# A First Attempt at Dedekind Numbers

Solaman Huq

April 13th, 2015

# 1 Introduction

The purpose of this paper is to give an explanation of my first attempt at calculating Dedekind Numbers. While the algorithm I use is far from optimal, it does have the nice property of constructing a small portion of all possible monotone boolean functions (even smaller than the number of inequivalent ones). While this might seem advantageous, there are definitely areas for improvement.

# 2 Terminology and Notation

Before continuing, it is prudent to introduce the terminology and definitions I will be using in the paper:

· **configuration** Will refer to the state of a boolean input.
e.g. 110110 would be a configuration for an MBF of input size = 6.

· When using a configuration, I note the input size as in the following example: 110110:d6.
I will always write out the full input, but this helps avoid confusion.

· **accepted configuration** A state of the boolean input in which an MBF would accept.
e.g. 10:d2 is an accepted configuration for $x \vee y$.

· **configuration level** Probably the more tedious definition, this will refer to the number of "off" inputs of *accepted* configurations.
e.g. The accepted configuration 111111:d6 would reside in the $0th$ configuration level, and $0101 : d4$ will reside in the 2nd configuration level.

# 3 Algorithm

## 3.1 Basic Algorithm

Note: we will use input size of 6 as an example throughout the paper.
The algorithm uses a Top-down approach to constructing MBF's. It will start with the MBF with no accepted configurations, then the MBF with one accepted configuration, {1111111}:d6 and then a combination of all of the configurations in the 1st level, namely, any of {011111, 101111, 110111, 111011, 111101, 111110 }:d6. We store these MBF's in a stack, and after this the algorithm starts to deviate.

But before continuing, note that any of these boolean functions are indeed monotone. If you take any of the 1st level configurations and flip the "off" bit to "on", you will get the 0th level configuration 111111:d6, which is also an accepted configuration.

From here on, we pop-off an MBF from the stack and generate all of its children MBF's and add them to the stack as well. How we generate those children, is all that is left to question.
We take the highest level $l_i$ of a given MBF, and take each of the configurations in them. We take every combination of $l_i + 1$ of them and perform a boolean and operation on them. If the result is a configuration that would reside on the $l_{i+1}$ level, then we add it to a list of potential configurations. When this is done, we take each combination of potential configurations add create a new MBF by adding them to the previous one, and storing this on the stack.

First, we will prove probably the more curious claim: that the configurations computed are indeed correct. If you take $l_i + 1$ configurations at the $l_i$ level, and accept them if the resulting configuration would reisde on the $l_{i+1}$ level, you are essentially saying that the $l_i + 1$ configurations have exactly $inputSize - l_{i+1}$ inputs in common. As there are $l_i + 1$ of them, each must differ in exactly one bit. If you then took the computed configuration, and flipped one of its "off" bits to "on", the resulting configuration would be that of one of the $l_i + 1$ configurations. Therefore, the configuration is monotonic with respect to the previous level.
Further, as any configuration computed in this way maintains this property, we can create an MBF by of adding any combination of them to the current MBF.

To help illustrate this claim, suppose we start at configuration level 3, we would then need 4 configurations to produce any level 4 configurations. Suppose we have {111000:d6, 110100:d6, 110010:d6, 110001:d6}. If we boolean and them, we have 110000:d6. We can see that if we flip any "off" bit of 110000:d6 to "on", it will turn into one of the configurations used to make it.

Inductively, this means that if a configuration is monotonic to its previous level, it is monotonic to all previous levels, and the function is therefore an MBF.

### 3.2 Equivalence Classes

Following the previous algorithm, we note that the total number of children for a particular MBF is only dependent on the configurations in its highest configuration level. Using this, we can reduce the overall running time of the algorithm by computing equivalence classes by level. Namely, if a set of configurations at level $l$ is equivalent to another set of configurations at level $l$, they would both produce the same number of children, so computing the children once is sufficient.

### 3.3 Areas of improvement

Currently, the algorithm calculates equivalent configuration sets by brute force. Namely, we use every permutation of the input bits to convert a configuration set to another. So for any configuration in d6, we would compute 6! different configurations! Through some minor analysis, we can see that many of these permutations lead to redundant configurations. Any algorithm that evolves from this one will have to use a very careful method in order to avoid this stupidly large overhead. Though, I imagine the more robust algorithms out there already have this built in to them.