ethash算法伪代码

```python
def mkcache(cache_size, seed):
    o = [hash(seed)]
    for i in range(1, cache_size):
        o.append(hash(o[-1]))
    return o
```

这个函数是通过seed计算出来cache的伪代码。

伪代码略去了原来代码中对cache元素进一步的处理，只展示原理，即cache中元素按序生成，每个元素产生时与上一个元素相关。

每隔30000个块会重新生成seed（对原来的seed求哈希值），并且利用新的seed生成新的cache。

cache的初始大小为16M，每隔30000个块重新生成时增大初始大小的1/128 ——128K。

```python
def calc_dataset_item(cache, i):
    cache_size = cache.size
    mix = hash(cache[i % cache_size] ^ i)
    for j in range(256):
        cache_index = get_int_from_item(mix)
        mix = make_item(mix, cache[cache_index % cache_size])
    return hash(mix)
```

这是通过cache来生成dataset中第i个元素的伪代码。

这个dataset叫作DAG，初始大小是1G，也是每隔30000个块更新，同时增大初始大小的1/128——8M。

伪代码省略了大部分细节，展示原理。

先通过cache中的第i%cache_size个元素生成初始的mix，因为两个不同的dataset元素可能对应同一个cache中的元素，为了保证每个初始的mix都不同，注意到i也参与了哈希计算。

随后循环256次，每次通过get_int_from_item来根据当前的mix值求得下一个要访问的cache元素的下标，用这个cache元素和mix通过make_item求得新的mix值。注意到由于初始的mix值都不同，所以访问cache的序列也都是不同的。

最终返回mix的哈希值，得到第i个dataset中的元素。

多次调用这个函数，就可以得到完整的dataset。

这个函数通过不断调用前边介绍的calc_dataset_item函数来依次生成dataset中全部full_size个元素。

```python
def calc_dataset(full_size, cache):
    return [calc_dataset_item(cache, i) for i in range(full_size)]
```

```python
def hashimoto_full(header, nonce, full_size, dataset):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, dataset[dataset_index])
        mix = make_item(mix, dataset[dataset_index + 1])
    return hash(mix)

def hashimoto_light(header, nonce, full_size, cache):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, calc_dataset_item(cache, dataset_index))
        mix = make_item(mix, calc_dataset_item(cache, dataset_index + 1))
    return hash(mix)
```

这个函数展示了ethash算法的puzzle：通过区块头、nonce以及DAG求出一个与target比较的值，矿工和轻节点使用的实现方法是不一样的。

伪代码略去了大部分细节，展示原理。

先通过header和nonce求出一个初始的mix，然后进入64次循环，根据当前的mix值求出要访问的dataset的元素的下标，然后根据这个下标访问dataset中两个连续的的值。

（思考题：这两个值相关吗？）

最后返回mix的哈希值，用来和target比较。

注意到轻节点是临时计算出用到的dataset的元素，而矿工是直接访存，也就是必须在内存里存着这个1G的dataset，后边会分析这个的原因。

```python
def mine(full_size, dataset, header, target):
    nonce = random.randint(0, 2**64)
    while hashimoto_full(header, nonce, full_size, dataset) > target:
        nonce = (nonce + 1) % 2**64
    return nonce
```

这是矿工挖矿的函数的伪代码，同样省略了一些细节，展示原理。

full_size指的是dataset的元素个数，dataset就是从cache生成的DAG，header是区块头，target就是挖矿的目标，我们需要调整nonce来使hashimoto_full的返回值小于等于target。

这里先随机初始化nonce，再一个个尝试nonce，直到得到的值小于target。

```python
def mkcache(cache_size, seed):
    o = [hash(seed)]
    for i in range(1, cache_size):
        o.append(hash(o[-1]))
    return o

def calc_dataset_item(cache, i):
    cache_size = cache.size
    mix = hash(cache[i % cache_size] ^ i)
    for j in range(256):
        cache_index = get_int_from_item(mix)
        mix = make_item(mix, cache[cache_index % cache_size])
    return hash(mix)

def calc_dataset(full_size, cache):
    return [calc_dataset_item(cache, i) for i in range(full_size)]

def hashimoto_full(header, nonce, full_size, dataset):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, dataset[dataset_index])
        mix = make_item(mix, dataset[dataset_index + 1])
    return hash(mix)

def hashimoto_light(header, nonce, full_size, cache):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, calc_dataset_item(cache, dataset_index))
        mix = make_item(mix, calc_dataset_item(cache, dataset_index + 1))
    return hash(mix)

def mine(full_size, dataset, header, target):
    nonce = random.randint(0, 2**64)
    while hashimoto_full(header, nonce, full_size, dataset) > target:
        nonce = (nonce + 1) % 2**64
    return nonce
```

这里是整个流程的伪代码，同时分析矿工需要保存整个dataset的原因。

红色框标出的代码表明通过cache生成dataset的元素时，下一个用到的cache中的元素的位置是通过当前用到的cache的元素的值计算得到的，这样具体的访问顺序事先不可预知，满足伪随机性。

由于矿工需要验证非常多的nonce，如果每次都要从16M的cache中重新生成的话，那挖矿的效率就太低了，而且这里面有大量的重复计算：随机选取的dataset的元素中有很多是重复的，可能是之前尝试别的nonce时用过的。所以，矿工采取以空间换时间的策略，把整个dataset保存下来。轻节点由于只验证一个nonce，验证的时候就直接生成要用到的dataset中的元素就行了。

以太坊总供应量

ETHER DISTRIBUTION OVERVIEW

Genesis (60M Crowdsale+12M Other):	72,009,990.50 Ether
+ Mining Block Rewards:	25,903,969.97 Ether
+ Mining Uncle Rewards:	1,818,301.25 Ether
= Current Total Supply	**99,732,261.72 Ether**

Data Source: Total Eth Supply API

$ PRICE PER ETHER

In USD:	$518.77
In BTC:	0.07321

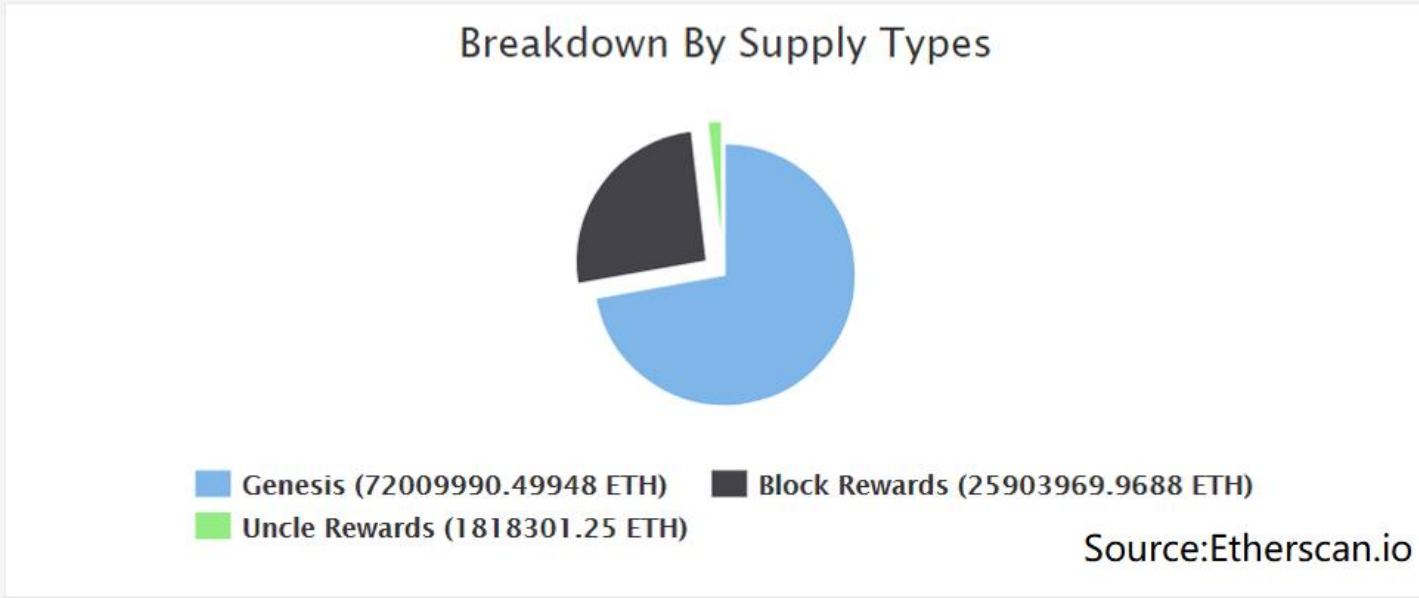
Data Source: CryptoCompare

99,732,261.72

Total Ether Supply

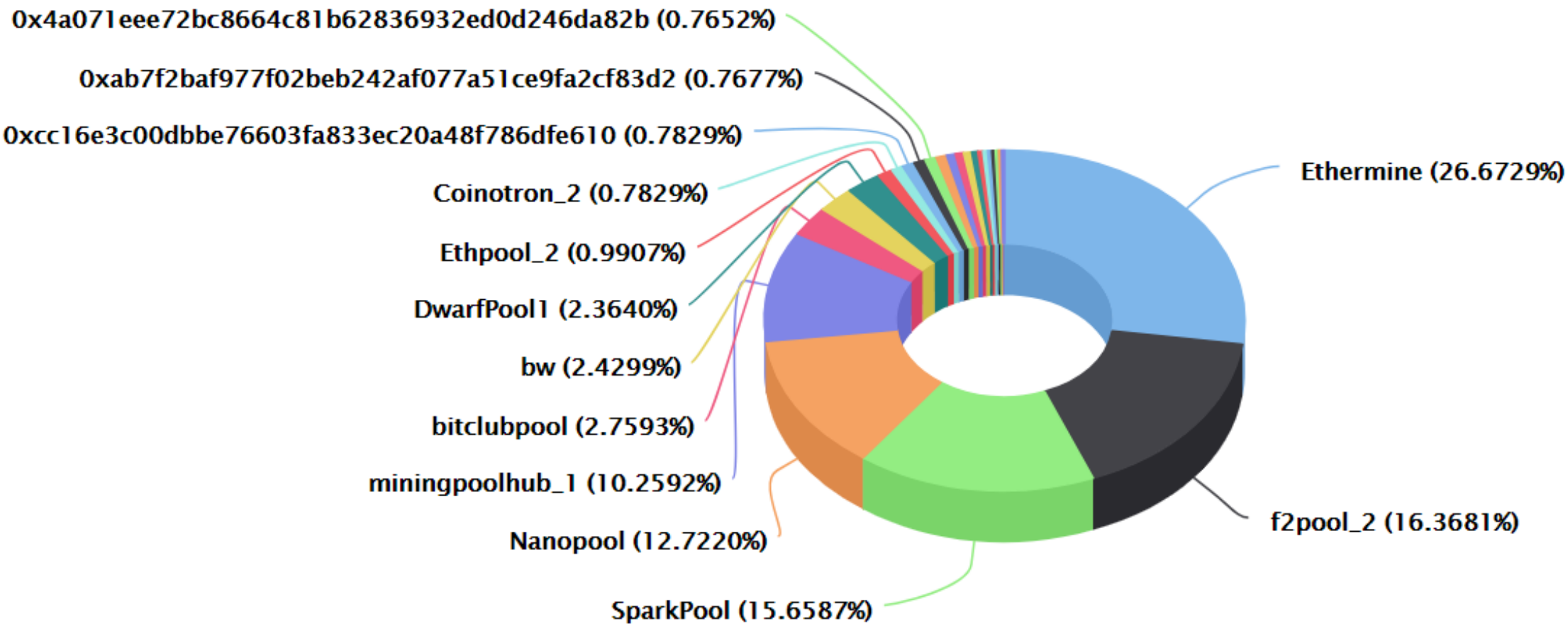
$51,738,105,411

Market Capitalization



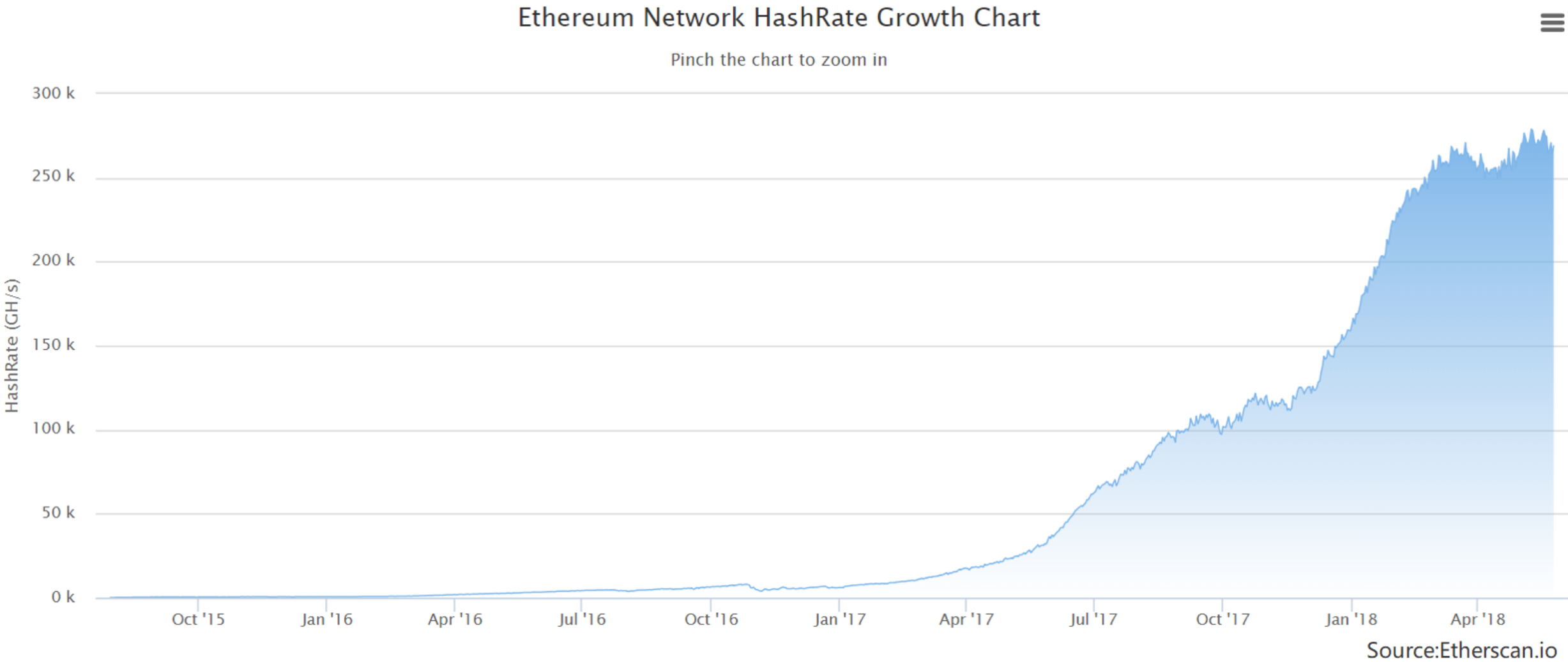
Ethereum Top 25 Miners by BLOCKS

In The Last 7 Days
Source: Etherscan.io




Ether Historical Prices (USD)
Pinch the chart to zoom in
Ether Price (USD)
1500
1250
1000
750
500
250
0
Oct '15
Jan '16
Apr '16
Jul '16
Oct '16
Jan '17
Apr '17
Jul '17
Oct '17
Jan '18
Apr '18
Source:Etherscan.io


Ether Historical Market Capitalization Chart (USD)
Pinch the chart to zoom in
Ether Market Cap (USD)
150 B
125 B
100 B
75 B
50 B
25 B
0 B
Oct '15
Jan '16
Apr '16
Jul '16
Oct '16
Jan '17
Apr '17
Jul '17
Oct '17
Jan '18
Apr '18
Source:Etherscan.io


Ethereum Network HashRate Growth Chart
Pinch the chart to zoom in
300 k
250 k
200 k
150 k
100 k
50 k
0 k
HashRate (GH/s)
Oct '15
Jan '16
Apr '16
Jul '16
Oct '16
Jan '17
Apr '17
Jul '17
Oct '17
Jan '18
Apr '18
Source:Etherscan.io