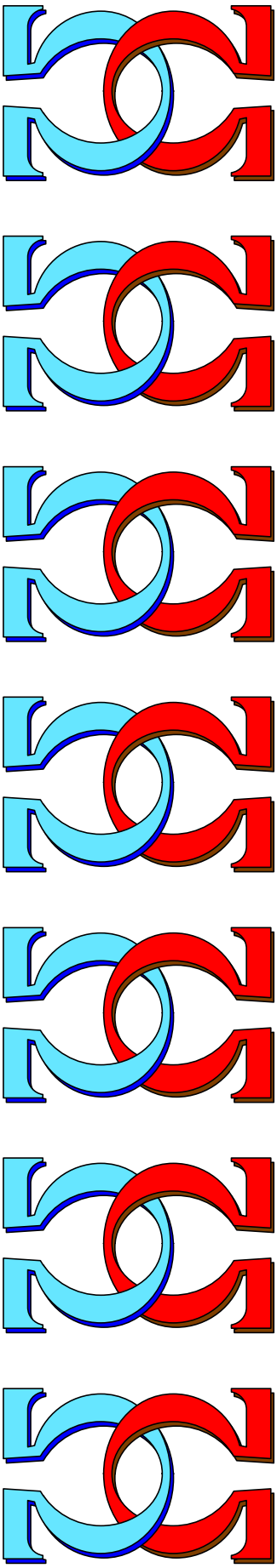


CDMTCS
Research
Report
Series

Balance Machines: A New Formalism for Computing

Joshua J. Arulanandham
Michael J. Dinneen

Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-256
December 2004

Centre for Discrete Mathematics and
Theoretical Computer Science

Balance Machines: A New Formalism for Computing

Joshua J. Arulanandham and Michael J. Dinneen
Department of Computer Science
The University of Auckland
Auckland, New Zealand
{joshua,mjd}@cs.auckland.ac.nz

December 1, 2004

Abstract

The Balance Machine is a newly proposed natural computational model that consists of components resembling physical balances. The model has only one operation, "balancing", which suffices in principle to perform universal computation. An interesting feature of the balance machine is its bidirectional operation: it can compute "forwards and backwards", i.e. both a function and its partial inverse can be computed spontaneously using the same machine. Also, the machine exhibits a different kind of parallelism—a "bilateral parallelism".

The aim of this note is two–fold: To introduce a formalism for computing with balance machines—a convenient notation for representing the mechanical model on paper, and to demonstrate its expressive power by solving two NP–complete problems, namely, Set Partition and Knapsack.

1 Computing without knowing how to count

Standard theoretical formalisms and programming languages of mainstream computer science influence us to think of computation mainly in terms of arithmetic/logical operations and symbol processing. We introduce a new computing formalism which is not based on arithmetic/logic or symbol processing, but which can, in principle, serve the purpose of its classical, better–known counterparts. Our formalism pertains to computing with a Balance Machine, an unconventional mechanical model of computation first proposed in [1].

The idea of computing using a balance might seem bizarre. It can be explained with the following example which is hypothetical and rather crude, yet serves our purpose. Suppose there is an illiterate farmer who has no knowledge of numbers/numerals

and counting, but nevertheless wishes to "count" if all of his sheep return safely after going out for grazing. How would he manage to do it without even knowing how to count 1, 2, 3, ...? (Well, he could use pebbles to keep count of his sheep, but that would not illustrate our idea of a balance!) He installs a gigantic physical balance in his backyard. Before sending the sheep out for grazing he makes all his sheep stand on the left side of the balance and puts a huge barrel on its right. By trial and error he finds out to what extent the barrel needs to be filled, say, with a liquid, to balance the sheep's weight. After the sheep return, all he does is to make the sheep stand on the left side of the balance again, place the same barrel on its right and see if they balance or not[1]. The point is that the actual numerical *value* of the weight of sheep (or their count given by a number) is not important for this computation; it is not necessary that the farmer should scribble the value (a number) on a piece of paper or memorize it. In fact, the illiterate farmer's approach to compute using weights of possibly *unknown* values—that is, we do not know the values of the numbers representing them, but only know of "something that balances this"—can be put to use, literally in every conceivable computing situation! For example, as shown in Figures 2 and 6, the process of adding two quantities x and y can be reduced to finding "something that balances x and y weights (placed on the left pan)".

Typically, a balance machine "computer" consists of components that resemble ordinary physical balances (see Figure 1), each with a natural tendency to spontaneously balance their left and right pans: If we start with certain fixed weights, representing *inputs*, on some[2] of the pans, then the balance–like components would vigorously try to balance themselves by filling the rest of the pans with suitable (liquid) weights representing the *outputs*. Roughly speaking, the proposed machine has a natural ability to load itself with output weights that "balance" the input. This balancing act can be viewed as a computation. There is just one rule that drives the whole computing process: *the weights on the left and right pans of the individual balances should be made equal.* Note that the machine is designed in such a way that the balancing act would happen automatically by virtue of physical laws: i.e., the machine is self-regulating.[3] In [1] we have shown that all computations can be ultimately expressed using one primitive operation: *balancing*; this sort of intuition suffices to conceptualize/implement *any* computation performed by a conventional computer. Armed with the *computing = balancing* intuition, we can see basic computing operations in a different light.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to the proposed computational model; Section 3 introduces symbolic representations of balance machines and uses these to solve a variety of computing problems; Section 4 discusses the notion of *bilateral computing*; Sections 5 and 6 demonstrates the power

[1]The farmer assumes that the sheep would not have gained considerable weight in a single day, after grazing!

[2]The machine is not necessarily a single balance; it can be a combination of two or more balances.

[3]If the machine cannot eventually balance itself, it means that the particular instance does not have a solution.

Figure 1: A physical balance.

of the new computing formalism by solving two NP–complete problems, namely, Set Partition and Knapsack respectively; Section 7 concludes the paper.

2 The Balance Machine model

At the core of the proposed natural computational model are components that resemble a physical balance. In ancient times, the shopkeeper at a grocery store would place a standard weight in the left pan and would try to load the right pan with a commodity whose weight equals that on the left pan, typically through repeated attempts. The physical balance of our model, though, has an intrinsic self-regulating mechanism: it can automatically load (without human intervention) the right pan with an object whose weight equals the one on the left pan. See Figure 2 for a possible implementation of the self-regulating mechanism.

In general, unlike the one in Figure 2, a balance machine may have more than just two pans. There are two types of pans: pans carrying *fixed* weights which remain unaltered by the computation, and pans with *variable* liquid weights that are changed by activating the filler–spiller outlets. Some of the fixed weights represent the inputs, and some of the variable ones represent outputs. The following steps constitute a typical computation by a given balance machine:

> (i) Plug in the desired inputs by loading weights to pans. Pans with variable weights can be left empty or assigned with arbitrary weights. This defines the initial configuration of the machine.
>
> (ii) Allow the machine to balance itself: the machine automatically adjusts the variable weights till left and right pans of the individual balance(s) become equal.
>
> (iii) The weights of liquid collected in the output pans denote the "output" of the computation.

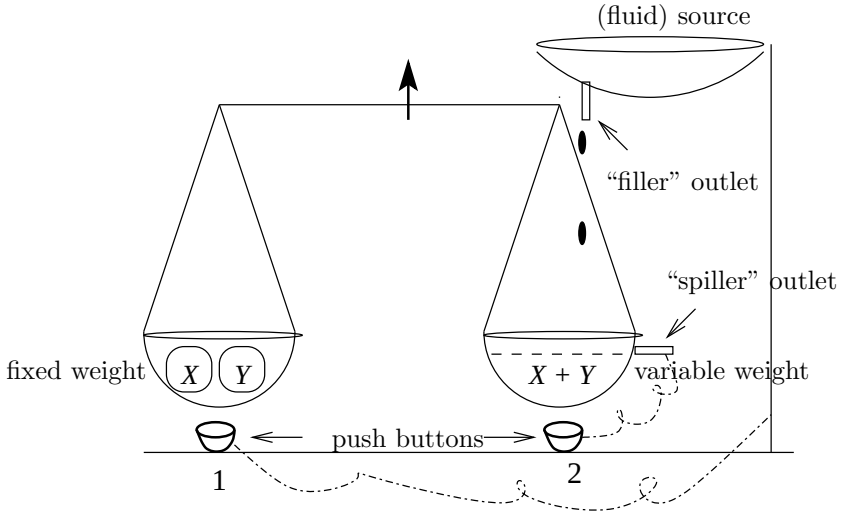


Figure 2: A self-regulating balance. The *source* is assumed to have an arbitrary amount of a liquid–like substance. When activated, the *filler* outlet lets liquid from the source into the right pan; the *spiller* outlet, on being activated, allows the right pan to spill some of its contents. The balancing is achieved by the following mechanism: the spiller is activated if at some point the right pan becomes heavier than the left (i.e., when push button (2) is pressed) to spill off the extra liquid; similarly, the filler is activated to add extra liquid to the right pan just when the left pan becomes heavier than the right (i.e., when push button (1) is pressed). Thus, the balance machine can "add" (sum up) inputs x and y by balancing them with a suitable weight on its right: after being loaded with inputs, the pans would go up and down till they eventually find themselves balanced.

3 Symbolic representations of balance machines

It might be convenient if we develop some kind of a pictorial or textual representation of balance machines, rather than having to conjure an image of real physical balances like the one shown in Figure 2 while computing with them. In this Section we develop symbolic representations of balance machines and use them to solve a variety of computing problems.

See Figure 3 for one pictorial, textual representation of a machine that adds two quantities. In the textual representation, the keyword *balance* represents a balance with left and right pans. The variable names to the left and the right of the comma represent weights on the left and right pans respectively. Those beginning with small letters represent fixed weights and the ones starting with capital letters, variable weights. Note that there can be more than one variable on each side of the comma, connected together by the '+' sign which simply denotes a "grouping" of weights that are attached to the same side of the balance. More details regarding syntax will be given shortly by way of examples and, the syntax is formally described at the end of this section using context–free grammar rules.

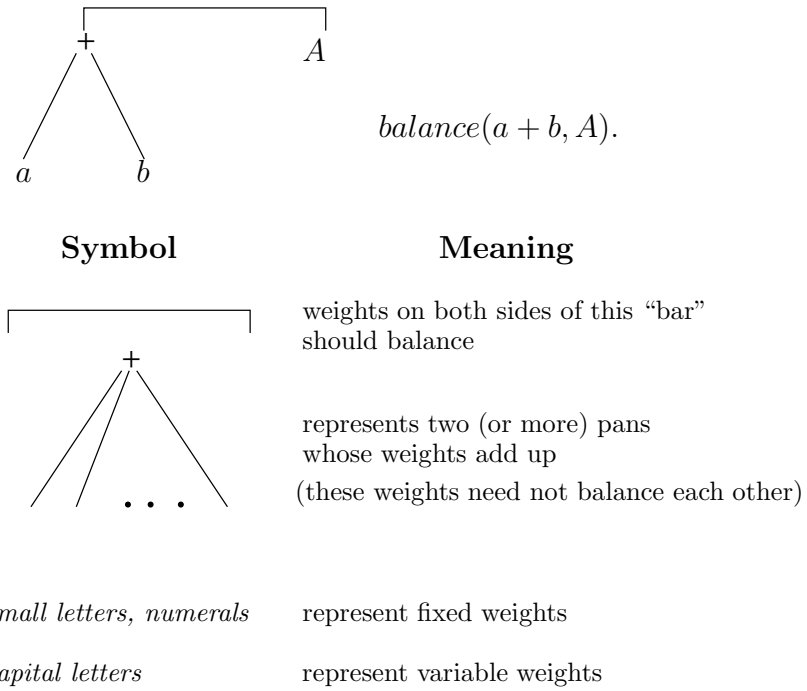


Figure 3: Pictorial, textual representations of a simple balance machine that performs addition.

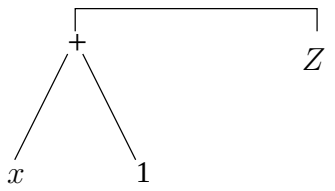
In what follows, we give examples of a variety of balance machines that carry out a wide range of computing tasks—from the most basic arithmetic operations to solving linear simultaneous equations. Balance machines that perform the operations *increment*, *decrement*, *addition*, and *subtraction* are shown in Figures 4, 5, 6, and 7, respectively. Legends accompanying the figures detail how they work.

Balance machines that perform *multiplication by* 2 and *division by* 2 are shown in Figures 8, 9, respectively. Note that in these machines, one of the weights/pans takes the form of a balance machine[4] which demonstrates a kind of recursion.

We now introduce another technique of constructing a balance machine: having a common weight shared by more than one machine. Another way of visualizing this situation is to think of pans belonging to two different machines being placed one over the other. We use this idea to solve a simple instance of linear simultaneous equations. See Figures 10 and 11 which are self–explanatory. In the textual representation a semicolon has been used to separate the individual balances; weights/pans that are shared among the individual machines bear the same variable names.

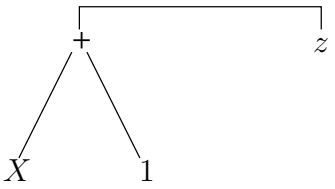
Having shown how to construct balance machines for a few computing problems, we wish to state that we can in fact construct, using balance machines, all primitive

[4]The weight contributed by a balance machine is assumed to be simply the sum of the individual weights on each of its pans. The weight of the bar and the other parts is not taken into account for the sake of simplicity.



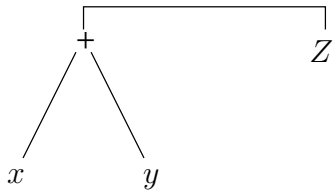
$balance(x + 1, Z)$.

Figure 4: Increment operation. Here x represents the input and Z represents the output. The machine computes $increment(x)$. Both x and '1' are fixed weights clinging to the left side of the balance machine. The machine eventually loads into Z a suitable weight that would balance the combined weight of x and '1'. Thus, eventually $Z = x + 1$, i.e., Z represents $increment(x)$.



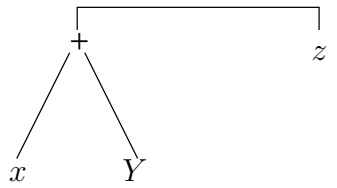
$balance(X + 1, z)$.

Figure 5: Decrement operation. Here z represents the input and X represents the output. The machine computes $decrement(z)$. The machine eventually loads into X a suitable weight so that the combined weight of X and '1' would balance z. Thus, eventually $X + 1 = z$, i.e., X represents $decrement(z)$.



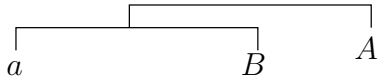
$balance(x + y, Z)$.

Figure 6: Addition operation. The inputs are x and y and Z represents the output. The machine computes $x + y$. The machine loads into Z a suitable weight, so that the combined weight of x and y would balance Z. Thus, eventually $x + y = Z$, i.e., Z would represent $x + y$.



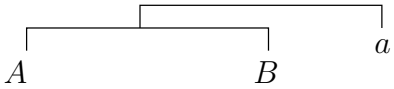
$balance(x + Y, z)$.

Figure 7: Subtraction operation. Here z and x represent the inputs and Y represents the output. The machine computes $z - x$. The machine loads into Y a suitable weight so that the combined weight of x and Y would balance z. Thus, eventually $x + Y = z$, i.e., Y would represent $z - x$.



$balance(balance(a, B), A)$.

Figure 8: Multiplication by 2. Here a represents the input and A represents the output. The machine computes $2a$. The combined weights of a and B should balance A: $a + B = A$; also, the individual weights a and B should balance each other: $a = B$. Therefore, eventually A will assume the weight $2a$.



$balance(balance(A, B), a)$.

Figure 9: Division by 2. The input is a and let A represent the output. The machine "exactly" computes $a/2$. The combined weights of A and B should balance a so that $A + B = a$; also, the individual weights A and B should balance each other: $A = B$. Therefore, eventually A will assume the weight $a/2$.

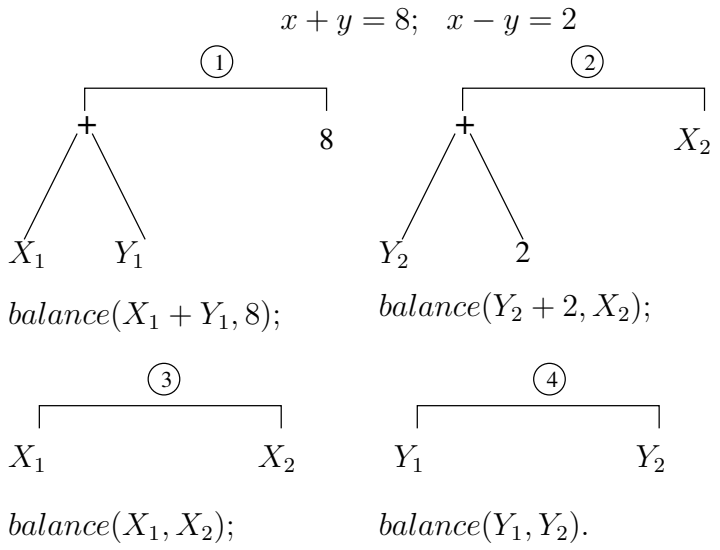


Figure 10: Solving simultaneous linear equations. Balance machines (1) and (2) realize the equations in a straightforward manner. Note that both X_1 and X_2 represent the same variable x, and therefore must be made equal; this also applies to Y_1 and Y_2. The constraints $X_1 = X_2$ and $Y_1 = Y_2$ will be enforced by balance machines (3) and (4). Observe the sharing of pans between them. The individual machines work together as a single balance machine.

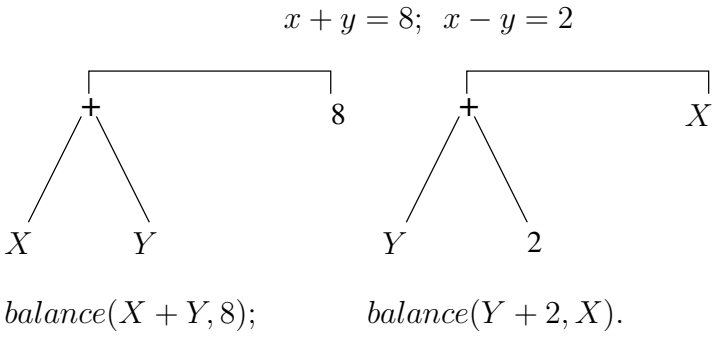


Figure 11: Solving simultaneous linear equations (easier representation). This is a simpler representation of the balance machine shown in Figure 10. Machines (3) and (4) are not shown; instead, we have used the same (shared) variables for machines (1) and (2).

hardware components that serve as the building blocks of a general purpose universal digital computer: logic gates, memory cells (flip-flops), and transmission lines [1].

A formal description of the syntax of the textual notation used for representing balance machines is given below, using context–free grammar rules:

$$
\begin{array}{lcl}
\mathit{Statement} & \rightarrow & \mathit{Balances}. \\
\mathit{Balances} & \rightarrow & \mathit{Balance}; \mathit{Balances} \mid \mathit{Balance} \\
\mathit{Balance} & \rightarrow & \textbf{balance}(\mathit{Weight}, \mathit{Weight}) \\
\mathit{Weight} & \rightarrow & \mathit{Weight} + \mathit{Weight} \mid \mathit{NonnegativeReal} \mid \mathit{Variable} \mid \mathit{Balance}
\end{array}
$$

4 Bilateral computing

An important property of balance machines is that they are *bilateral computing* devices. See [2] where the notion of bilateral computing was first proposed. Typically bilateral computing devices can compute both a function and its (partial) inverse, using the same mechanism. For instance, the same underlying balancing mechanism is used in both increment and decrement operations (see Figures 4 and 5), except for the fact that we change fixed weights to variable ones and vice versa. Also, compare machines that (i) add and subtract (see Figures 6 and 7) and (ii) multiply and divide by 2 (see Figures 8 and 9).

There is a fundamental asymmetry in the way we normally compute: while we are able to design circuits that can *multiply* quickly, we have relatively limited success in *factoring* numbers; we have fast digital circuits that can "combine" digital data using AND/OR operations and realize Boolean expressions, yet no fast circuits that can determine the truth value assignment satisfying a Boolean expression. Why should computing be easy when done in one "direction", and not so when done in the other "direction"? In other words, why should *inverting* certain functions be hard, while *computing* them is quite easy? It may be because our computations have been based on rudimentary operations like addition, multiplication, etc. that force an explicit distinction between "combining" and "scrambling" data, i.e. *computing* and *inverting* a given function. On the other hand, a primitive operation like *balancing* does not do so. It is the same balance machine that does both addition and subtraction: all it has to do is to somehow balance the system by filling up the empty variable pan representing output; whether the empty pan is on the right (addition) or the left (subtraction) of the balance does not particularly concern the balance machine! In the bilateral scheme of computing, there is no need to develop two distinct intuitions—one for addition and another for subtraction; there is no dichotomy between functions and their (partial) inverses.

We can mathematically characterize a bilateral computing system as follows: Consider a surjective (not necessarily bijective) function $f : X \rightarrow Y$ and the set of functions $G = \{g : Y \rightarrow X \mid f(g(y)) = y, \forall y \in Y\}$. A bilateral computing system is one which can implement f as well as some $g \in G$, using the same intrinsic "mechanism" or "structure".

In the sections that follow we show how two classic NP–complete problems can be solved under a bilateral computing scheme, using balance machines.

5 The Set Partition problem

Set Partition, a well–known hard computing problem which is NP–complete [3] can be stated as follows:
Input: A set S of positive numbers.

Question: Can S be "split" into two disjoint subsets S_1, S_2 such that $S_1 \cup S_2 = S$ and $sum(S_1) = sum(S_2)$, where $sum(S_1)$ and $sum(S_2)$ represent the sum of the elements in S_1 and that of the elements in S_2, respectively?

In what follows we will try to *rephrase* the problem in such a way that one could use the "Balance Machine vocabulary" to address it.

Consider a set $S = \{x_1, x_2, \ldots x_n\}$ of cardinality n. Let us suppose, we *can* partition it into disjoint sets S_1 and S_2 as required. The existence of a partition can be pictured as follows: imagine two sets of "pockets"—n pockets on the left side and n on the right side of a balance, say, which are meant to hold the elements of S; the elements from S can be seen as being distributed/scattered across the pockets—some are held by left side pockets, and the rest on the right (of course, some pockets will be left empty). The pockets on the left, taken together, and those on the right weigh the same. Note that every element in S will be found in *one* of the pockets, *either* in one of the left ones *or* in those on the right, never on both sides. To make things simple and orderly, we can safely assume that the *first* element in S will be found either in the *first* left side pocket or in the *first* right side pocket (i.e., one of them will always be empty), the *second* element in S will be found either in the *second* left pocket or in the *second* right pocket, and so on. This scheme of picturing the partition can be formally expressed as follows.

Let variables l_1, l_2, …, l_n and r_1, r_2, …, r_n represent the left and the right pockets respectively. Then, every element $x_i \in S$ will be held by either l_i or r_i, i.e. *one* of the following conditions will be true:
A) $l_i = x_i$ and $r_i = 0$.
B) $l_i = 0$ and $r_i = x_i$.
The following condition will also be true since we *can* partition S:
C) $l_1 + l_2 + \ldots + l_n = r_1 + r_2 + \ldots + r_n$.

Conversely, consider a set S of cardinality n (whether it is partitionable is not known yet): If one can generate n left and n right pockets for which there exists an assignment, obeying condition (A) or (B), such that $l_1 + l_2 + \ldots + l_n = r_1 + r_2 + \ldots + r_n$, then there should exist a way to partition S—into disjoint sets S_1 and S_2. This is because $l_1 + l_2 + \ldots + l_n$ can be seen as constituting $sum(S_1)$ and $r_1 + r_2 + \ldots + r_n$ as $sum(S_2)$.

Thus, the problem of deciding if a given set can be partitioned or not, can be recast as the problem of generating required number of left and right pockets/variables and checking if an assignment satisfying conditions (A) or (B), and (C), exists. And, this "new" version of the problem is more suitable for solving with balance machines. A possible solution is discussed below:

Given a set $S = \{x_1, x_2, \ldots x_n\}$ of cardinality n, we use a special type of balance machines to check if S can be partitioned or not. The machines have the following properties:
(i)The filler–spiller outlets let out liquid only in discrete "drops".
(ii) There is a maximum weight which each pan can hold.

$balance(L_1 + L_2 + \ldots + L_n,\ R_1 + R_2 + \ldots + R_n)$;
$balance(L_1 + R_1, x_1)$;
$balance(L_2 + R_2, x_2)$;
$\vdots$
$balance(L_n + R_n, x_n)$.

Note that the maximum weight which pans representing variables L_i and R_i can hold is x_i, and that they are filled up in "drops", each weighing x_i. This creates an "all or nothing" situation for the pans, thus satisfying either condition (A) or (B). When there *is* a way to partition the set, the balances would stop, assigning variables L_1, L_2, ..., L_n, R_1, R_2, ..., R_n with suitable values that constitute one such partition; when such a partition is impossible, the balances would keep "swinging" up and down, thus trying out various possible assignments forever, and do not come to a halt[5].

6 The 0/1 Knapsack problem

In this Section we attempt to solve yet another NP–complete problem using balance machines— the 0/1 Knapsack [3], which can be defined as follows:

Input: Objects representing "weights" and their corresponding "values": (w_1, v_1), (w_2, v_2), ..., (w_n, v_n), a knapsack capacity w and target value t.

Question: Is there an $A \subseteq \{1, 2, ..., n\}$ such that $\sum_{i \in A} w_i \leq w$ and $\sum_{i \in A} v_i \geq t$?

Let us initially consider a slightly modified version of the problem: we shall force $\sum_{i \in A} w_i = w$ and $\sum_{i \in A} v_i = t$. We can use an approach quite similar to the one used to solve the Set Partition problem.

Use n pockets $x_1, x_2, \ldots, x_n$ to represent weights and another set of n pockets $y_1, y_2, \ldots, y_n$ to represent their corresponding values (unlike set partition, we do not have "left" and "right" pockets). The following conditions are to be met with:
A) A pocket is either empty, or contains a weight/value: ($x_j = 0$ or $x_j = w_j$) and ($y_j = 0$ or $y_j = v_j$).

[5]We do not know of a way to actually *detect* this situation.

B) If a weight–pocket is empty, then so is the corresponding value–pocket: $x_j = 0 \Rightarrow y_j = 0$.
C) If a weight–pocket contains a weight, then the corresponding value–pocket will contain its value: $x_j = w_j \Rightarrow y_j = v_j$.
D) The weight–pockets should together weigh w and the value–pockets should weigh t: $x_1 + x_2 + \ldots + x_n = w$ and $y_1 + y_2 + \ldots + y_n = t$.

We make use of the special type of balance machines used in the case of Set Partition.

$balance(X_1 + X_2 + \ldots + X_n, w)$;
$balance(Y_1 + Y_2 + \ldots + Y_n, t)$;
$balance(X_1 + Y_1, Sum_1)$;
$balance(X_2 + Y_2, Sum_2)$;
$\vdots$
$balance(X_n + Y_n, Sum_n)$.

Note that the maximum weights which pans representing variables X_i, Y_i and Sum_i can hold are w_i, v_i and $w_i + v_i$ respectively and that they are are filled up only in "drops", each weighing w_i, v_i and $w_i + v_i$ respectively. This "all or nothing" restriction helps satisfy (A), (B) and (C). When there is a solution, i.e. a subset satisfying the given conditions, the balances would stop, assigning variables X_1, X_2, ..., X_n, Y_1, Y_2, ..., Y_n with values that constitute one such solution; when such a partition is impossible, the balances would keep "swinging" up and down, thus trying out various possible assignments forever, and do not come to a halt.

In fact, we *can* solve the original version of the 0/1 Knapsack Problem if we replace the first two balances with the following:

$balance(Extra_1 + X_1 + X_2 + \ldots + X_n, w)$;
$balance(Y_1 + Y_2 + \ldots + Y_n, t + Extra_2)$;
Now, due to the inclusion of $Extra_1$ on the left side, the quantity $X_1 + X_2 + \ldots + X_n$ could afford to be even less than w, since $Extra_1$ can fill itself up with whatever it takes to balance w. Likewise $Extra_2$ allows us to obtain more than the target value t.

7 Conclusions

We have introduced a formalism for computing based on balance machines. Computing using this formalism requires us to think differently and recast a computing problem as: "What values should x, y, z, etc. take so as to balance this and that and that?". We have demonstrated how to think in this fashion for two classic NP–Complete problems. Once recast in the manner just mentioned, a natural physical system such as a set of balances working together in parallel can spontaneously work out the answer (if it exists). Unlike a digital computer these machines do not seem to need a "program" detailing *how* to arrive at a solution. All we need to do is to code the constraints in the form of balances, and let them do the job.

The kind of parallelism one sees in balance machines is different from conventional parallelism. We get some sort of a "bilateral parallelism" which is worth further exploration in the future. Though we have not analyzed the time characteristics of balance machines, we believe that they will outperform digital computers when applied to NP–Complete problems, owing to their inherent bilateral parallelism.

References

[1] J. J. Arulanandham, C. S. Calude, M. J. Dinneen. Balance machines: Computing = balancing, *Lecture Notes in Comput. Sci.* 2933, Springer Verlag, Berlin, 2004, 36–48.

[2] J.J. Arulanandham, C.S. Calude, M.J. Dinneen. Solving SAT with bilateral computing, *Romanian Journal of Information Science and Technology* 6, 1–2 (2003), 9–18.

[3] M. Garey, D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.