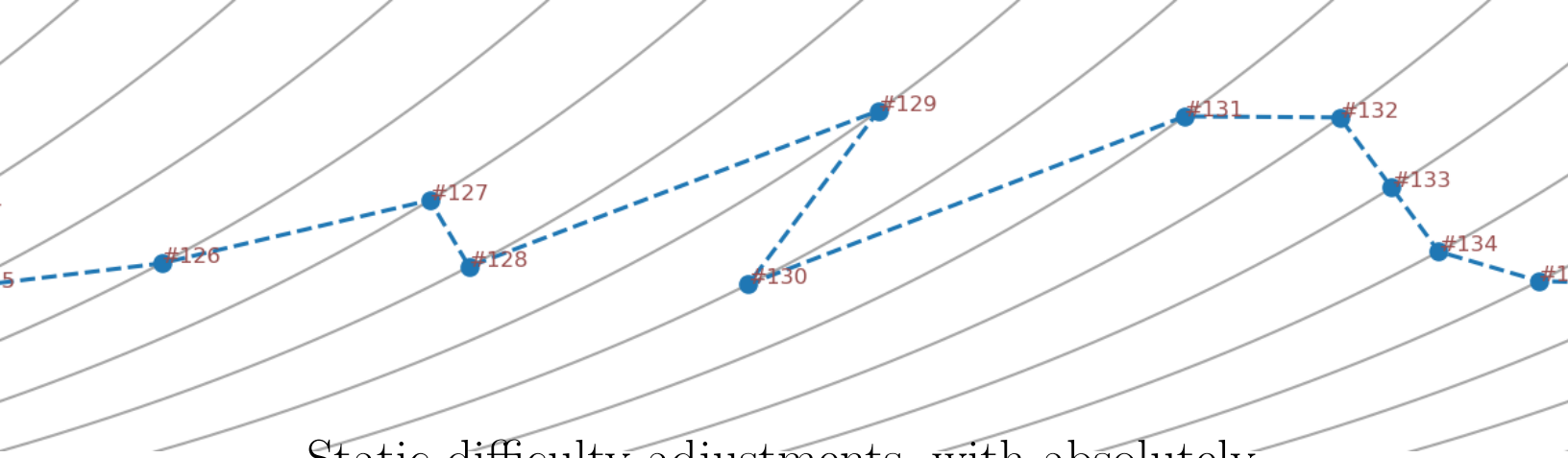

Static difficulty adjustments, with absolutely scheduled exponentially rising targets (DA-ASERT) — v.2

Mark B. Lundeberg

July 31, 2020

A version of this document was originally shared around in November 2019. Since it has gained a lot of attention in mid-2020, I've revised it. —M

There is an interesting cryptocurrency difficulty algorithm that has the following form:[1]

$$\text{target}_{N+1} = \text{target}_N \exp([t_N - t_{N-1} - T]/\tau), \tag{1}$$

where N is the current height of the blockchain, t_N is the timestamp of the most recent block, and t_{N-1} is the timestamp of the previous block. The parameter T is the targeted average block interval (e.g., $T = 10$ minutes on bitcoin). The parameter τ is a relaxation time and practically could be anywhere from (say) $50T$ to $1000T$.

Remarkably, this can be rewritten as an expression that makes no reference to the previous block's difficulty, using some constants $\text{target}_{\text{ref}}$, t_{ref}, and h_{ref} (more on their values later, see Sec. 1.3):

$$\text{target}_{N+1} = \text{target}_{\text{ref}} \exp([t_N - t_{\text{ref}} - (N - h_{\text{ref}})T]/\tau), \tag{2}$$

(to prove this equivalence, simply take (2) and calculate $\text{target}_{N+1}/\text{target}_N$). I refer to this as *absolutely scheduled exponentially rising targets* (ASERT) since, as can be seen, it is predetermined what will be the target_{N+1} in terms of t_N.

This document explores the properties of Eq. (1) ('relative exponential') and Eq. (2) ('absolute exponential') as candidate difficulty algorithms.[2]

[1] First suggested by Jacob Eliosoff as `simpexp`.

[2] Some more great discussion on this topic: "Unstable Throughput: ...".

Contents

1 Idealized properties

Since the ideal forms Eq. (1) and Eq. (2) are equivalent, we can discover useful properties on one form, map it to the other one, and vice versa. This section will explore such properties, whereas in the next section I'm going to explore how the equivalence can break down in practice.

1.1 Basic security against rapid-mining attacks

A common problem in difficulty algorithms is that extreme conditions of hashrate fluctuations or timestamp manipulation might cause the difficulty algorithm to go 'off-schedule' and start producing blocks at an average period faster than T. This is a concern since it would mean a malicious miner can abuse the algorithm, continuously collecting block subsidies much faster than the intended rate, thereby causing premature inflation and flooding the market.

The form (2) displays a remarkable property, which is that when you push the schedule ahead by an additional time $+\tau$, the difficulty must increase by a factor $\times e$, without exception. Sustained rapid mining would require getting more and more ahead of schedule, producing an exponentially rising difficulty, and thus after a short time (some multiple of τ) the rapid mining

would become impractically hard. Since the relative form (1) is equivalent, it inherits this property.

Most difficulty algorithms *approximate* this property (see Sec. 2.3 on drift) but the exponential algorithms let us exactly see and analyze it.

1.2 Memoryless

Remarkably, even though Eq. (2) has manifestly an absolute schedule, the algorithm practically also behaves as if it has a short term memory — this can be seen in the equivalence to Eq. (1), which only refers to the previous block. The algorithms are also 'memoryless' in the sense of hysteresis, or reversibility: if the difficulty rises and falls by any factor, the algorithm returns to exactly the same state it would have been in, without the perturbation. This can be seen more directly in Eq. (2), but it applies exactly to Eq. (1).

Note that this memorylessness of ASERT does break down when there is clipping (Sec. 2.1). In such cases the absolute schedule does manifest itself, in a very undesireable manner.

1.3 Significance and flexibility of reference constants

Eq. (2) can be rewritten to combine all the reference constants into one, i.e.,:

$$\begin{aligned} C &= \text{target}_{\text{ref}} \exp((-t_{\text{ref}} + h_{\text{ref}}T)/\tau) \\ \text{target}_{N+1} &= C \exp((t_N - NT)/\tau), \end{aligned} \tag{3}$$

which shows that the constants $\text{target}_{\text{ref}}$, t_{ref}, and h_{ref} have no independent meaning, and are devices I just made up. The constant C would of course be an astronomically large number but it would be no less meaningful.

Even though they are arbitrary, the three separated constants can be helpful for thinking about the behaviour, and a practical implementation might want to keep separate constants:

- You can choose a fixed height M, and assign: $\text{target}_{\text{ref}} = \text{target}_M$, $h_{\text{ref}} = M - 1$, and $t_{\text{ref}} = t_{M-1}$. This is convenient when the difficulty algorithm is activated: if M is the last block not under ASERT control, then these values ensure continuity of the difficulty exactly as if the

relative form Eq. (1) were activated instead.[3] (This approach was used in the prior version of this document, but it is by no means the only way to see things.)

- Nearly identical (but distinct) behaviour would be obtained by setting: $\text{target}_{\text{ref}} = \text{target}_M$, $h_{\text{ref}} = M$, and $t_{\text{ref}} = t_M$. Note this doesn't correspond to activation of the relative algorithm but it is close.

- For simplicity it may be helpful to assign $t_{\text{ref}} = 0$ (Unix epoch time), or $h_{\text{ref}} = 0$, or $\text{target}_{\text{ref}} = 1$, to make those values disappear from the equation. If one such replacement is done, that still leaves one arbitrary degree of freedom. Two such replacements leaves just one remaining constant.

- If you are starting a new cryptocurrency project, you might consider setting t_{ref} to the genesis time t_0, set h_{ref} to 0, and $\text{target}_{\text{ref}}$ to an initial block-1 difficulty based on an educated guess of the expected initial economics and hashrate supply.

1.4 Generalizations

There is a generalized form of Eq. (2):

$$\text{target}_{N+1} = \text{target}_{\text{ref}} \exp([f(t_N, t_{N-1}, \ldots) - (N - h_{\text{ref}})T - t_{\text{ref}}]/\tau), \quad (4)$$

where $f(t_{\text{a}}, t_{\text{b}}, \ldots, t_m)$ is a function that computes some kind of typical timestamp out of the provided recent timestamps, which primarily should weight onto *recent* timestamps for good response. For example, $f()$ might use a mean, a weighted mean, a median, a maximum, or something else. The simpler equation (2) is a special case for $f(t_{\text{a}}, t_{\text{b}}, \ldots) = t_{\text{a}}$. Converting this back to the relative form in terms of the last block, this becomes:

$$\text{target}_{N+1} = \text{target}_N \exp([f(t_N, t_{N-1}, \ldots) - f(t_{N-1}, t_{N-2}, \ldots) - T]/\tau), \quad (5)$$

which shows how any complex time-mixing-averaging process can be properly incorporated to not have irreversibility defects. For simplicity I'm not going

[3] It might seem that you could take this and set $M = N - n$ as a moving reference point, but this is not equivalent. This creates an very badly behaving algorithm due to decoupling (it will show oscillations with a period of $n+1$). The only useful value is $n = 0$, in which case you recover Eq. (1) exactly. So, either $M = N$, or $M =$ constant.

to discuss Eqns. (4),(5) much, but many of the nice properties of interest described herein will also apply to these general forms. A practical warning, though: inclusion of older timestamps may produce oscillations, even with a large τ.

It is also easy to convert these algorithms into a real-time form, by involving t_{N+1} into the equation as well. Then, the difficulty of mining can drop gradually over time even without a block actually being produced yet. Remarkably, this converts the simplest absolute form Eq. (2) into one that is independent of the previous block's information, and yet it functions perfectly well.

1.5 Approximate equivalance to 'WTEMA'

We can take (1) and assume large τ, so that the argument is going to be generally small, and approximate $\exp(z) \approx 1 + z$. This produces a difficulty algorithm known as WTEMA:[4]

$$\text{target}_{N+1} = \text{target}_N \cdot (1 - \alpha + \alpha x), \tag{6}$$

where $x = (t_N - t_{N-1})/T$, and $\alpha = T/\tau \ll 1$.

I would argue that (1) (or (2)) is what WTEMA was "meant to be", as it corrects the major flaw: the target passes through 0 (difficulty has a singularity) when $x = 1 - \alpha^{-1}$. This requires a large backwards timestamp jump but is disastrous if it happens. WTEMA also lacks the perfect reversibility of the exponential forms, which can be seen as a flaw. On the other hand, the close connection to (2) explains why WTEMA works so well, and Sec. 1.1 explains why it too is intrinsically secure against rapid-mining attacks: the WTEMA difficulty is never easier than the exponential one. We also see that (5) provides a general template for how WTEMA ought to be modified to use median times, etc.: just set $x = (f(t_N, t_{N-1}, \ldots) - f(t_{N-1}, t_{N-2}, \ldots))/T$. So, if one wants an ultra-simple algorithm that doesn't require calculating exponentials, then WTEMA is a fantastic choice (supposing the singularity is inaccessible due to timestamp ordering rules).

1.6 Rather unique

It might be asked, are there other absolutely scheduled algorithms which make sense to try? As far as I can tell right now, no – the algorithm needs to

[4] via Tom Harding

have a time-translation invariance property, along with scale-free (multiplicative) increases/decreases in difficulty. This naturally leads to an exponential form. There are of course many different forms of the function $f(t_{\mathrm{a}}, t_{\mathrm{b}}, \ldots)$ that could be tried in the generalized version Eq. (4).

2 Discussion and practicalities

2.1 Clipping

It may seem attractive to take a mathematically simple difficulty algorithm, and then introduce 'clipping': limits on the upward and downward movement of the difficulty relative to the prior block. Unfortunately, it is actually quite dangerous to artificially restrict the downward movement of target (upward movement of difficulty). It may be possible for an attacker to exploit this limitation and use unusual sequences of block timestamps in order to pump the difficulty up and down, each time hitting the limit, so that on average the difficulty is driven down and the attacker can mine many blocks with low effort.[5]

Until now I've been treating Eq. (1) and Eq. (2) as equivalent, and as explained in Sec. 1.1 they both have a basic immunity to timestamp manipulations. But if clipping is introduced, then these two forms (relative and absolute) are no longer equivalent.

A previous version of this document suggested that the absolute form (Eq. (2)) is still safe for clipping (unlike Eq. (1)). This is partly true: if an attacker tries to pump the absolute algorithm, they cannot succeed because the algorithm will not forget the 'right' difficulty level. However, there is another concern: adding clipping to the absolute version means that if the clipping limits are actually hit, then the algorithm may stick to the limit for a too-long time, and overshoot the proper endpoint. Depending on the reaction of the miners, this can induce stalls, bursts, and oscillations.

With the relative version, increases in difficulty (decreases in target) must not be clipped, as this is precisely what opens the door to exploits. On

[5] See "timespan limit attack" described at `https://github.com/zawy12/difficulty-algorithms/issues/30`; Note that in some algorithms, such exploits may even be possible without artificial clipping. In general these exploits tend to require non-monotonic timestamps and so a strong defense against unknown exploits is to have a timestamp monotonicity rule.

the other hand, I am not aware of any exploits arising from the clipping of *decreases* in difficulty (increases in target), in a relative algorithm.[6] Still, I would recommend against doing any kind of clipping.

On a realistic blockchain there is one inevitable form of clipping: there is a fundamental maximum to the target value. The hard limit is set by the hash bitwidth, though in the case of bitcoin the rule is arbitrarily tighter than this by a factor of 2^{32}. When the difficulty algorithm hits such a limit, it counts as a difficulty clipping, and the absolute version (Eq. (2)) will be susceptible to overshoot, whereas the relative version (Eq. (1)) will behave more reasonably. Anyway, this should only be happening when the hashrate is very low, i.e., the chain in question has no market value.

2.2 Computation

It is of course not a good idea to rely on floating point math in a real DA due to possibly inconsistent rounding between different math libraries / CPUs,[7] so, how does one do an exponential algorithm?

As an example, equation (2) can be recast in the following form using integer bit shifts and multiplies (recall that targets are in fact very large integers), essentially performing an explicit mixture of low precision fixed point and floating point math (here, / indicates floor division).

$$a = (b[t_N - t_{\text{ref}} - (N - h_{\text{ref}})T] + 2^{15})/2^{16}$$
$$\text{target}_{N+1} = [\text{target}_{\text{ref}} \times 2^{a/2^{16}-16}] \cdot [f(a \bmod 2^{16})]$$

where $b = \text{round}(2^{16} \cdot [\tau^{-1}T] \ln 2)$ and $f(N)$ is some appropriate polynomial that approximates $2^{16}2^{N/2^{16}}$ over the interval $N = 0 \ldots 65535$. This only needs to be accurate to the $\sim 0.1\%$ level to be entirely satisfactory; a third-order polynomial provides $\sim 0.01\%$.

It is of course crucial in an implementation to specify the exact values of the reference constants, so that every implementation and installation can calculate reproducible values. There are many ways to do this, due to the freedoms mentioned in Sec. 1.3.

[6] The WTEMA algorithm (Eq. (6)) actually exhibits a natural sort of 'soft clipping' of extreme difficulty decreases, when looked at in comparison to the exponential form. This has various side-effects but on the whole they might be neutral or even helpful.

[7] Even though IEEE754 requires perfect consistency for basic arithmetic, it does *not* for transcendentals like exp(), and there are various other pitfalls even for basic arithmetic.

With such an absolute difficulty algorithm, any small errors will get immediately forgotten on the next block, so it works fantastically even with a coarse approximation to exponential. As far as I can tell this is the only real practical advantage of using absolute instead of relative, and this trades off with its poor clipping behaviour. In the relative version Eq. (1), a higher accuracy is required, though some small rounding errors are acceptable. It might appear that that even small errors could compound over and over without limit, but in practice the block production rate gives feedback to the difficulty algorithm, and so the actual effect is bounded and tends to self-compensate in the long run. Note though: errors in the relative algorithm do enable rapid-mining attacks, so errors must be kept quite small in magnitude.[8]

2.3 Drift

Recently there has been discussion on the topic of drift: a measurement of how far the chain has deviated from exactly one block per time T, for desired time T. This, of course, depends on where you are measuring relative to. In general if we use a starting height S with timestamp t_S, then the drift for block N (assumed $> S$) is defined as:

$$\text{drift}(N, S) = (N - S)T - (t_N - t_S), \tag{7}$$

which is positive if the average time per block has been less than T. If we try to define an *absolute drift*, there are various reasonable choices for S: the genesis ($S = 0$), or the time when the difficulty first became unclipped, or the time when the difficulty algorithm first activated, etc. In any case, the choice of S only affects the *offset* of the absolute drift. This means that relative changes in drift are well defined, so let's talk about them here.

Under the most basic possible situation, that of a steady hashrate, a difficulty algorithm will have a particular average time per block, and we can assume the algorithm is properly designed to exactly match the desired T value in this most basic situation;[9] thus, almost by definition, the drift value should not change appreciably under steady hashrate.

[8] For example, suppose the miners can use timestamp selection to consistently induce multiplicative errors of $(1 + \epsilon)$ on Eq. (1). They can then steadily mine blocks with an average time $(1 - \epsilon\tau/T)T$. E.g., if $\epsilon = 10^{-4}$ and $\tau/T = 10^3$, they can mine at 10% advantage. Note that if the response time τ is lengthened, their advantage *increases*.

[9] A subtlety: this might be different from the constant named T in the algorithm itself.

The situation is different with non-steady hashrate. Standard difficulty algorithms have short term memory in the sense that they only observe recent time differences, and so they are unaware and uncaring of the total value of drift. This naturally tends to cause an accumulation of drift in the situation of rising hashrate, as exemplified by the following thought experiment. Suppose the hashrate rises by 10%, so blocks are coming 10% too fast. The difficulty algorithm has some typical reaction time τ before it "notices" this (the reaction time must be fairly long to not be badly affected by randomness of mining), and once this happens it resets the difficulty to a 10% higher level. Because the time of 10% faster blocks lasted a time of about τ, this means that the drift increased by about $+0.1\tau$. Repeating this process will add more drift each time. Notice that a longer τ produces more drift.

As a case study we can look at the BTC chain, which has used a simple periodic averaging algorithm since inception. For the first year or so (until roughly block 36000), the hashrate was too low and the difficulty was clipped to the minimum value.[10] The block time averaged around 15 minutes, compared to $T = 10$ minutes, and this year of slow blocks caused the drift (measured from genesis) to *decrease* steadily, reaching a low point around -140 days. Then, enough hashrate came in to push above minimum difficulty. Since that time, the hashrate has steadily risen, by an overall factor of $X \approx e^{30}$, and the current drift has risen by $+370$ days since the low point.

It may be noticed that the definition of drift looks very closely related to the argument of the exponential in the absolute exponential algorithm (it is especially obvious in Eq. (3)). This close connection means that changes in drift for the exponential algorithm (either relative or absolute) can be derived exactly. If hashrate increases by a factor of X then difficulty increases by the same factor (plus or minus temporary variations). If the difficulty increases by a factor of X then the drift must increase by precisely:[11]

$$\Delta\text{drift} = +\tau \ln(X).$$

For a perfect exponential algorithm, the T constant is precisely the effective T. However, if WTEMA is done with a large value of α, then mining variance causes the time per block (the effective T) to be noticeably longer than the T value in the formula (but, this depends on the timestamp distribution). This raises interesting questions of whether compensation should be done, and what attacks that might open up. Practically, α will be chosen to be quite small and this distinction is minor.

[10] `https://btc.com/stats/diff`

[11] The absolute drift is: $\tau \ln \frac{\text{target}_{\text{ref}}}{\text{target}_{N+1}} - (t_{\text{ref}} - t_S + (S - h_{\text{ref}})T)$.

This is a corollary to the security argument in Sec. 1.1, where it was remarked that the difficulty changes in the exponential DAAs are directly related to changes in drift.

As indicated by the thought experiment, non-exponential algorithms will tend to have a similar behaviour, increasing by $+\tau \ln(X)$ for some characteristic value of τ, provided the hashrate increases at a suitably gentle slope. In the above BTC study we can see that it indicates a reaction time τ of (370 days)/ $\ln(e^{30})$ = 12.3 days, which unsurprisingly is close to the algorithm's difficulty retargeting interval of 2016 blocks. But, non-exponential algorithms will additionally have some excess drift contribution (plus or minus) if the hashrate changes too quickly. Non-exponential algorithms may also have a gradual drift accumulation even with steady hashrate (due to randomness of block times, or due to bugs), though this is usually small in magnitude since it's an easily detected defect.

In all proper short-term-memory algorithms (not ASERT), accumulation of rounding errors also causes drift, but this should typically be negligible compared to other drift sources. Clipping events also cause offsets in drift (but not in ASERT, which fights to get back to its preferred drift vs difficulty curve, with unfortunate consequences). Since large errors and clipping ought not to be happening in a well-designed system, it is reasonable to view the absolute and relative exponential algorithms as having equivalent drift.