MIT Digital Currency Initiative and the University of Brasilia presents

Cryptocurrency Design and Engineering

Lecture 19: Market Incentives
Taught by: Pieter Wuille
Date: November 18th, 2025
MAS.S62

About me

- I discovered Bitcoin in 2010
  - On a chat channel about Haskell language
- Started contributing to reference code in 2011, full-time since 2014
  - It was a different time: there were **no** tests
- Worked on variety of protocol & code improvements
- My current main project: cluster mempool, will talk about later
- This lecture: Bitcoin focus, though principles are more general

Network nodes & market incentives

The motivating questions for this lecture are how to deal with two hard problems for the network:

- How to keep node running permissionless (DoS resistance)
- How to keep mining permissionless

While they are very different problems, in both cases the answer will be to rely on market incentives, as cryptography does not suffice (but it helps!).

Synchronization at a high level

Network nodes' roles are essentially synchronizing information and validating it:

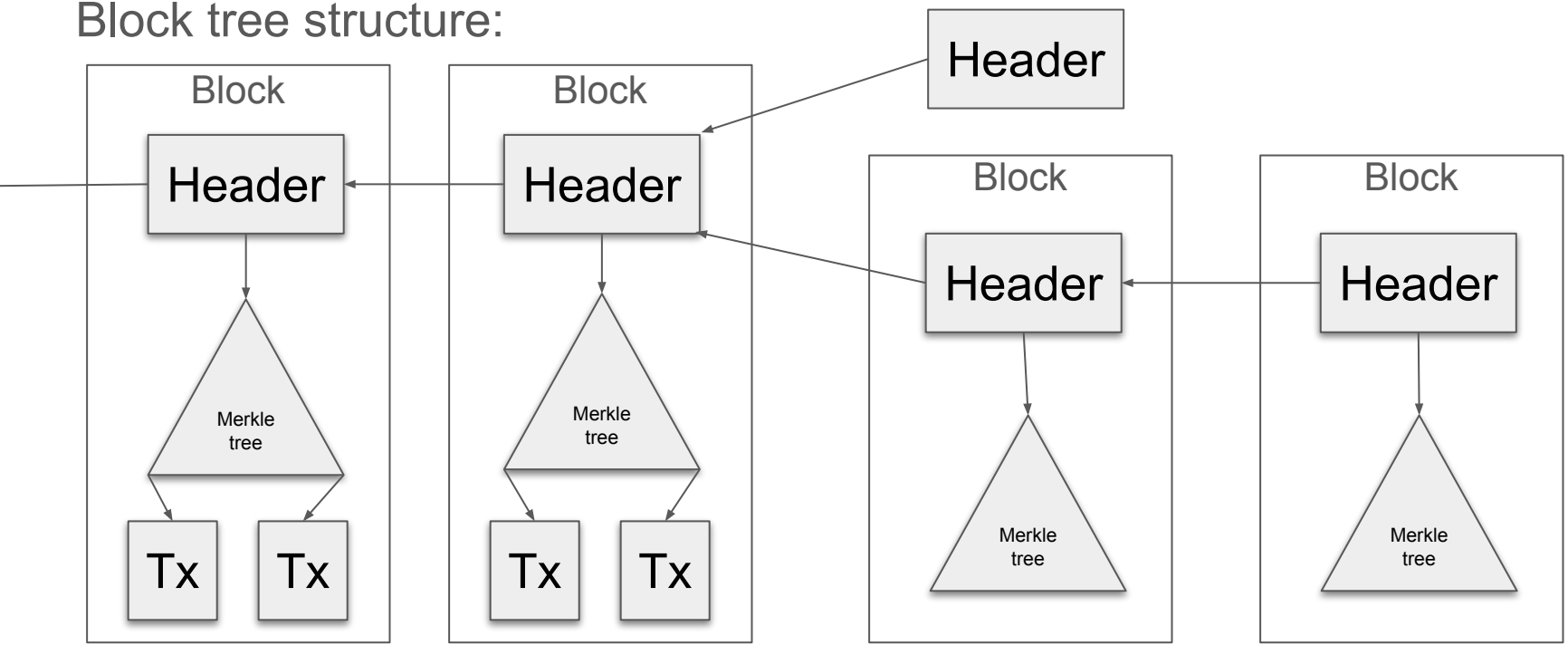
- Block data (headers + included transactions)
  - Rate limited by retarget + block weight limit
- Mempool (transaction data, before being mined):
  - Less rate limited…
- Node metadata (IP addresses)

This lecture will focus on block data (Part 1) and transaction data (Part 2), while retaining DoS resistance and permissionlessness.

Part 1: block synchronization

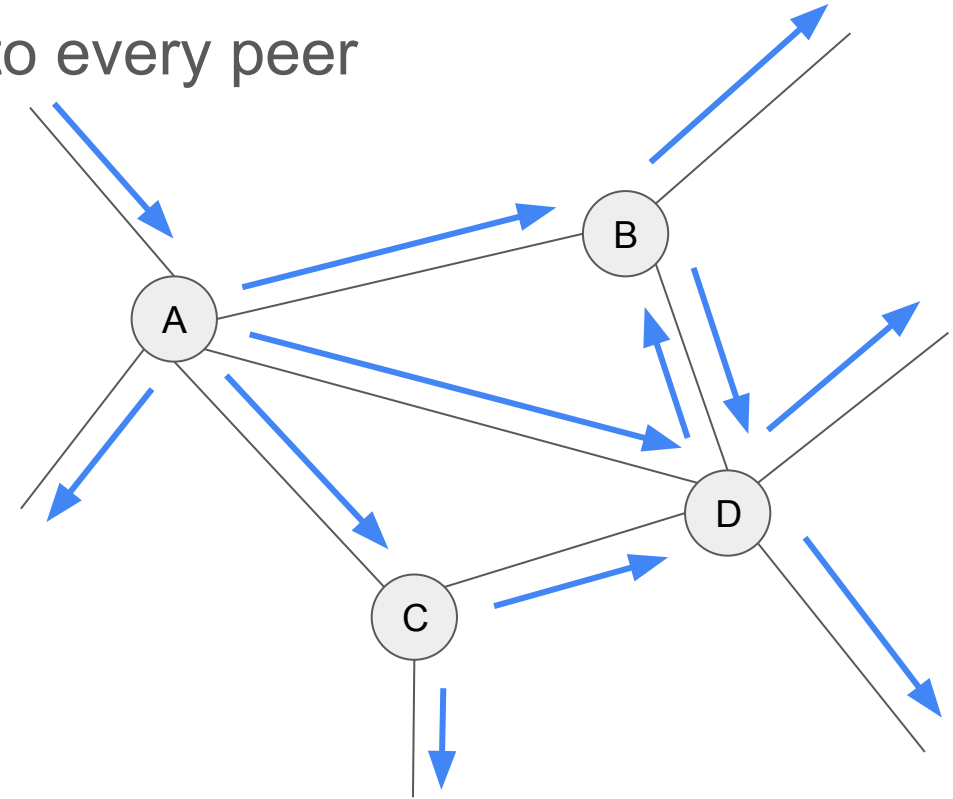
Part 1: DoS-resistant block synchronization

Block tree structure:



Block sync: attempt 1

Just send every new block to every peer



Wasteful:

- O(n*m bandwidth)
  - n = nodes
  - m = peers/node
- O(n) should suffice

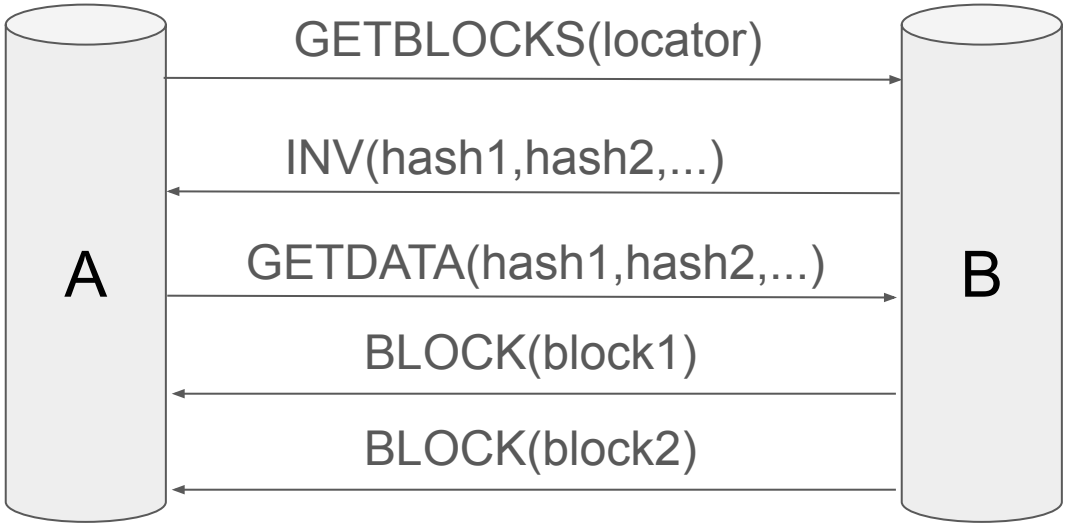
Block sync: attempt 2

Announce hash of every new block (“inventory”) and let peer ask for full data if they do not already have:

- -> INV(hash)
- <- GETDATA(hash)
- -> BLOCK(block)

What if you just restarted? Ask for all inventories?

Block sync: Bitcoin in 2009



Block sync: orphan block problem

Real problem: orphan blocks (block without known parent)

- Peers may announce block you do not have parent for.
- What to do? Cannot verify without parent!
  - Store on disk until parent arrives? Disk fill attack.
- Attack: cheap to create garbage block, expensive to verify.

Old solution: orphan pool, but this stopped working around 2014.

Better sync protocol was possible, but there are more problems.

Block sync: worthless chain problem

Even with valid PoW, attackers can fill disk:

- Fork off at genesis, create long, valid, low-difficulty chain.
- The receiver does not know if this chain it's being fed ends up with a most-work tip, until the end.
- Storing it is a disk-fill DoS
- Solution back then: hardcoded checkpoints in the software
  - Confuse security model, change in consensus!
- Checkpoints gone in Bitcoin Core v30 (just released)

Block sync: inherently sequential protocol

Even ignoring these problems, block synchronization is painful:

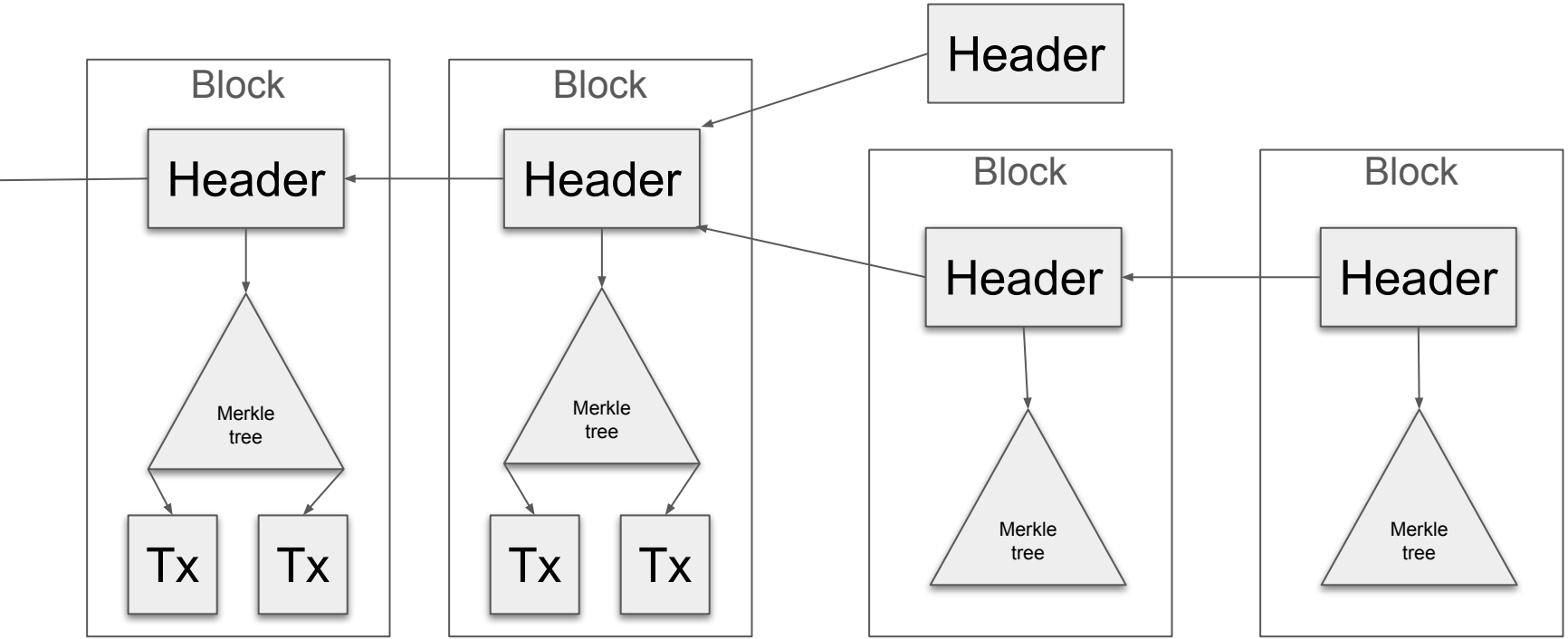
- We can only validate block after parents known
- We cannot store blocks permanently before validating

Result: hard to download multiple blocks in parallel

Observation:

- We do not need parent blocks to validate PoW, just headers!
- Full block validation still needs parent blocks, but valid-PoW headers are already expensive to create.

Block sync: Bitcoin 2014: headers-first sync



Block sync: Bitcoin 2014: headers-first sync

Idea: two phases (though mostly parallel):

- Header synchronization:
  - From all peers, learn about their best chains
  - Validate PoW, store headers
- Block synchronization:
  - Only download blocks for validated, promising, headers
  - No more orphan blocks
  - Spread over all peers, in any order
  - Fully validate (scripts, UTXOs) once all blocks arrive

Block sync: Bitcoin 2014: headers-first sync

Principle:

- Download and verify things that are cheap to validate, or expensive to create first.

Headers vs blocks is an extreme case of this!

Many more cases where it is less pronounced.

Block sync: Bitcoin in 2022: headers presync

With headers-first sync, we eliminated the disk-fill attack for **blocks**, but the issue still exists for **headers**:

- During header sync we do not know if chain will end up being valuable.
- Checkpoints protect against header spam, but ugly security model, and by 2022, unmaintained and old.
- CVE-2019-25220

Possibility: fancy NiPoPow, but hard to deploy, urgent…

Block sync: Bitcoin in 2022: headers presync

Solution: three phases (mostly in parallel)!

- Header presync: download, validate, but do not store headers, only tiny commitment.
- Header resync: download, compare, store headers.
- Block sync: like before, download & validate along best header chain.

Result: reduce need for checkpoints to known work.

Block sync: Bitcoin in 2022: headers presync

Headers presync: remember 1-bit salted hash per ~600 blocks

Headers redownload: verify with ~15000 header lookahead buffer

Block sync: Bitcoin miscellaneous

BIP130 (“sendheaders”): allow nodes to announce new blocks by blasting out header directly, avoiding INV roundtrip.

BIP152 (“compact blocks”): don’t send full transaction data, just short hashes, allow receiver to reconstruct from mempool, or request missing.

Block sync: summary

Thanks to the asymmetry of PoW (expensive to create, cheap to validate), we can effectively protect against DoS by downloading/validating headers before transaction data.

Blocks being rate-limited by PoW too effectively bounds node download/validation cost.

Part 2: mempool synchronization

Mempool sync: reason for mempool

Why?

- How do transaction creators (wallets) get transactions to miners?
- How do people assess transaction fees?
- For compact blocks, how do we make sure receivers have (most) transactions already?

Mempool sync: naive solution

Naive solution: miners announce their existence and endpoint IP. Wallets connect and submit directly to miners:

- Makes mining not anonymous (unless rely on Tor)
- How to announce IPs? Blockchain controlled by existing miners.
- Hurts permissionlessness of becoming miner.

Why do we need decentralized consensus / PoW?

Censorship resistance!

- If we were ok with a fixed set of block-content deciders, just give them all a key, require 50% to sign, get rid of PoW.
- Miners include all competitive tx, because otherwise new miners would appear, thanks to permissionless.
- Requires miners to have access to tx market.

Mempool sync: through P2P nodes

DoS potential:

- Valid transaction creation is not rate limited by PoW.
- Transactions are larger and harder to validate than block headers.

Observation: transactions pay fee, and bid in block space market.

Mempool sync: fee prioritization

Use feerate to predict which transactions will make it into blocks, because miners are incentivized by fee.

- Needs to be accurate, otherwise attackers get free relay attack on the network (pay themselves with non-confirming transactions, free bandwidth!).
- This is hard.
- Transaction relay is altruism.

Mempool sync: what is the mempool

The set of transactions waiting to be mined

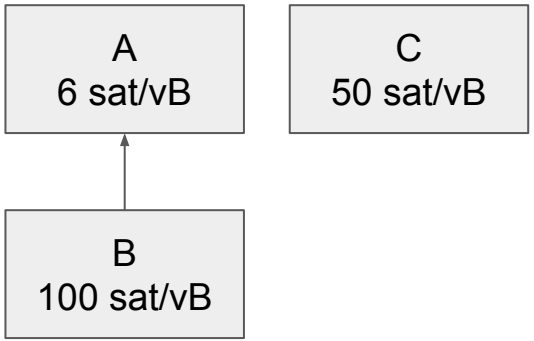
- No “the” mempool, each node has their own.
  - There cannot be consensus on mempool.
- Allows nodes to reason about what transactions will confirm, for validity (incl. dependencies) and fees (even harder with dependencies).

Mempool sync: complications

- For DoS reasons, mempool is limited in size.
  - Needs eviction algorithms if it grows too big.
  - Dynamic entry feerate.
- Conflicts:
  - What to do when there are conflicting transactions?
  - First-seen is not incentive compatible.
  - RBF rules

Mempool sync: complication: CPFP

Imagine a high-feerate child spending low-feerate parent



B can only be included if A is too.

We can treat them as being grouped together

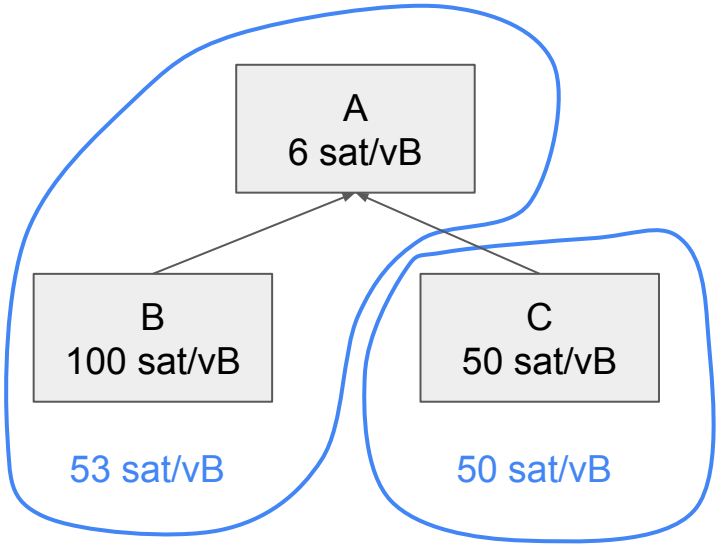
- Including A+B is better than C
- A has its effective feerate bumped to 53 sat/vB
- Relayed across network!

This enables child-pays-for-parent (CPFP)

- Recipient can bump priority of payment
- Hope that miners pick it up.

Ancestor set block building

Algorithm implemented since 2016



- For every tx, compute feerate of ancestors
- Include ancestor set with highest feerate
- Update ancestor feerates of remainder
- Repeat

In practice:

- Sufficient for CPFP
- Not for children-pay-for-parent (CnPFP)
- Ancestor sets are precomputed
- Necessitates ancestor set size limit

Mempool sync: complication: eviction

For DoS protection, size of mempool is limited.

When it grows too large, evict lowest feerate transactions. Mimics block building:

Block building:

- For every tx, keep ancestor feerate
- Include highest feerate ancestor set
- Update ancestor feerates of remainder
- Repeat while block not full
- Needs ancestor set limit

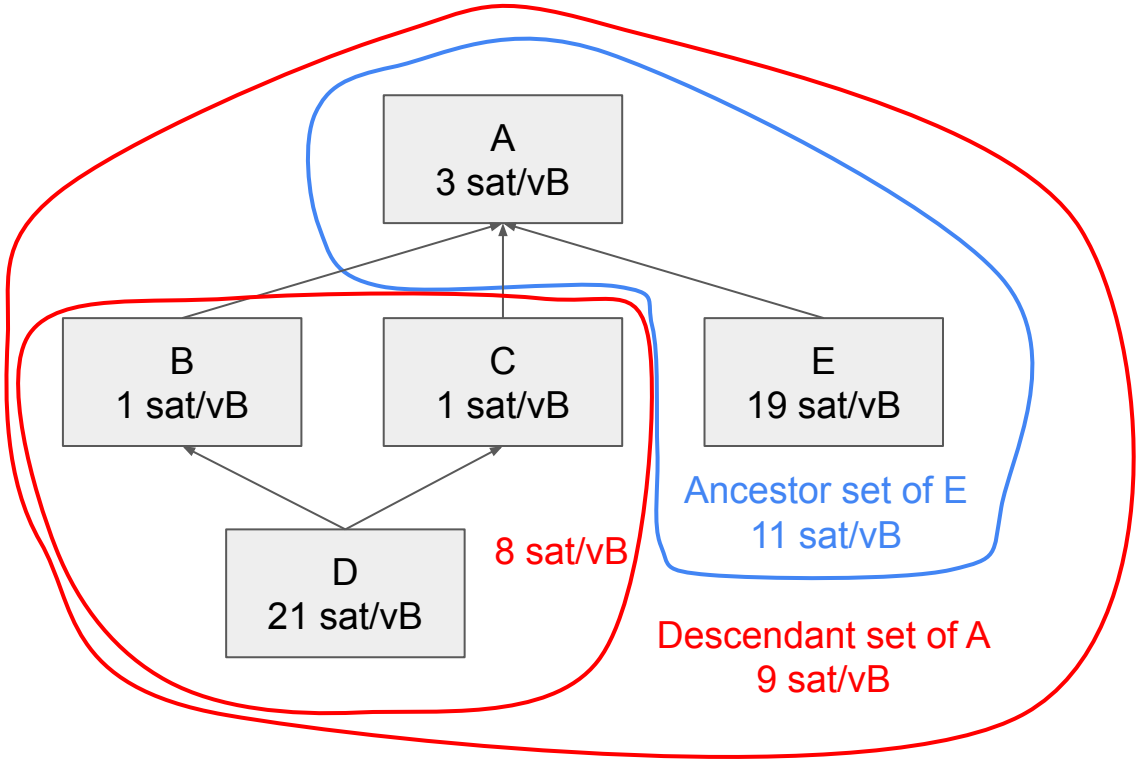
Eviction:

- Keep tx descendant feerates
- Evict lowest feerate set
- Update descendant feerates
- Repeat while too large
- Needs descendant set limit

Exactly the opposite of block building?

Mempool sync: eviction is broken!

Consider this example:



First eviction includes first thing to include!

Block building and eviction not opposites.

Just an example. It can get harder.

Mempool sync: lack of total ordering

Ideal eviction algorithm:

- Run block building algorithm on entire mempool
- Evict what the algorithm would include last
- Effective feerate = feerate of chunk a transaction is in

Sadly:

- Infeasible: ancestor set algorithm too slow for whole block
- Can’t run it continuously
- Ancestor set algorithm is not even optimal

Mempool sync: various heuristics

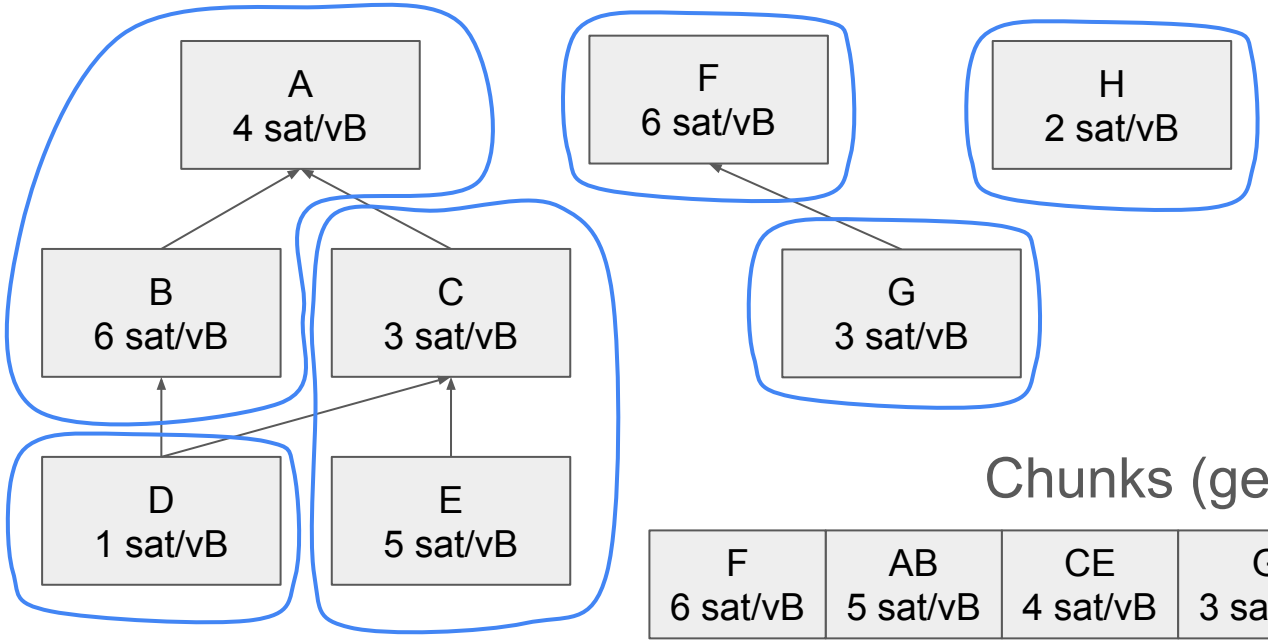
Eviction is not the only node decision aiming to guess tx desirability:

- **Fee estimation**: not just individual feerate
- **Replacement**: need to know if new is better
- **Flood protection**: determine what to send first

All of these use approximate heuristics, which are inconsistent, and in some cases, just wildly off.

Mempool sync: total ordering as solution

Given ordering: F, A, B, C, E, G, H, D (ignore how we got this)



Pick highest feerate prefixes

Chunks (generalize CPFP):

F	AB	CE	G	H	D
6 sat/vB	5 sat/vB	4 sat/vB	3 sat/vB	2 sat/vB	1 sat/vB

Total ordering as solution (cont)

Feerates:

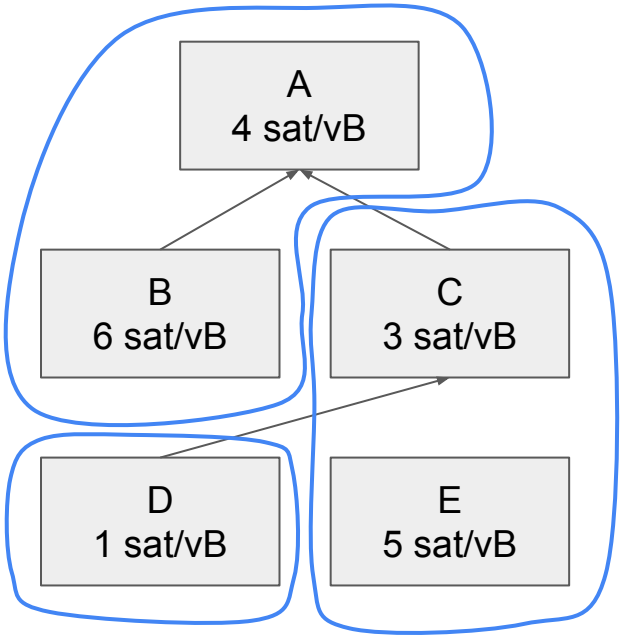
F 6 sat/vB	AB 5 sat/vB	CE 4 sat/vB	G 3 sat/vB	H 2 sat/vB	D 1 sat/vB

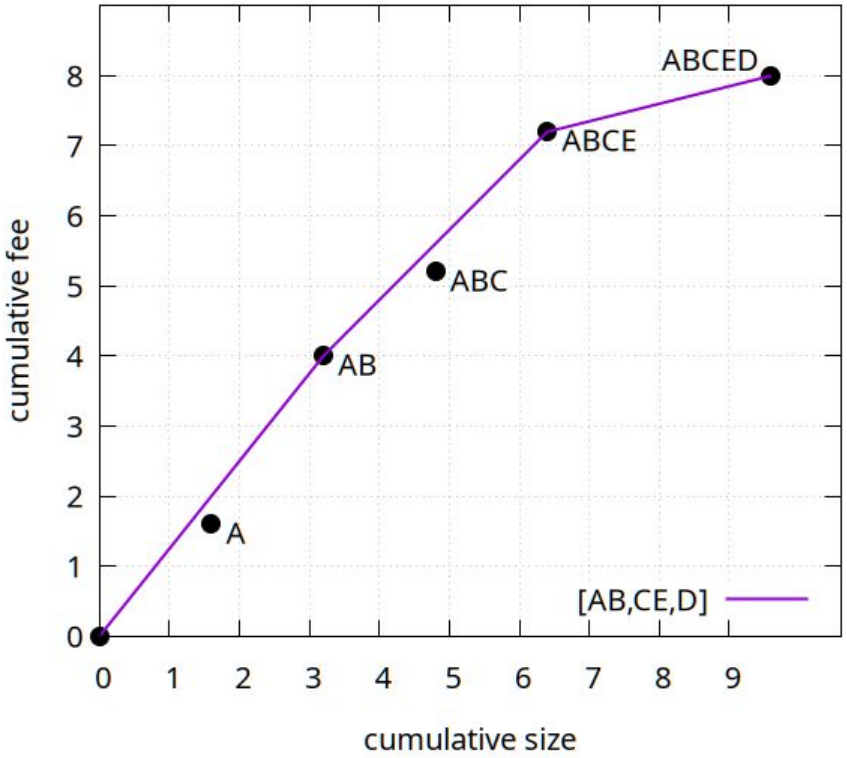
Block building: (*)

Eviction: 

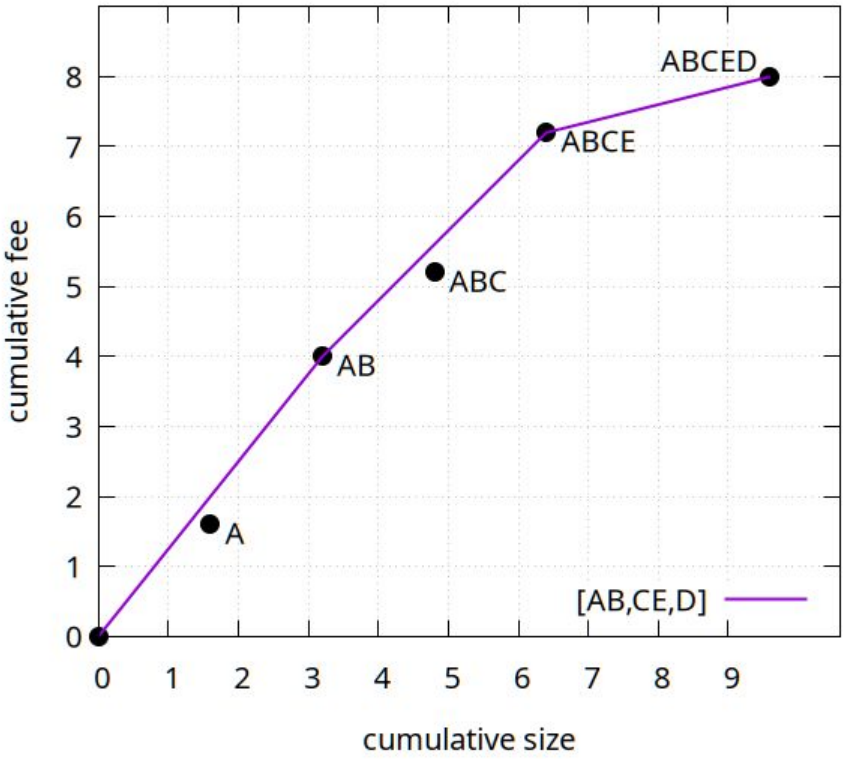
Replacement?

Fee-size diagrams for comparing transactions





Fee-size diagrams for comparison (cont)

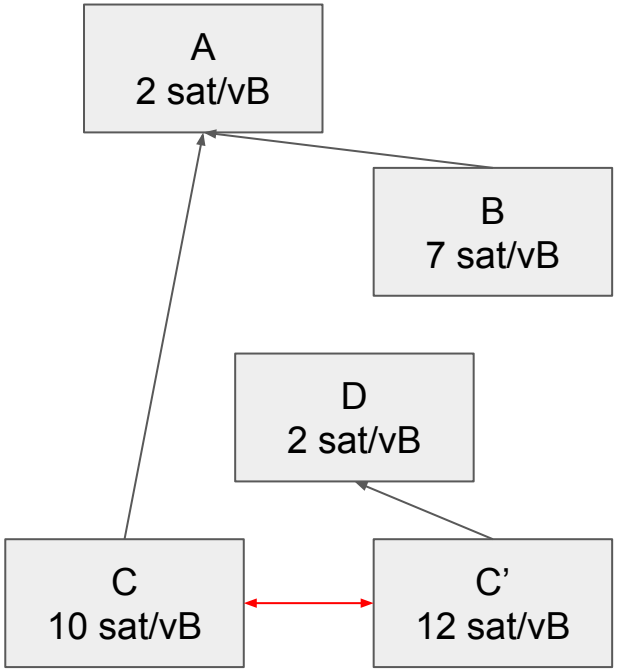


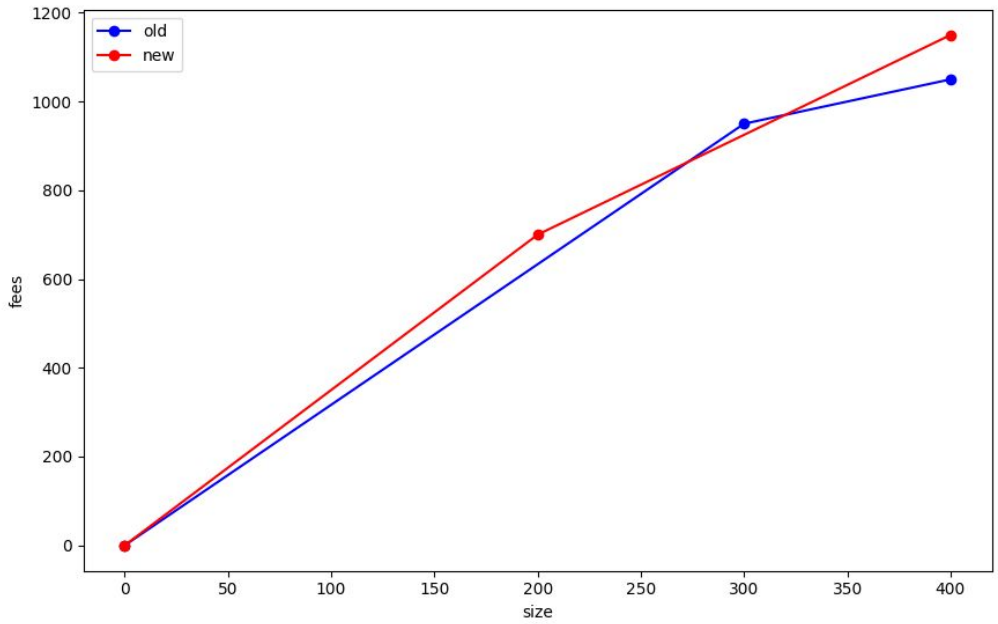
Lines on diagram are chunks

Approximate fee at size

Higher diagram = better, regardless of cutoff

Fee-size diagrams for evaluating replacements





Total ordering as solution (cont)

Feerates:
(estimation + flood)

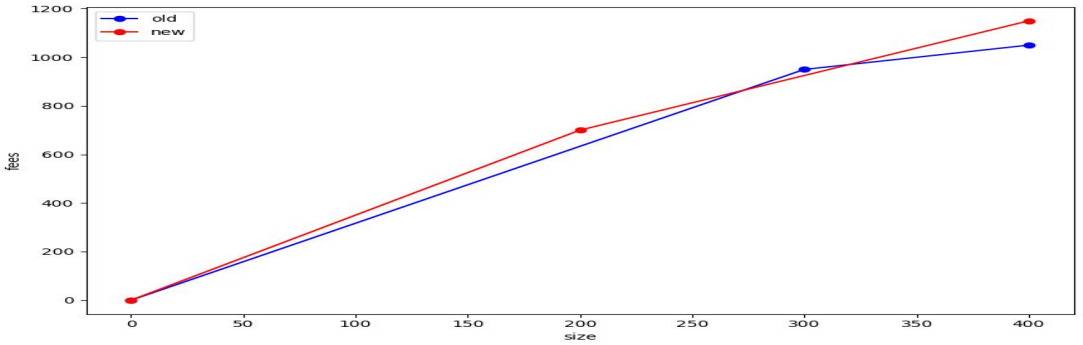
F 6 sat/vB	AB 5 sat/vB	CE 4 sat/vB	G 3 sat/vB	H 2 sat/vB	D 1 sat/vB

Block building: →

(*)

Eviction: ←

Replacement?



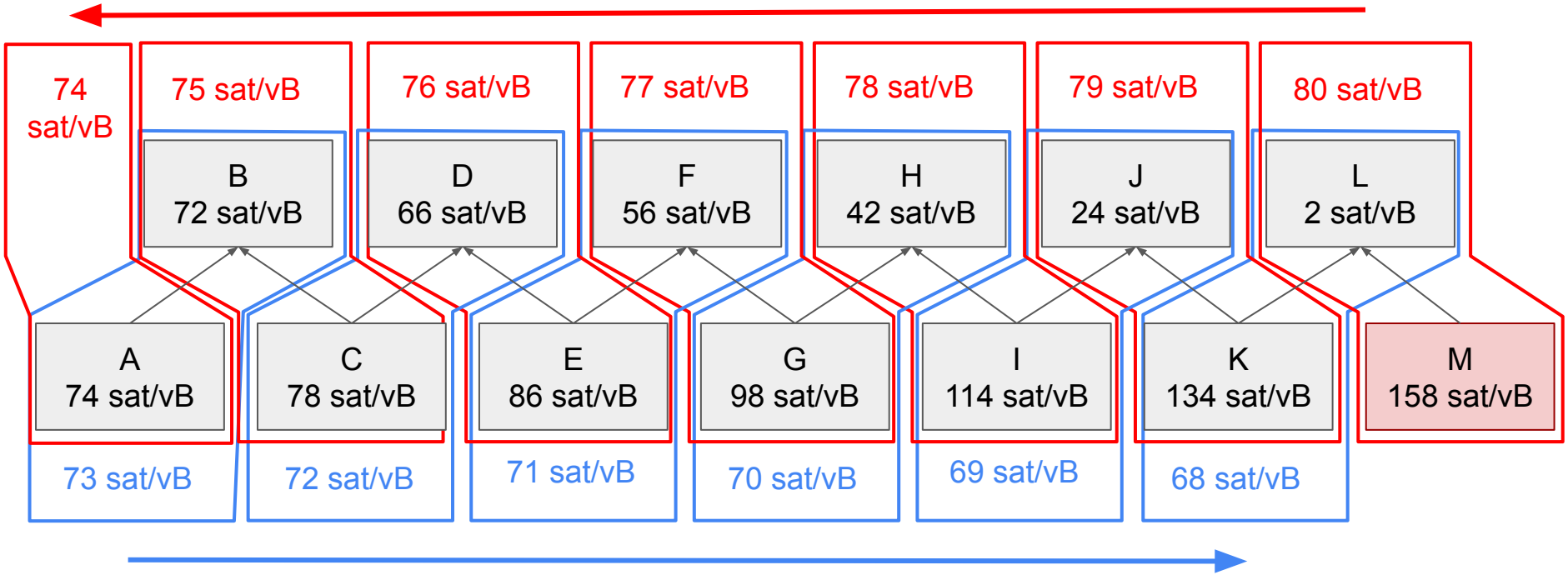
Making it feasible

Precomputation!

- We need effective feerates, but can’t recompute them all the time.
- Can we just recompute the ones that change?
- Depends on how many effective feerates can change…

How many effective feerates can change?

All connected transactions! “Reversing trellis” example:



Solution: limit cluster sizes

- Call connected components of mempool “clusters”
- Chunks do not cross cluster boundaries!
- Limit the number of transactions per cluster
  - Instead of ancestor/descendant limits
- Any time mempool changes, run mining algorithm on affected clusters and remember chunks/feerates
- At all times, chunks and effective feerates are known for all transactions

Cluster Mempool project

- Use clusters, chunks, total ordering, chunk feerates, …
- For block building, eviction, replacement, flood protection
- Ancestor/descendant limits (25) go away, replace with cluster count limit (64).
- RBF replacement rules are replaced with “mempool gets better” (diagram based).
  - In practice, people do repeated bumping anyway, not follow rules

Making it even better

With the block building algorithm just running on individual, small, clusters: can we do better than ancestor-set?

- Ancestor-set is enough for CPFP, but not for more complex.

I came up with an $\mathcal{O}(n \cdot \sqrt{2^n})$ **optimal** algorithm, much better than the naive $\mathcal{O}(2^n)$approach. So proud!

Until …

stefanwouldgo 1 Jan 29

Hi sipa, thanks for your great work on this.

> sipa:
>
> We don't have a proof that finding the optimal linearization (or finding the highest-feerate topologically-valid subset) is NP-hard, so it's possible a polynomial algorithm exist.

I've been thinking about this problem for some months now, alternating between looking for an algorithm and a reduction from some NP hard problem. It really is not obvious. However, DeepSeek R1 finally helped me find the surprising answer:

Finding a highest-feerate topologically-valid subset is possible in $O(nm \log(n^2/m))$ time, and this has been shown in 1989 by Gallo, Grigoriadis and Tarjan ("A FAST PARAMETRIC MAXIMUM FLOW ALGORITHM AND APPLICATIONS", SIAM J. COMPUT. Vol. 18, No. 1, pp. 30-55, February 1989, you can find it on sci-hub). Actually, there have been even earlier algorithms for this, quoted in this article, though they are a little slower. For your reference, the problem is called maximum-ratio closure problem (p. 48), the only difference being the direction of the arrows in the graph.

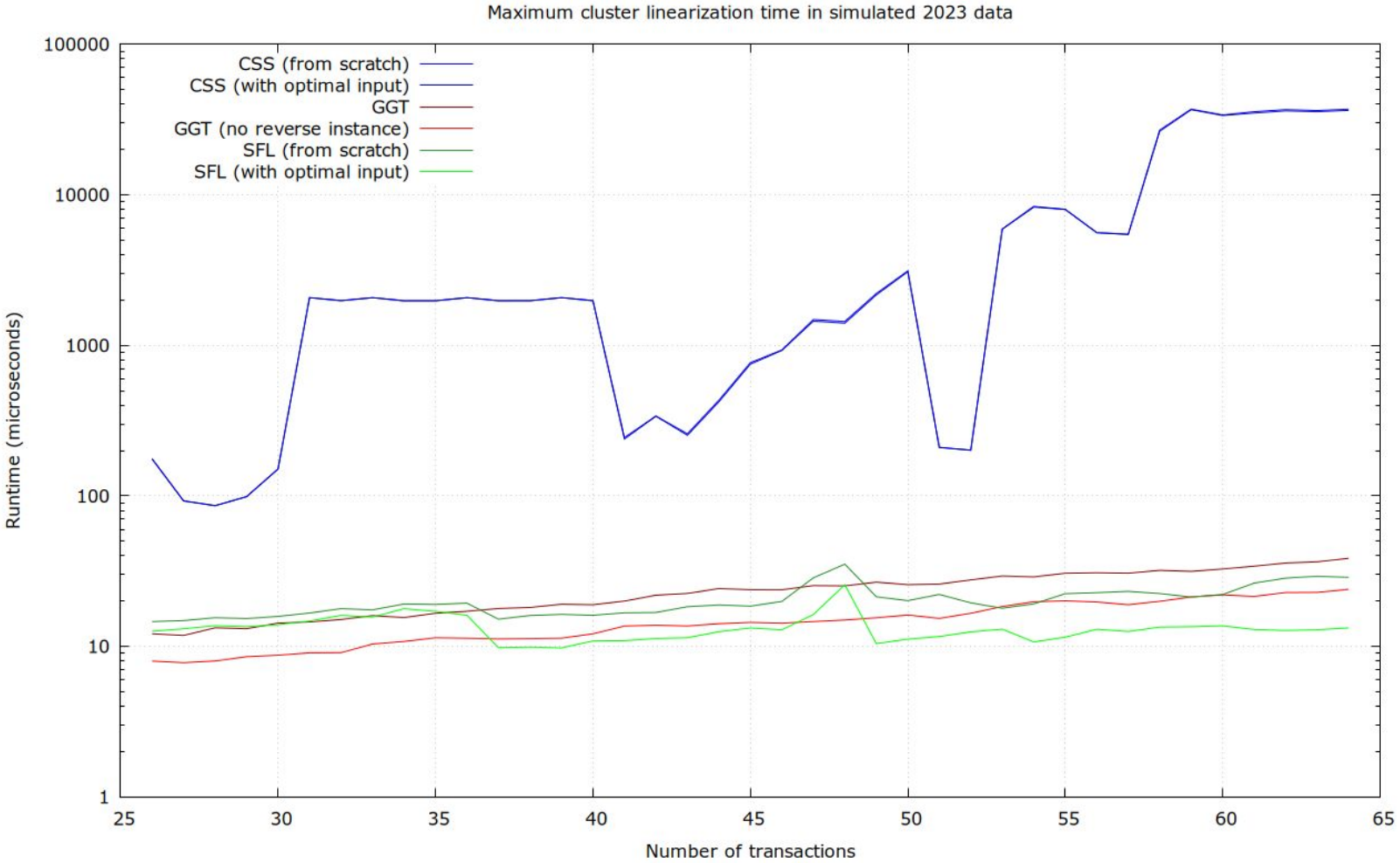
An O(n^3) algorithm was published in 1989…

and found by an LLM.

Actually thinking of using yet another algorithm (SFL).

- Based on simplex
- Load existing order and improve it.
- Make random improvements.
- Stop at any time.

Trait	CSS	SFL	GGT
Proven worst-case	$\mathcal{O}(n \cdot 2^n)$	$\mathcal{O}(\infty)$	$\mathcal{O}(n^3)$
Conjectured worst complexity	$\mathcal{O}(n \cdot \sqrt{2^n})$	$\mathcal{O}(n^5)$	$\mathcal{O}(n^3)$
Historical avg. runtime (µs) $(n \leq 64)$	80	10	22
Historical worst runtime (µs) $(n \leq 64)$	31500	26	33
Extrapolated worst runtime (µs) $(n \leq 64)$	700,000,000	1,000,000	10,000
Anytime algorithm	Needs budgeting	Natively	May lose progress
Improving existing	Through LIMO [1]	Natively	Merging afterwards
Fairness	Hard	Easy	Hard
Integer sizes	SF $(\times, <)$	$2SF$ $(\times, <, -)$	$4S^2F$ $(\times, /, <, +, -)$
Ancestor sort mix	Yes	No	No
Minimal chunks	Natively	No	No


Maximum cluster linearization time in simulated 2023 data
CSS (from scratch)
CSS (with optimal input)
GGT
GGT (no reverse instance)
SFL (from scratch)
SFL (with optimal input)
Runtime (microseconds)
100000
10000
1000
100
10
1
25
30
35
40
45
50
55
60
65
Number of transactions