happen. For each event O_k, we store the edge j_k that started to move, the time t_k at which it happened, and the height $\hat{h}_k$ of the block at that time. We also store the total number of such events, n_1. Procedure 2 removes a suffix of the tentative sequence $O_1, O_2, \ldots O_{n_1}$, leaving a prefix that contains only those Type 1 events that actually do happen. To do this, one considers potential Type 2 events from left to right, and checks whether they would have prevented a Type 1 event to the left of them. If so, the respective Type 1 event is removed from the tentative sequence. Even though Procedure 2 considers potential Type 2 events, its only goal is to gather sufficient information about Type 2 events so as to be able to determine the correct sequence of Type 1 events. In particular, potential Type 2 events that are deemed irrelevant to this goal are not considered by Procedure 2. The task of determining the correct sequence of Type 2 events is left to Procedure 3, which is called only after the correct sequence of Type 1 events was determined.

Procedure 1: Find Type 1 events

- $k = 1$, $t_0 = 0$.
- $j =$ `the left end of the block whose right end is at` $n-1$.
- `While` $j > 1$ `and edge` $j-1$ `is not blocked, do`
  - $j_k = j - 1$ /* The next edge that opens is immediately to the left of j */
  - $t_k = t_{k-1} + (n-j)(h(v_{j-1}) - h(v_j)), \hat{h}_k = h(v_{j-1})$.
  - /* Move to the next block to the left */
  - $j =$ `the left end of the block whose right end is at` $j-1$.
  - $k = k + 1$.
- $n_1 = k - 1$.

Lemma 5. *Procedure 1 runs in $O(n)$ time.*

We now describe Procedure 2. Let $t = t_{n_1}$ be the time at which the last Type 1 event happens (according to the output of Procedure 1). We start with $i = j_{n_1} + 1$ and see if edge e_i becomes blocked before time t. If not, then we move to the edge to the right (by setting $i = i + 1$) and continue. If e_i does become blocked before t, we update t to be the time at which the previous Type 1 event happened (set $n_1 = n_1 - 1$, and $t = t_{n_1}$). If i is still to the right of the new j_{n_1} (since $j_k < j_{k-1}$ for all k), we continue with the same i, otherwise we set $i = j_{n_1} + 1$. We end when $i = n$.

One difficulty here is that we need to determine if e_i becomes blocked by time t in $O(1)$ steps. Let $\Delta x_i(t)$ be the distance traveled by cylinder i at time t (in the current round). Then e_i is blocked by time t iff $\Delta x_i(t) \geq w_i - x_i$. Note that $\Delta x_i(t) = \Delta x_{i-1}(t) +$ the increase in the height of v_i at time t. This increase in height is $(\hat{h}_{n_1} - h(v_i))$. So we can iteratively compute $\Delta x_i(t)$ in $O(1)$ steps.

This gives rise to another difficulty, if t changes then we might have to go back and recompute Δx_i starting from $i = j_{n_1}+1$. This might make the procedure run in quadratic time. We get around this by using the observation that $\Delta x_i(t_k) - \Delta x_i(t_{k-1}) =$ the distance traveled by cylinder i in the time interval $[t_{k-1}, t_k]$, which by Lemma 3 is equal to

$$\frac{(i - j_k + 1)(t_k - t_{k-1})}{n - j_k}.$$

Thus when we update t, we can also update $\Delta x_i(t)$ in $O(1)$ steps and continue with the same i.

Procedure 2: Eliminate Type 1 events

- $i = j_{n_1} + 1$, $\Delta x_i = \hat{h}_{n_1} - h(v_i)$.
- `While` $i < n$ `and` $n_1 > 0$ `do`

- If $\Delta x_i > w_i - x_i$ then /* Edge i would prevent edge j_{n_1} from opening */
  * $n_1 = n_1 - 1$.
  * If $i > j_{n_1}$ then /* i is still to the right of the new j_{n_1} */
    · $\Delta x_i = \Delta x_i - \frac{(i-j_{n_1}+1)(t_{n_1+1}-t_{n_1})}{n-j_{n_1}+1}$.
  * Else
    · $i = j_{n_1} + 1$, $\Delta x_i = \hat{h}_{n_1} - h(v_i)$.
- Else
  * $i = i + 1$.
  * $\Delta x_i = \Delta x_{i-1} + \hat{h}_{n_1} - h(v_i)$.

Lemma 6. *Procedure 2 runs in $O(n)$ time.*

We now describe Procedure 3 that computes the sequence of Type 2 events, and the times at which these happen. Note that the time at which an edge could potentially become blocked depends on all the events that happen prior to that, since each event changes the speed at which the cylinder moves. In particular, it depends on when any of the edges to the left become blocked. In addition, whether an edge ever becomes blocked depends on the Type 2 (blocking) events that happen to the right of the edge. Thus the dependencies go both ways and resolving these dependencies is a challenge. A naive algorithm that attempts to iteratively find the next event in the sequence takes $O(n^2)$ time, whereas our goal is to compute the entire sequence in $O(n)$ time. To do so we build the sequence of Type 2 events from left to right. We will borrow an approach that we used for constructing the sequence of Type 1 events, which was to first build the sequence under a simplifying assumption that certain blocking events do not happen, and then correct for the fact that they do happen. For Type 1 events, this construction took two stages, Procedure 1 and Procedure 2. For Type 2 events, we have only one stage, Procedure 3, but this procedure takes many rounds. Procedure 3 scans edges from left to right, and at every round it considers one more edge. It assumes that no blocking event happens to the right of the edge currently scanned. This implies that this edge (say edge e_i) necessarily eventually becomes blocked and is tentatively added to the sequence of blocking events. At this time we do another scan from right to left of the tentative sequence of the Type 2 events we have constructed so far to determine which ones can be removed because e_i is blocked earlier to them. In fact this is necessary to determine the exact time at which e_i becomes blocked. This nested loop hints at a quadratic running time, but we show that the time is indeed $O(n)$ based on the observation that once an event is removed from the sequence it is never returned.

First of all, before we proceed further, we update the x_i values upto time $\tau_0 = t_{n_1}$, the time of the last Type 1 event. Note that at this point all the heights that have changed are in $[j_{n_1}+1, n-1]$, and they are all equal to $\hat{h}_{n_1}$. It is easy to see that this update can be done in $O(n)$ time.

Our algorithm builds the following iteratively, starting with the left most edge and moving right: the sequence of Type 2 events, ending at e_i becoming blocked, assuming no events happen to the right of e_i. The sequence of events, say $E_1, E_2, \ldots, E_{n_2}$ in the order in which they happen, is maintained as an array. For each event E_k in the sequence, we store the corresponding edge that was blocked, say i_k, the time at which the event happened, say τ_k, and the increase in the height of the block when that event happened, say Δh_k. The total number of such events n_2 is also maintained. Also for the sake of convenience, set i_0 to be the edge at the left end of the block containing j_{n_1}, that is $[i_0+1, j_{n_1}]$ is a block. The time τ_0 as mentioned earlier is set to t_{n_1}. Δh_0 is set to 0.

At the beginning of the i^{th} iteration, we have the sequence of events upto $i-1$, that is the last event is e_{i-1} becoming blocked. In the i^{th} iteration, we check if e_i becomes blocked before e_{i-1}. If not then we insert e_i after e_{i-1} and proceed to the next iteration. Otherwise, we iteratively consider the previous event in the sequence and do the same, until we either find an event that happens before e_i is blocked, or we eliminate all events in the sequence in which case e_i will be the only event in the new sequence.

We now show how to determine whether e_i becomes blocked before E_k or not. As before, let $\Delta x_i(\tau_k)$ be the distance moved by cylinder i at time τ_k. Let $j = i_k$ be the edge that was blocked during event E_k. Note that

$$\Delta x_i(\tau_k) = \Delta x_j(\tau_k) + (i-j)\Delta h_k.$$

This is because, the distance moved by cylinder i is equal to the distance moved by cylinder j plus the water that accumulated at the vertices in the range $[j+1, i]$. Further we know that $\Delta x_j(\tau_k) = w_j - x_j$ since e_j became blocked at τ_k. The exception is when $k = 0$ in which case $\Delta x_i(\tau_k) = 0$. Thus we can determine if $\Delta x_i(\tau_k) \geq w_i - x_i$, which gives us the required answer.

Finally, once we have determined the right position k, we update E_{k+1} to be the event that e_i becomes blocked. We set $i_{k+1} = i$ and let $j = i_k$. The time τ_{k+1} is given by

$$w_i - x_i = \Delta x_i(\tau_{k+1}) = \Delta x_i(\tau_k) + \frac{i-j}{n-j}(\tau_{k+1} - \tau_k),$$

from which one gets

$$\tau_{k+1} = (w_i - x_i - \Delta x_i(\tau_k))\,\frac{n-j}{i-j} + \tau_k.$$

$\Delta h_{k+1} = \Delta h_k + \frac{1}{n-j}(\tau_{k+1} - \tau_k)$. Also n_2 is set to $k+1$.

Procedure 3: Find Type 2 events

- $n_2 = 0$.
- `For` $i = i_0 + 1$ `to` $n - 1$ `do` /* When is edge e_i blocked? */
  - $k = n_2$, $j = i_k$.
  - `If` $k \neq 0$, `then` $\Delta x_i = w_j - x_j + (i-j)\Delta h_k$, `Else` $\Delta x_i = 0$.
  - `While` $k > 0$ `and` $\Delta x_i > w_i - x_i$, /* e_i is blocked before E_k */
    - $k = k - 1$, $j = i_k$.

* `If` $k \neq 0$, `then` $\Delta x_i = w_j - x_j + (i-j)\Delta h_k$, `Else` $\Delta x_i = 0$.

- /* Insert e_i being blocked as the event E_{k+1} */
- $i_{k+1} = i$.
- $\tau_{k+1} = (w_i - x_i - \Delta x_i)\,\frac{n-j}{i-j} + \tau_k$.
- $\Delta h_{k+1} = \Delta h_k + \frac{1}{n-j}(\tau_{k+1} - \tau_k)$.
- $n_2 = k+1$.

Lemma 7. *Procedure 3 runs in time $O(n)$ time.*

Proof. Naively, each time the inner (While) loop for Procedure 3 might go from n_2 to 0 and this would give an n^2 bound. However, note that each iteration of the inner While loop eliminates an edge blocking event, and every such event can be eliminated only once. Thus there can be only $O(n)$ iterations of the inner loop overall and hence the running time of this procedure is $O(n)$.

Procedure 3 finds all the Type 2 events upto edge $n-1$, assuming that nothing happens to the right of $n-1$. That is, Procedure 3 ignores the possibility that the heights of v_{n-1} and v_n might become the same and the round ends due to that. Therefore we next compute at what time the heights become equal and determine if some events need to be eliminated because of that. The height of v_n at time t is simply $h(v_n) - t$. The height of v_{n-1} however depends on the sequence of events that happen. Recall that for the Type 1 events, we actually stored the height of v_{n-1} at each t_k, which was $\hat{h}_k$. So for every k from 1 to n_1, we can compare the heights of v_{n-1} and v_n and see if they cross over, that is at time t_k, the height of v_{n-1} is smaller than that of v_n but at time t_{k+1} it is larger. In that case, we set $n_1 = k$, and $n_2 = 0$. If the heights never cross over during Type 1 events, we then move on to Type 2 events. Once again, we stored the *height increment* of v_{n-1} at each τ_k, which was Δh_k. Therefore as before we can compare the heights of the two vertices at time τ_k for every k from 1 to n_2 and see if they cross over. If they do cross over at k, then we set $n_2 = k$.

Finally, once we have determined the entire sequence of events in a round, we can update the block structure and the new x_i values. Suppose first that the round ended with e_{n-1} becoming blocked. In this case the new blocks are $[i_0+1, i_1], [i_1+1, i_2], \ldots, [i_{n_2-1}+1, n-1]$. Everything to the left of i_0, that is everything in the range $[1, i_0]$ remains unchanged. We also know the heights of each of these blocks, so finding the new x_i values is easy. In case the round ended with the heights of v_{n-1} and v_n becoming equal, then everything is as before, except that the last block is $[i_{n_2}+1, n]$. It is easy to see that everything after Procedure 3 can be done in $O(n)$ time. This completes the proof of Theorem 1.

Divide and Conquer We now show how the technology developed for the special case of opening the valve for the rightmost edge can be used to solve the PB problem. First, we show that essentially the same algorithm can be used to solve the case when it is the middle edge whose valve is closed to begin with. Then we show how to use this case to apply a divide and conquer technique to solve the entire problem.

Lemma 8. *Given the CBS solution with all edges but $e_{n/2}$ open, the BS solution can be found in $O(n)$ time.*

The divide and conquer strategy we follow is the most natural one: recursively, each half can be solved separately by keeping the middle valve closed. We then combine them by opening the middle valve.

Theorem 2 *The BS can be found in $O(n \log n)$ time.*

4 Conclusion and Open Problems

We gave an $O(n \log n)$ algorithm for a natural load balancing problem on paths. The same problem can be generalized to trees, and trees in hypergraphs. Extending our techniques to handle these cases is an interesting open problem. Our problem is also a special case of a power-minimizing scheduling problem for which the best known algorithm runs in time $O(n^2 \log n)$. A challenging open problem is if our algorithm can be extended to solve this problem. Also, the original motivation for our problem was that it could be useful in obtaining a faster algorithm for bipartite matching. Improving the running time for this problem is a long-standing open problem.

Acknowledgements

We thank Nikhil Bansal for directing us to relevant references.

References

[COS01] Richard Cole, Kirstin Ost, and Stefan Schirra. Edge-coloring bipartite multigraphs in $O(E \log D)$ time. *Combinatorica*, 21(1):5–12, 2001.

[FM95] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Syst. Sci.*, 51(2):261–272, 1995.

[GKK09] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in $O(n \log n)$ time in regular bipartite graphs. *CoRR*, abs/0909.3346, 2009. Also to Appear in STOC 2010.

[HK71] John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. In *FOCS*, pages 122–125, 1971.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[IM81] Oscar H. Ibarra and Shlomo Moran. Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication. *Inf. Process. Lett.*, 13(1):12–15, 1981.

[LY05] Minming Li and Frances F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. In *MFCS*, pages 652–663, 2005.

[LYY06] Minming Li, Andrew C. Yao, and Frances F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. *Proceedings of the National Academy of Sciences of the USA*, 103:3983–3987, 2006.

[YDS95] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In *FOCS*, pages 374–382, 1995.

A Missing Proofs

Proof. Of Proposition 1. At least one optimal solution exists, since the feasible region is compact and the objective function is real and continuous. Assume for the sake of contradiction that there are two optimal solutions. Then their average is also a feasible solution of strictly lower value (by strict convexity of sum of squares).

Proof. Of Proposition 2. In a perfectly balanced solution, the height of every node is exactly $h = (\sum_{i=1}^{n} h_i + \sum_{j=1}^{n-1} w_j)/n$. Compute h (in linear time). Now scan the path edge by edge starting at e_1, and set $x_j = h - h_j - (w_{j-1} - x_{j-1})$ (and fail if this value is negative or larger than w_j).

Proof. of Lemma 1. The fact that at least one BS solution exists follows from Lemma 2 (which states that the optimal solution has block structure). We show that there cannot be two BS solutions. Assume for the sake of contradiction that there are two BS solutions S1 and S2. Let $H_1(i) = \sum_{j=1}^{i} h(v_j)$ be the sum of heights of the first i vertices in S1. Similarly define $H_2(i)$. It is easy to see that if edge e_i is oriented to the left in S1, then $H_1(i) \geq H_2(i)$. Similarly, if e_i is oriented to the right in S1, then $H_1(i) \leq H_2(i)$. Suppose for some i, $H_1(i) > H_2(i)$, we show that this implies that $H_1(i+1) > H_2(i+1)$. Inductively, this leads to a contradiction and implies that the heights in the two solutions (and hence the block structure) are the same.

If $H_1(i) > H_2(i)$ then the edge e_i cannot be oriented to the right in S1. Therefore $h(v_{i+1}) \geq h(v_i)$ in S1. Similarly, $H_1(i) > H_2(i)$ implies that e_i is not oriented to the left in S2. Therefore $h(v_{i+1}) \leq h(v_i)$ in S2. Thus $H_1(i+1) > H_2(i+1)$.

Proof. Of Lemma 2. Assume that a solution does not have a block structure. Then there are two adjacent vertices (say v_i and v_{i+1}) of unequal height (say $h(v_i) > h(v_{i+1})$) with the edge between them not oriented away from the higher node. Hence (in our example), $x_i > 0$. Lowering x_i by some small $0 < \epsilon \leq \min[x_i, (h(v_i) - h(v_{i+1}))/2]$ will lower the value of $(h(v_i))^2 + (h(v_{i+1}))^2$ and hence will improve the objective function. This implies that the optimal solution must have block structure. The uniqueness of the optimal solution and the uniqueness of BS solutions now implies that every BS solution is optimal.

Proof. Of Lemma 8. When we open $e_{n/2}$, we compute the sequence of events to the left of it exactly as before, independent of what happens to the right, ending at the event of $e_{n/2}$ becoming blocked. Symmetrically, we can do the same thing to the right of $e_{n/2}$ as well: as the cylinder $n/2$ moves to the left, Type 1 events happen from left to right, and then Type 2 events happen from right to left. The entire sequence can be computed exactly as in the other case. The two sides are independent because the events are driven by the movement of cylinder $n/2$, which separates the two sides. In the final step, we check if the heights of $v_{n/2}$ and $v_{n/2+1}$ cross over during this process. Now both the heights depend on the sequence of events on the corresponding sides. However, since we store the heights at the time of occurrence of each event, it is easy to see that this can be done in $O(n)$ time. It is also easy to see that the new block structure can be computed quickly.

Proof. Of Theorem 2. In the first round, open odd numbered edges, 1, 3, 5, etc. In the second round open edges with numbers 2 modulo 4. Then 4 modulo 8 and so on. Each round takes $O(n)$ time, and there are $\log n$ rounds.

B Dynamic Programming algorithm

Theorem 3 *The PB problem can be solved in time $O(n^3)$.*

Proof. We prepare a dynamic programming table T with $n-1$ columns (one for each edge), two rows, and 0/1 entries. Entry $T(1,i) = 1$ iff it is *plausible* that there is a BS solution in which edge e_j is oriented to the left, and entry $T(2,i) = 1$ iff it is plausible that there is a BS solution in which edge e_j is oriented to the right. Here the notion of plausibility does not mean that there actually is a BS solution with this edge orientation, but only that the algorithm has not ruled out such a possibility.

The entries of the table are filled out from the lowest index column to the largest. Having filled out columns 1 up to $i-1$, column i is filled as follows. $T(1,i)$ can be set to 1 if by orienting e_i to the left, v_i can become the righthand side vertex of a block. We need to determine whether there is a plausible lefthand side vertex to the block. Such a vertex may either be any vertex immediately following a previously oriented edge (there are $2(i-1)$ such possibilities to check, given the two possible orientations) or v_1. Given a candidate starting vertex v_j for the block, together with the orientation of the edge preceding it, there is a linear time algorithm (linear in $i-j+1$) to check whether the segment $[i,j]$ can be made into a block (similar to Proposition 2). If indeed $[i,j]$ can be made into a block, one needs to check that the orientation of edge e_{j-1} is consistent (goes from a block of greater height to a block of lower height), and only then $T(1,i)$ can be set to 1. At worse, computing entry $T(1,i)$ takes $O(i^2)$ time, and similarly for entry $T(2,i)$. When the table is full, it remains to check (as above) whether v_n can be part of a block that begins after any one of the plausible edges. The total running time is $O(n^3)$.

C The iterated greedy algorithm

Start from the left and greedily maximize number of vertices at average height. Consider only non-average vertices. Between any two vertices of different signs (one below and one above), there must be an oriented edge immediately following the first of the two. Cut at these oriented edges, and repeat on the subparts that remain. Running time is at most $O(n^2)$ because an iteration takes linear time and either returns a balanced solution or orients at least one edge.

A lower bound for the iterated greedy algorithm.

Let M be a sufficiently large (in particular, larger than $n!$). Let A (for average) be such that $A = (\sum_{i=1}^{n} i!)/n$. For $1 \leq j \leq n-2$, let the weight of the edges be $w_j = \sum_{i=j+1}(A - i!)$, and let $w_{n-1} = 0$. For $1 \leq j \leq n-1$, let the initial vertex heights be $h_j = j! + M - w_j$ and let $h_n = n! + M$.

In the optimal solution, all edges are oriented left, and the heights are $h(i) = i! + M$. Observe that the average height is $M + A$, and only v_n exceeds the average height.

If the alternative algorithm is run from left to right then in fact the first pass already gives the optimal solution. However, the algorithm does not detect this, and instead cuts off the last vertex. In further passes, it will continue to find the optimal solution, but nevertheless just cut off the last remaining vertex (orienting the edge preceding it to the left), because all vertices before it are below the target average (of the respective iteration). Hence the total running time is $\Omega(n^2)$.

If the alternative algorithm is run from right to left, v_n is above average, edges are oriented to the right, and the only vertex to be below average is v_1. So again v_n is cut off. In subsequent passes, it will always be the case that only the rightmost vertex is above average, and only v_1 is below average. Hence again, linear time will be spent to cut off a single vertex, giving total running time of $\Omega(n^2)$.

A variation on the lower bound is to add a leftmost vertex v_0 of initial height $h_0 = M$, and give the edge $(0, 1)$ some small weight such as $w_0 = 3$. In the optimal solution, the heights of both v_0 and v_1 are then $M + 2$, but all (but last) passes of the iterated greedy algorithm that start from v_0 will give v_0 a height of $M + 3$. Hence we will no longer be in the situation that the algorithm actually finds the optimal solution quickly (without being aware of this).