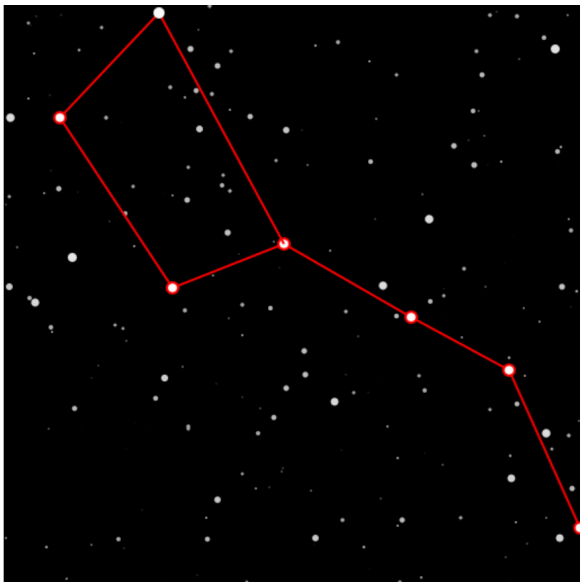
Dipper Proof-of-work

@Datavetaren

January 15, 2019



Abstract

Proof-of-work is the mechanism to prove that work has been done. It typically amounts to a ("random") search problem where once a solution is found, it is easy (and requires almost no resources) to verify. Traditionally, these proof-of-work algorithms have solely been focused on CPU power, but new algorithms have emerged that take advantage of other resources such as memory. As the value of cryptocurrencies go up there has been a race among vendors to provide application specific circuits (ASICs) which bring an advantage over traditional off-the-shelf hardware for that particular proof-of-work problem. This comes with pros & cons: specialized hardware dedicated to a certain cryptocurrency secures its network, but as specialized hardware is a complicated and expensive business it generally leads to a handful companies providing the ASICs (first for themselves) leading to miner centralization. In this paper I'll present a new proof-of-work algorithm that tries to cope with this problem.

1 Introduction

In cryptocurrencies (e.g. bitcoin [3]) proof-of-work is used to guard transaction history and enforce ordering. By embedding digital signatures with a proof-of-work stamp and by sequencing proof-of-work stamps, it becomes hard and expensive to rewrite history. It is currently the only known method that is resistent to history rewrite attacks (also called "reorgnizations.") Once a comfortable amount of proof-of-work has been stacked, the users within the network can be relatively confident that their transaction can't be reversed. To ensure the network stays resistent, a difficulty parameter is also added and tuned once the network accumulates more computing power. By tuning this parameter we can ensure real-world time correlates with the proof-of-work being done.

Traditionally, the only resource being considered in proof-of-work algorithms has been computing power, but lately a new class of algorithms, such as "Cuckoo Cycle" proof-of-work [4], also enforces utilization of memory. The hope is that making the proof-of-work algorithm memory-hard makes it more difficult to create application specific circuits (ASICs) that solves the proof-of-work in much less time than conventional off-the-shelf hardware/computers. This specialized hardware typically results in making the proof-of-work network more centralized.

In this paper I'll describe a new proof-of-work memory-hard algorithm that is different from "Cuckoo Cycle" but shares some of its properties. The distinction however is by choosing a non-trivial problem that represents a whole class of existing and interesting problems, makes it meaningless to construct hardware so specialized for no other use case beyond solving the proof-of-work itself.

2 Method

The flowchart 1 provides an overview of the method being used. Given a seed for (semi) random generator we generate a star cluster (or galaxy.) This star cluster is designed so that in order to find a solution it is more or less required to keep the data points sorted in memory. In the standard design we require 2^{27} (= 134,217,728 stars) being generated which requires about 2.4 GB of memory, but the exact amount varies depending on how you choose your data structures.

The quest is to find the star constellation "The Big Dipper" N number of times (in the standard design N = 16) and each round will provide what

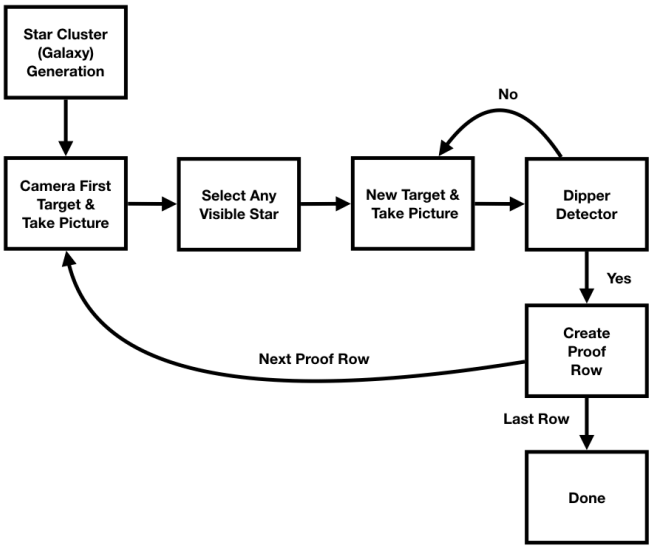


Figure 1: Algorithm Outline

is called a proof row. The final proof will consist of N rows.

The orientation of the camera is, as the generation of stars, determined by the seed value and a nonce. Everything is calibrated so the uniform distribution of stars and the sensitivity of the lens yields approximately the same number of stars being rendered regardless of how the camera is orientated and zoomed.

In each proof round, before looking for the "Big Dipper," we require the proof of one visible star from the first camera target. Then the camera target is changed, and we keep changing the target until the "Big Dipper" is found. The "random star visibility proof" is added to prevent the user from cheating, as its details are explained in a section 8. Once the "Big Dipper" is found we can record the identifiers (32-bit) of the 7 stars that form the constellation. The entire proof round just consists of 9 32-bit integers.

Once we have a final proof it requires limited resources to verify it. We don't need to render the entire galaxy, just the stars that is given by each proof row. Thus, very limited memory (and CPU) is required to verify a proof. This is in line with other memory-hard proof-of-work algorithms.

3 Star Setup

We'll be using the $siphash(key, i)$ [2] function to compute semi-random values given a seed key and an arbitrary index value i. All parameters are 64-bit unassigned integers including its return value. In general we'll use $siphash(key, i)$ as an abbreviation to $siphash(Blake2(key), i)$ where Blake2 is the hash function applied to the key message.

The first step is to compute a star cluster that is uniquely generated via key (or seed.) Each star is expressed in Cartesian coordinates (x,y,z) constrained in a finite region of space. We'll define the position of each star as:

$$star_x^i = normalize(siphash(key, 3i)) \quad (1)$$
$$star_y^i = normalize(siphash(key, 3i + 1)) \quad (2)$$
$$star_z^i = normalize(siphash(key, 3i + 2)) \quad (3)$$
$$(4)$$

The range of i ($0 \leq i < max$), (the number of stars to be generated,) is defined by the super-difficulty parameter, which I'll talk about later.

These unassigned 64-bit integer coordinates can be reinterpreted as values within $-0.5 \leq v < 0.5$ using the transformation:

$$normalize(v) = \frac{v}{2^{64} - 1} - \frac{1}{2}$$

It should be noted that in a consensus system it would be unwise in general to use floating point numbers to carry out the actual computations. Even though the IEEE 754 standard [1] defines the representation of floating point numbers, it does not define the exact behavior of all functions applied on this representation. For example, it's not uncommon to use 80-bit registers for floating point computations where the memory representation is IEEE 754 (which at most fits 64 bits.) Thus, the combination of hardware vendor and/or version of your compiler, its register allocation and spilling strategy may affect the outcome of the end result. It's in fact impossible to get 100% equal behavior on all platforms. Therefore, you should in general never use the equality operator on floating point numbers.

To overcome this floating point deficiency, all computations that are part of the consensus computation verify everything via either a custom fixed point representation or a custom floating point representation. As a miner you are free to use hardware floating point for your calculations, but

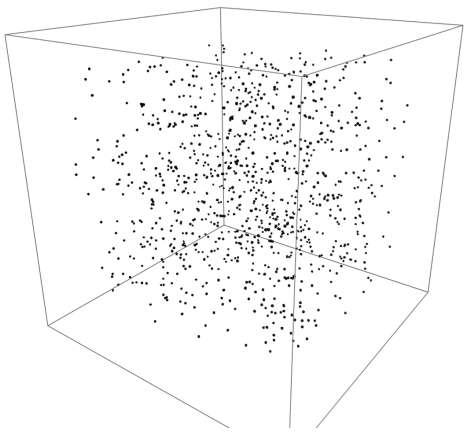


Figure 2: Stars in space

to verify your solution you need to use the built-in fixed-point representation that is very likely much slower. But if only a solution needs to be verified, that performance degradation isn't the majority of the overall computation. Furthermore, empirically, I haven't been able to produce a solution on my hardware where the fixed-point version fails. This is because the result is verified within some tolerance. And the tolerance set is such that it should work properly for most of the time. Worst case, as a miner, you'd have to discard a detected solution and find another one.

4 Grid Partitioning

To make the problem scale invariant we'll partition the space into sub cubes, or a grid. The parameter n will determine the super-difficulty of this proof-of-work such that the number of sub cubes along each axis is 2^n. For example, if $n = 8$ we'll partition the space into $256 \times 256 \times 256$ sub cubes.

The number of stars S_n will be determined by the formula:

$$S_n = 2^{3n+3}$$

So for $n = 8$: $S_n = 2^{27} = 134{,}217{,}728$ stars (134 million.) For every $n \geq 1$ the density of stars will become 8 stars per unit or sub cube.

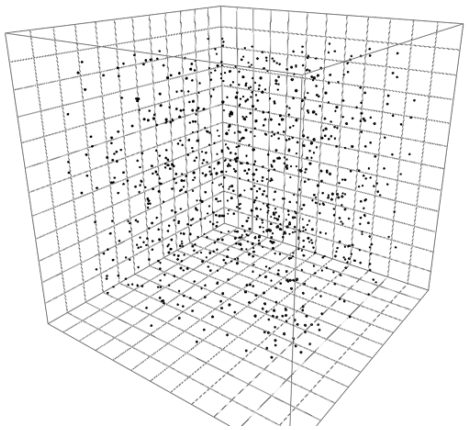


Figure 3: Space with Grid

5 The Camera

In this region of space we position a camera at coordinates (0,0,0) and pinpoint at certain coordinates (x,y,z) and project those through the lens to see if we can find "The Big Dipper" (orientation and scale invariant) within a certain tolerance.

The field of vision is captured by the volume of a pyramid (see Figure 4.) The orientation of the camera can be described by some vector algebra. If the position of the camera is $\boldsymbol{C} = (0, 0, 0)$ and the target point is $\boldsymbol{P} = (P_x, P_y, P_z)$ we can compute the direction vector as $\boldsymbol{D} = \boldsymbol{P} - \boldsymbol{C} = \boldsymbol{P}$. If we assume the existance of a proxy up vector (0,0,1) then the plane orientation of the pyramid is:

$$dir_x = \boldsymbol{D}_{norm} \times (0, 0, 1) \quad (5)$$

$$dir_y = \boldsymbol{D}_{norm} \times dir_x \quad (6)$$

Where $\boldsymbol{D}_{norm}$ is the normalized (unit) vector of $\boldsymbol{D}$, i.e. $\|\boldsymbol{D}\| = 1$, or

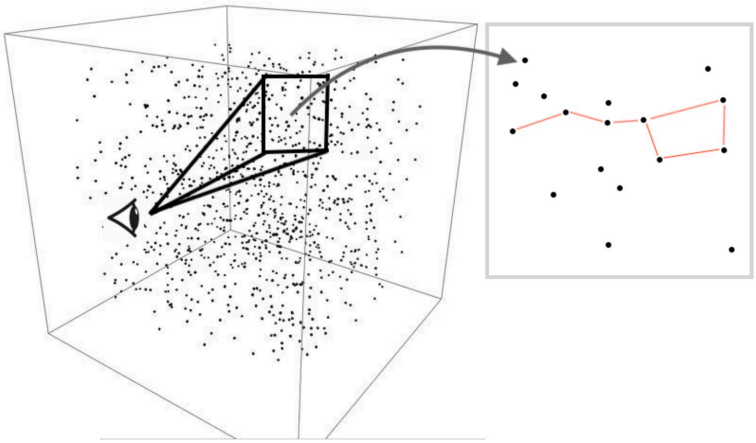


Figure 4: Looking for Big Dipper

$$D_x = \frac{D_x}{\sqrt{{D_x}^2 + {D_y}^2 + {D_z}^2}} \tag{7}$$

$$D_y = \frac{D_y}{\sqrt{{D_x}^2 + {D_y}^2 + {D_z}^2}} \tag{8}$$

$$D_z = \frac{D_z}{\sqrt{{D_x}^2 + {D_y}^2 + {D_z}^2}} \tag{9}$$

If we fix the volume of the pyramid, then the vision field/volume will be the same regardless where the target point is located. Therefore, if the camera zooms in on an object, the lens will become more sensitive and the end result is that the same number of stars will approximately be projected regardless of chosen target.

Let's make the volume constant to be $V = 32$ (sub cubes) then we'll approximately see $S_V = 8 \cdot 32 = 256$ stars. The height h of the pyramid is $\|\boldsymbol{D}\|$, recall that $\boldsymbol{D} = \boldsymbol{P}$ and the using the formula of the volume of a pyramid we can compute the volume base length as: $l = \sqrt{\frac{32 \cdot 3}{h}}$.

By sorting the stars into sub cubes, we can quickly approximate the sub cubes that need to be considered, basically moving along the direction vector and consider the points within the pyramid. For each star within each relevant sub cube we calculate:

$$S_x = (\boldsymbol{S} \cdot dir_x)\frac{\|\boldsymbol{P}\|}{l\|\boldsymbol{S}\|} \tag{10}$$

$$S_y = (\boldsymbol{S} \cdot dir_y)\frac{\|\boldsymbol{P}\|}{l\|\boldsymbol{S}\|} \tag{11}$$

S_x and S_y is the the projection of the star, and it is on the display if $-0.5 \leq S_x < 0.5$ and $-0.5 \leq S_y < 0.5$ (same as ${S_x}^2 < 0.25$ and ${S_y}^2 < 0.25$.) We also need to check that $\|\boldsymbol{S}\| < \|\boldsymbol{D}\|$.

6 The Detector

Once we projected all the stars on the screen the next step is to make the detector that finds "The Big Dipper." In the previous section we've computed the projection of all stars on the screen, let's denote those as the set Σ.

The algorithm is relatively naïve. For every pair of stars $(s_i, s_j) \in \Sigma$ we assume it matches the first two stars in the "Big Dipper." From this we can compute the rotation and scaling matrix that reorients the "Big Dipper" into this position. Then we look for stars within the remaining spots (by using a template of the "Big Dipper" and apply the rotation & scaling matrix.) If we can find all the stars of "Big Dipper" within a specified tolerance, then we're done. Let's define delta of the two stars as $ds_{ij} = s_j - s_i$ and the delta of the two template stars $dt^{12} = t_2 - t_1$, then rotation and scaling matrix becomes:

$$R = \frac{1}{\|dt^{12}\|}\begin{bmatrix} (dt_x^{12}ds_x^{ij} + dt_y^{12}ds_y^{ij}) & (dt_x^{12}ds_y^{ij} - dt_x^{12}ds_y^{ij}) \\ (dt_x^{12}ds_y^{ij} - dt_x^{12}ds_y^{ij}) & (dt_x^{12}ds_x^{ij} + dt_y^{12}ds_y^{ij}) \end{bmatrix} \tag{12}$$

The remaining position of the other stars can be computed as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = R\begin{pmatrix} dt_x^{1k} \\ dt_y^{1k} \end{pmatrix} + \begin{pmatrix} s_x^i \\ s_y^i \end{pmatrix} \tag{13}$$

We've chosen a really tolerance of 8 pixels as it is quite easy to find the "Big Dipper" otherwise. Of course, the tolerance is scaled as well with $\frac{\|ds^{ij}\|}{\|dt^{12}\|}$ as the transformation either magnifies (or shrinks) the template coordinates.

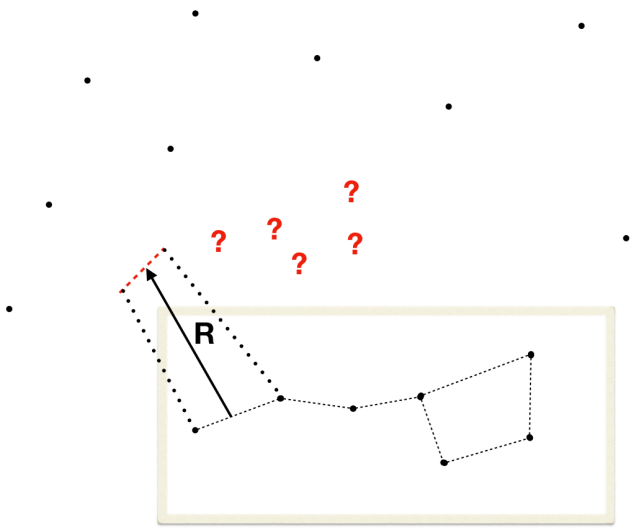


Figure 5: Detecting Big Dipper

7 Scanning

The camera is fixed at origin (0,0,0) and the target point is defined by:

$$P_x(M,I) = normalize(siphash(Key, M2^{32} + 3I)) \quad (14)$$
$$P_y(M,I) = normalize(siphash(Key, M2^{32} + 3I + 1)) \quad (15)$$
$$P_z(M,I) = normalize(siphash(Key, M2^{32} + 3I + 2)) \quad (16)$$

Here I is the iteration (nonce) and we attempt to find the smallest I such that the "Big Dipper" is found using the detector algorithm described in the previous section.

M is the nonce offset and the final nonce is computed by adding it with a factor of 2^{32} to I. The nonce offset is computed so it is the sum of all previous nonces of the previous proof rounds (0 if we run the first proof round.) The idea here is to make it as difficult as possible for the user to guess what region of space we're browsing. Otherwise, the miner could precalculate the region of space of interest and optimize the data structures to reduce memory consumption. As this proof-of-work is supposed to be memory-hard it would defeat its purpose otherwise.

8 Verification

For each proof round, it can be quickly verified by 9 (32-bit) integers. 7 for the identification numbers for the stars that form the constelltation ("Big Dipper".) The first two integers are:

1. One 32-bit integer to represent the identification number of a visible star. (Camera nonce is computed from the sum of all previous nonces, or 0 if it's the first proof round.)

2. One 32-bit integer to represent the nonce for which the "Big Dipper" is found.

One may ask why the first integer is needed. Imagine that the miner only chooses to compute for all stars except for those where $X < 0$. Then (as stars are uniformly distributed) only half the stars need to be stored in memory, and the frequency of detecting the "Big Dipper" will be reduced by 50% as well. So in essence you can trade memory for more computing power. However, by first proving the visibility of a star in a random direction of

Table 1: Final proof table for seed "hello42"

	Star Id	Nonce	Star 1	2	3	4	5	6	7
0:	0x02b0b23a	3661	0x02fce78e	0x07dddf36	0x0317fea9	0x0259cd01	0x01e9cf33	0x0645e995	0x029f7899
1:	0x0742d211	6475	0x02500b81	0x05b96272	0x000b265f	0x06b3d6df	0x06bec53a	0x062e1809	0x07d2d434
2:	0x03ffd407	96	0x07b0ec40	0x0357c963	0x01dd1477	0x01e2bc17	0x0305e087	0x0573fa0c	0x0048d0b1
3:	0x051a8408	745	0x02d79a09	0x07391c00	0x04fe0924	0x05534c39	0x00cedbc3	0x0464608f	0x05fef0f5
4:	0x07892cdc	974	0x053281cc	0x02d90498	0x055c81f2	0x0369acca	0x02cc25c2	0x01ef5d14	0x06c16cc3
5:	0x017620f8	422	0x0342bcd2	0x00089f65	0x04922f96	0x00d18122	0x06c9f69a	0x02c0c1e3	0x0318d092
6:	0x068d4870	725	0x01b704c7	0x0240194a	0x046cfd68	0x059dac60	0x0699cd53	0x06c0826d	0x01535867
7:	0x053bab7e	105	0x04299322	0x017ba0f9	0x03ba4baf	0x07a70b44	0x00618c3f	0x0616cf48	0x05429640
8:	0x0645a02b	1991	0x0232faae	0x0780a6bd	0x06f11a37	0x05a5290b	0x00319d8d	0x07e6a409	0x0560fed2
9:	0x0599f1fd	2691	0x07d7da20	0x075f5855	0x056c05cf	0x0193d3b5	0x056c52d5	0x02b1057e	0x05fa03db
10:	0x01589ee6	2362	0x03eed026	0x05173285	0x03f5c32f	0x0550cdce	0x0478af9e	0x016ee98a	0x05687533
11:	0x024bd110	1575	0x0063ee93	0x044a9809	0x010873fc	0x04ffae40	0x01659204	0x07d3f284	0x01cab5fb
12:	0x06a26456	1679	0x0702f7dd	0x00b4d9a6	0x028d5fbc	0x01e33b30	0x04d1e657	0x07da4c5c	0x032e7a99
13:	0x00373e17	1220	0x04b07fbf	0x002b76c1	0x059e3298	0x0556241f	0x07f96ab8	0x0154413f	0x02d4764c
14:	0x0166366a	563	0x05c95d2d	0x05bc5c7f	0x07f337bc	0x0459d279	0x05163267	0x0713f7be	0x056cc93f
15:	0x06373ebb	3664	0x05688427	0x05da67dd	0x06dd783b	0x0318f912	0x022bb402	0x05f640e3	0x03e96d3e

the camera, it would be very difficult to come up with a proof where large non-uniform portions of stars have been excluded.

It should be noted that if one attempts to remove half of the stars uniformly (or randomly), then the visibility proof is large unaffected, but then the chance of detecting the "Big Dipper" is reduced significantly. For example, imagine you have detected the "Big Dipper" with all stars included and then remove half of those stars on the screen you're observing, then the probability that the existing "Big Dipper" is unaffected is approximately $0.5^{3.5}$ or 8.9%. Thus, a uniform removal of half the stars would require more than 10x more computing power. We can also complement the proof row with the expection value of finding the "Big Dipper." Empericial studies have shown that it's very rare to get more than $20,000$ iterations for one proof row. So if the nonce is required to be less than $20,000$, then it becomes almost impossible to come up with a proof where one have removed half of the stars uniformly.

With these extra percautions, only 8 stars have to be rendered for verífying each proof row. And if we use 16 proof rows, a total of 8×16 star renderings are required, which is still limited in terms of resources being used for verification.

In table 8 we've shown the complete proof for seed "hello42." Note the nonces in the second column are nonces found within that proof row. Let's quickly go through this table step by step.

The first row does not have any previous rows, so the nonce offset is 0. We then compute the nonce for the visiblity of the star with id 0x02b0b23a. The star's coordinates are:

$$
\begin{aligned}
S_x^{\texttt{0x02b0b23a}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02b0b23a} \cdot 3)) && (17)\\
&= normalize(\texttt{0x74fb5bbb55a740cc}) = -0.0430396 && (18)\\
S_y^{\texttt{0x02b0b23a}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02b0b23a} \cdot 3 + 1)) && (19)\\
&= normalize(\texttt{0x7239fc1b2b25e1ce}) = -0.0538027 && (20)\\
S_z^{\texttt{0x02b0b23a}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02b0b23a} \cdot 3 + 2)) && (21)\\
&= normalize(\texttt{0x74604de995b0ecd2}) = -0.0454055 && (22)
\end{aligned}
$$

The camera target to use to verify the visibility of this star is:

$$
\begin{aligned}
P_x &= normalize(siphash(\text{"}hello42\text{"}, 0)) && (23)\\
&= normalize(\texttt{0x405de8e964c10000}) = -0.248567 && (24)\\
P_y &= normalize(siphash(\text{"}hello42\text{"}, 1)) && (25)\\
&= normalize(\texttt{0x30f3e2e3f9cd0000}) = -0.308779 && (26)\\
P_z &= normalize(siphash(\text{"}hello42\text{"}, 2)) && (27)\\
&= normalize(\texttt{0x3cdbe1b5b1ca0000}) = -0.262270 && (28)
\end{aligned}
$$

Thus, if star coordinates are defined by the vector $S = (-0.0430396, -0.0530827, -0.0454055)$ and the camera target point is $P = (-0.248567, -0.308779, -0.262270)$, the distances from the point of origin $(0, 0, 0)$ are:

$$
\begin{aligned}
\|\boldsymbol{S}\| &= \sqrt{0.0430396^2 + 0.0538027^2 + 0.0454055^2} = 0.0825154 && (29)\\
\|\boldsymbol{P}\| &= \sqrt{0.248567^2 + 0.308779^2 + 0.262270^2} = 0.475305 && (30)
\end{aligned}
$$

The length at the "base of the pyramid" is $\sqrt{\frac{V \cdot 3}{\|\boldsymbol{P}\|}} = 0.00346968$. V is the standard volume $V = \frac{32}{256^3}$ where $256 = 2^n$ with $n = 8$ (the super difficulty.)

From the camera target point we compute the direction vectors of the plane (of the lens):

$$
\begin{aligned}
\boldsymbol{dir_{up}} &= (0, 0, 1) && (31)\\
\boldsymbol{dir_x} &= \frac{\boldsymbol{P}}{\|P\|} \times \boldsymbol{dir_{up}} = (-0.778965, 0.627067, 0) && (32)\\
\boldsymbol{dir_y} &= \frac{\boldsymbol{P}}{\|P\|} \times \boldsymbol{dir_1} = (0.346011, 0.429827, -0.833982) && (33)
\end{aligned}
$$

The projections of $\boldsymbol{dir_x}$ and $\boldsymbol{dir_y}$ then become:

$$\boldsymbol{proj_x} = \frac{1}{0.00346968} \frac{\|\boldsymbol{P}\|}{\|\boldsymbol{S}\|} (\boldsymbol{S} \cdot \boldsymbol{dir_x}) = -0.351317 \tag{34}$$

$$\boldsymbol{proj_y} = \frac{1}{0.00346968} \frac{\|\boldsymbol{P}\|}{\|\boldsymbol{S}\|} (\boldsymbol{S} \cdot \boldsymbol{dir_2}) = -0.250114 \tag{35}$$

Thus, the two projections are within the range $-0.5 \leq x \leq 0.5$ and $\|\boldsymbol{S}\| \leq \|\boldsymbol{P}\|$, so it is indeed visible.

The next step is to verify that the "Big Dipper" is detected if we render the stars with id's `0x02fce78e`, `0x07dddf36`, `0x0317fea9`, `0x0259cd01`, `0x01e9cf33`, `0x0645e995` and `0x029f7899` using nonce 3661. Thus the coordinates of the first star are:

$$\begin{aligned}
S_x^{\texttt{0x02fce78e}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02fce78e} \cdot 3)) && (36)\\
&= normalize(\texttt{0xb9edd458100dfb14}) = 0.226285 && (37)\\
S_y^{\texttt{0x02fce78e}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02fce78e} \cdot 3 + 1)) && (38)\\
&= normalize(\texttt{0xb6343b7d22a7a31a}) = 0.211735 && (39)\\
S_z^{\texttt{0x02fce78e}} &= normalize(siphash(\text{"}hello42\text{"}, \texttt{0x02fce78e} \cdot 3 + 2)) && (40)\\
&= normalize(\texttt{0x46e7179e1d19a919}) = -0.223036 && (41)
\end{aligned}$$

The target of the camera is defined by:

$$\begin{aligned}
P_x &= normalize(siphash(\text{"}hello42\text{"}, 2^{32} \cdot 3661)) && (42)\\
&= normalize(\texttt{0xc2d3cf0cb277683a}) = 0.261044 && (43)\\
P_y &= normalize(siphash(\text{"}hello42\text{"}, 2^{32} \cdot 3661 + 1)) && (44)\\
&= normalize(\texttt{0xbf29863b41647a08}) = 0.246727 && (45)\\
P_z &= normalize(siphash(\text{"}hello42\text{"}, 2^{32} \cdot 3661 + 2)) && (46)\\
&= normalize(\texttt{0x3e6009ea269e1954}) = -0.256347 && (47)
\end{aligned}$$

Carrying out the computations for the other stars give the 3D coordinates and the corresponding 2D projections. I've also included the actual screen coordinates which are the normalized coordinates $-0.5 \leq x < 0.5$ mapped to $0 \leq x < 4096$.

Star Id	X	Y	Z	proj_x	proj_y	screen_x	screen_y
0x2fce78e	0.226285	0.211735	-0.223036	0.499184	0.489081	4092	4051
0x7dddf36	0.194579	0.182518	-0.191105	0.37706	0.215228	3592	2929
0x317fea9	0.179385	0.168843	-0.176187	0.207017	0.123377	2895	2553
0x259cd01	0.131669	0.124483	-0.129311	-0.0141417	-0.00279087	1990	2036
0x1e9cf33	0.16479	0.156398	-0.162349	-0.206084	0.0742827	1203	2352
0x645e995	0.23583	0.224704	-0.231588	-0.403034	-0.220803	397	1143
0x29f7899	0.251744	0.23904	-0.246032	-0.230847	-0.403915	1102	393

At this point we need to compute the transformation matrix for the "Big Dipper" template. The code uses these coordinates for the "Big Dipper" template

Star	X	Y
1	0	0
2	185	116
3	325	99
4	502	93
5	588	-30
6	830	63
7	805	236

The (2D) transformation matrix is computed by comparing the basis of the two vectors $(185, 116)$ and $(-500, -1122) = (x_1 - x_0, y_1 - y_0) = (3592 - 4092), (2929 - 4051)$. At this point all calculations are made in pure integers (we avoid all custom fixed/floating point.)

Taking the dot product of these two vectors will give us the X axis of the base vector. The Y axis is obtained by rotating the vector 90 degrees counter clockwise $(x, y) \rightarrow (y, -x)$:

$$R = \frac{1}{\sqrt{185^2 + 116^2}} \begin{pmatrix} R_{00} & R_{10} \\ R_{01} & R_{11} \end{pmatrix} = \begin{pmatrix} -1021 & -686 \\ 686 & 1021 \end{pmatrix} \quad (48)$$

$$R_{00} = (-500, -1122) \cdot (185, 116) = -222652 \quad (49)$$

$$R_{01} = (-500, -1122) \cdot (116, -185) = 149570 \quad (50)$$

$$R_{10} = -R_{01} \quad (51)$$

$$R_{11} = R_{00} \quad (52)$$

This transformation matrix will take the template coordinates and translate them to the coordinate system of interest using:

$$(x', y') = (x_i, y_i)(R) + (x0, y0) \quad (53)$$

Star	template	Translated	Actual	Error
3	(325,99)	(2882,2565)	(2895,2553)	$\|(13, -12)\| = 17.7$
4	(502,93)	(2034,2036)	(1990,2036)	$\|(-44, 0)\| = 44$
5	(588,-30)	(1244,2342)	(1203,2352)	$\|(-41, 10)\| = 42.2$
6	(830,63)	(403,1145)	(397,1143)	$\|(-6, -2)\| = 6.3$
7	(805,236)	(1065,413)	(1102,393)	$\|(37, -20)\| = 42.1$

The tolerance of the template is set to 8 pixels. This tolerance is translated as well, so we get tolerated error $8 \cdot \frac{\sqrt{500^2+1122^2}}{\sqrt{185^2+116^2}} = 45$ pixels. As seen, all the errors shown above are less than 45 pixels.

This was the first row of the proof. The second row will use nonce offset $3661 + 1 = 3662$. The third row will use nonce offset $3661 + 6475 + 2$ and so on (recall that nonce offset adds $2^{32} \cdot (offset)$ to the nonce.)

Performance of verifying a complete proof takes about 280 μs on a MacBook Pro 2.4 GHz with 8 GB 1600 MHz DDR3 RAM. And then I haven't really optimized the code.

9 Fine Tuning

As computers become faster, or more computers are joining the proof-of-work network, we need a parameter to fine tune its difficulty. The easiest way to achieve this is to put an additional constraint on the final solution. Let's say: $Blake2(solution) < H < 2^{256}$ where a descreasing H will make it harder to find an accepted solution. In essence, this is the strategy used in bitcoin [3] but here used as an additional constraint on the "Big Dipper" solution itself. (Same approach is taken in "Cuckoo Cycle.")

10 Course Tuning

We can set the super-difficulty using the parameter n. The setup is tuned so that only more memory is consumed while having all the other properties constant.

Using $n = 8$ consumes approximately 2.4 GB of memory, but is of course dependent on how you build up your data structures and what kind of optimizations you apply. $n = 9$ consumes 8 times more memory, so around 20 GB. The problematic aspect of just increasing n is that specialized hardware may make it difficult for off-the-shelf hardware to compete. Thus, although the current framework makes it possible to tune super-difficulty (automatically as well), it's determined that this is may be an unwise idea.

If specialized hardware comes online that solves the "Big Dipper" proof-of-work algorithm fast, it's likely that the same hardware can be used for general purpose vector algebra computations on big datasets. To quickly render 2.4 GB of memory using floating point operations with vector algebra is likely to be useful for many other applications such as computer graphics and/or computational physics. Therefore, if the hardware has other

use cases, it'll increase the chances of making that hardware more general purpose.

11 Conclusion

This proof-of-work algorithm is far from trivial. It is designed to consume a lot of memory and computation using floating point numbers. It is hard to come up with specialized hardware for computing this specific proof-of-work only, rather it makes more sense that incorporates computations on big datasets with floating point numbers and vector algebra in general; something that is useful for a big class of problems, e.g. computer graphics and/or computational physics. Thus, if specialized hardware is pursuited it is more likely to become off-the-shelf hardware for other applications as well. This by itself should reduce, but not eliminate, miner centralization.

References

[1] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[2] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: A fast short-input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, pages 489–508, 2012.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, May 2009.

[4] John Tromp. Cuckoo cycle: a graph-theoretic proof-of-work system, 2014.