Solution of hidden stabilizer circuits

qBitTensor Labs

It is our pleasure to announce the retirement of hidden stabilizer circuits. A number of innovative miners have come forward over the past couple of weeks in advance of this and gotten in touch with us to publish their solutions. Of these, we have selected the most innovative to describe here so that the wider community of miners on Subnet 63 may benefit from these ideas in the future. This document will describe the theory behind the solution and give a sample implementation provided by the miner.

1 Review

First, we provide a short review of the relevant concepts. Recall the problem definition:

Problem 1. (Hidden stabilizers)
Given an arbitrary unitary quantum circuit C on N qubits with guaranteed stabilizer state output $|\psi\rangle = C|0\rangle^{\otimes N}$, compute the N commuting stabilizers

$$\begin{aligned} S(|\psi\rangle) &\equiv \{S_1, ..., S_N\} \subset \mathcal{P}_N, \\ [S_i, S_j] &\equiv S_i S_j - S_j S_i = 0 \end{aligned} \tag{1}$$

that generate $\mathrm{Stab}(|\psi\rangle)$ in canonical form.

Here, we ask for the N commuting stabilizers that characterize the output stabilizer state $|\psi\rangle$. Each stabilizer is of the form of a so-called "Pauli string" – that is, $\mathcal{P}_N$ is the set of all signed N-fold tensor products of single-qubit Pauli matrices $\sigma \in \mathcal{P} = \{I, X, Y, Z\}$

$$\mathcal{P}_N \equiv \left\{ \pm \bigotimes_{k=1}^{N} \sigma \;\middle|\; \sigma \in \mathcal{P} \right\}. \tag{2}$$

For thoroughness, the single-qubit Pauli matrices are

$$\sigma_0 \equiv I \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \sigma_1 \equiv X \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_2 \equiv Y \equiv \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma_3 \equiv Z \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{3}$$

obeying $\sigma_k^2 = -i\sigma_1\sigma_2\sigma_3 = \sigma_0$ and $\sigma_i\sigma_j = \delta_{ij}\sigma_0 + i\varepsilon_{ijk}\sigma_k$ with i, j, and k ranging over $1, ..., 3$. Another important concept here is the *commutator*, defined as

$$[A, B] = AB - BA. \tag{4}$$

Commonly, we say that two objects A and B *commute* if their commutator is zero; for our Pauli matrices, we have

$$[\sigma_i, \sigma_j] = 2i\varepsilon_{ijk}\sigma_k, \tag{5}$$

meaning that any given pair of single-qubit Paulis either commutes and is self-inverse (when $i = j$), or anti-commutes (otherwise). When it comes to Pauli strings (that is, elements of $\mathcal{P}_N$), multiplication

then occurs element-wise according to these rules; hence every pair of elements of $\mathcal{P}_N$ likewise either commutes or anti-commutes, although some extra considerations need to be made to account for the fact that two strings with non-commuting components may still commute if the number of such elements is even.

Meanwhile, we also have the N-qubit Clifford group $\mathcal{C}_N$. Although it is typically thought of in terms of its generators, i.e. the gates H, S, and CNOT, it is principally defined via the property that, for all $c \in \mathcal{C}_N$, conjugation of any Pauli string P results in another Pauli string P':

$$P' = CPC^\dagger \in \mathcal{P}_N \quad \forall P \in \mathcal{P}_N, C \in \mathcal{C}_N. \tag{6}$$

This property is the main working principle behind the use of stabilizer tableaus as circuit simulators. Given an $N \times (N+1)$-bit representation of a set of (stabilizer) Pauli strings, conjugation of each string with any Clifford gate produces the stabilizer of the corresponding output state of the gate. In other terms, if we are given P as a stabilizer of $|\psi\rangle$, then we have

$$\langle\psi|P|\psi\rangle = 1. \tag{7}$$

Given some $C \in \mathcal{C}_N$, we then want to compute a stabilizer of $|\psi'\rangle \equiv C|\psi\rangle$, and can do so by preserving Equation 7 as

$$\langle\psi|C^\dagger CPC^\dagger C|\psi\rangle = \big(\langle\psi|C^\dagger\big)\big(CPC^\dagger\big)\big(C|\psi\rangle\big) = \langle\psi'|P'|\psi'\rangle = 1, \quad P' \equiv CPC^\dagger. \tag{8}$$

In the tableau representation, this operation can be performed efficiently by twiddling bits – Clifford circuits are guaranteed to take Pauli strings to Pauli strings, and we are guaranteed that the output Pauli string is representable as a tableau.

The challenge of the hidden stabilizers problem lies principally in the fact that tableaus are unable to represent anything other than pure Pauli strings. If we wish to conjugate a row of the tableau with anything other than a Clifford element – an R_X gate with rotation angle not equal to a multiple of $\pi/2$, for instance – then we're out of luck!

Thus a naive approach to the problem will simulate the given obfuscated circuit through some other means, and then find a set of stabilizers by searching through $\mathcal{P}$ for elements that give a $+1$ expectation value through the output state. This is obviously subject to an exponential bound, given $|\mathcal{P}_N| = 2 \times 4^N$. Although the number of elements whose expectation values need to be computed can be reduced by eliminating overall signs (if $\langle\psi|P|\psi\rangle = -1$, then we know $\langle\psi|(-P)|\psi\rangle = 1$) and narrowing the search to only strings that commute with stabilizers found earlier in the search, the naive approach is still infeasible in practice.

We've seen a number of miners mention solutions to convert the obfuscated circuit back to pure Clifford form. However, the following solution takes an alternative approach based on recent research that adapts a different technique known as *Pauli propagation*, with no need to pre-process the circuit.

2 Pauli propagation

In our communications, the miner whose solution we discuss below referenced two recent papers on arXiv [1,2] as invaluable resources. As a problem-solving technique, Pauli propagation is typically discussed in the context of problems that require the expectation values of specific operators to be evaluated. Generically, this sort of problem would be to, given some circuit C, compute the circuit's output state $|\psi\rangle \equiv C|0\rangle^{\otimes N}$ and then return $\langle\psi|A|\psi\rangle$ for some observable A.

Pauli propagation arises from a three key observations. First, the set of all pure (i.e. *unsigned*) Pauli strings $\overline{\mathcal{P}}_N$ forms a complete basis for all $2^N \times 2^N$ Hermitian matrices via linear combinations with real coefficients. That is, any observable A can be decomposed into a sum over Pauli strings,

$$A = \sum_{P \in \overline{\mathcal{P}}_N} a_P P, \quad a_P \in \mathbb{R}, \tag{9}$$

which means that we can take A, apply any Hermiticity-preserving transformation to A – say, f – and arrive at another sum over Pauli strings.

$$f(A) = \sum_{P \in \overline{\mathcal{P}}_N} a'_P P. \tag{10}$$

Note that for f taken as conjugation by Clifford operators, the tableau simulator evolution rules are an example of this kind of transformation.

Second, the transformation of Pauli strings under conjugation by Pauli (string) rotation gates has a nice closed form. We can see this easily by looking at the power series expansion of $R_P(\theta) = \exp(-iP\theta/2)$, considering that each $P^2 = I^{\otimes N}$:

$$\begin{aligned} e^{-iP\frac{\theta}{2}} &= \sum_{j=0}^{\infty} \frac{(-iP\theta/2)^j}{j!} \\ &= \left(\sum_{j=0}^{\infty} (-i)^{2j} \frac{(\theta/2)^{2j}}{(2j)!} (P^2)^j\right) + \left(-i \sum_{j=0}^{\infty} (-i)^{2j} \frac{(\theta/2)^{2j+1}}{(2j+1)!} (P^2)^j P\right) \\ &= I^{\otimes N} \left(\sum_{j=0}^{\infty} (-1)^j \frac{(\theta/2)^{2j}}{(2j)!}\right) - iP \left(\sum_{j=0}^{\infty} (-1)^j \frac{(\theta/2)^{2j+1}}{(2j+1)!}\right) \\ &= \cos\frac{\theta}{2} I^{\otimes N} - i \sin\frac{\theta}{2} P \end{aligned} \tag{11}$$

(which should be recognizable as a matrix version of Euler's formula). Hence, the conjugation of some other Pauli string Q by $R_P(\theta)$ amounts to

$$\begin{aligned} e^{iP\frac{\theta}{2}} Q e^{-iP\frac{\theta}{2}} &= \left(\cos\frac{\theta}{2} I^{\otimes N} + i \sin\frac{\theta}{2}\ P\right) Q \left(\cos\frac{\theta}{2} I^{\otimes N} - i \sin\frac{\theta}{2}\ P\right) \\ &= \cos^2\frac{\theta}{2}\ Q - i \cos\frac{\theta}{2} \sin\frac{\theta}{2}\ QP + i \sin\frac{\theta}{2} \cos\frac{\theta}{2}\ PQ + \sin^2\frac{\theta}{2}\ PQP. \end{aligned} \tag{12}$$

There are two cases here. If $[P, Q] = 0$, then the middle terms cancel and we are left with

$$\cos^2\frac{\theta}{2}\ Q + \sin^2\frac{\theta}{2}\ Q = Q. \tag{13}$$

Otherwise the PQP term can be rearranged with a minus sign and we have

$$\left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right) Q - 2i \cos\frac{\theta}{2} \sin\frac{\theta}{2}\ QP = \cos\theta\ Q - i \sin\theta\ QP, \tag{14}$$

which provides an easy way to compute the conjugation of Pauli strings with a very large class of non-Clifford operators.

Third, it is computationally very easy to evaluate the expectation value of any Pauli string with respect to the all-zero state $|0\rangle^{\otimes N}$: If the string contains any X's or Y's the expectation value is 0, otherwise it is $+1$.

Thus for a problem asking for the expectation value of some A via the output of some circuit C,

$$\langle A \rangle = \langle 0|^{\otimes N} C^\dagger A C |0\rangle^{\otimes N} \tag{15}$$

the Pauli propagation approach is to evolve not the state $|0\rangle^{\otimes N} \to |\psi\rangle = C|0\rangle^{\otimes N}$ forward before computing $\langle\psi|A|\psi\rangle$, but the observable $A \to A' = C^\dagger A C$ backward before computing $\langle 0|^{\otimes N} A' |0\rangle^{\otimes N}$. This is exactly the difference between the Schrödinger picture (i.e. state-oriented) and the Heisenberg picture (i.e. operator-oriented).

One notable concern to performing Pauli propagation with general circuits is that, naively by Equation 14, the number of terms in Equation 9 with non-zero weight will, in principle, approximately double with each non-Clifford gate in the circuit. While it is true that the number of terms in the sum can increase to dramatic scales in random circuits, there is often a significant amount of cancellation that can occur, and in practice it is useful to maintain a cutoff ε to truncate low-weight terms in order to keep memory requirements manageable.

3 Application to hidden stabilizers

In the solution of hidden stabilizers, we work with Equation 8 rather than Equation 15, which has the conjugation of the given circuit C flipped: we wish to compute $CPC^\dagger$ instead of $C^\dagger PC$ for some Pauli string P. Letting $C = U_n \cdots U_2 U_1$, we can see the difference more clearly:

$$\begin{aligned} C^\dagger PC &= U_1^\dagger U_2^\dagger \cdots U_n^\dagger \; P \; U_n \cdots U_2 U_1 \\ CPC^\dagger &= U_n \cdots U_2 U_1 \; P \; U_1^\dagger U_2^\dagger \cdots U_n^\dagger \end{aligned} \tag{16}$$

In the standard Heisenberg picture evolution (top), we propagate P through the circuit by folding gates U_k into P in *reverse* order, while in the hidden stabilizers solution (bottom), we fold them in the *normal*-time order, but work with the adjoints of gates instead. When the end of the circuit is reached, we don't actually compute the expectation value as in ordinary Pauli propagation, but instead end up with some other Pauli string P' that is a stabilizer of the output state of C. With this in mind, the steps to solve a hidden stabilizers problem with Pauli propagation are as follows.

1. Take the input circuit as C and parse into a sequence of gates;
2. Initialize a set of single-term Pauli sums to the stabilizer generators for the $|0\rangle^{\otimes N}$ state:
$$S(|0\rangle^{\otimes N}) = \{ZII \cdots I, IZI \cdots I, IIZ \cdots I, ..., III \cdots Z\}; \tag{17}$$
3. For each $s \in S(|0\rangle^{\otimes N})$, do:
    1. Compute $CsC^\dagger$ by stepping through C via $s \to UsU^\dagger$ for each U in the time-ordered decomposition of C according to the rules of Pauli propagation;
    2. Take the Pauli string in the final sum that has the weight with largest absolute value, rounding the weight to ± 1;
4. Canonicalize the final set of Pauli strings and return.

4 Sample code

The rest of this document will reproduce a sample implementation of the ideas described above as it pertains to solution of hidden stabilizers and was provided to the qBitTensor Labs team. The samples are in the language Haskell and depend only on the well-known packages `containers`, `vector`, and `text`.

```haskell
{-# LANGUAGE DuplicateRecordFields, NamedFieldPuns, OverloadedRecordDot #-}
import Data.Bits ((.&.), xor, bit, clearBit, setBit, popCount)
import Data.Function ((&))
import Data.List (foldl', maximumBy, zipWith4)
import qualified Data.Map as Map
import Data.Map (Map)
import Data.Maybe (fromMaybe, mapMaybe)
import qualified Data.Vector as Vec
import Data.Vector (Vector, (!), (//))
import Data.Word (Word32)
import Text.Printf (printf)
```

The miner starts by defining a basic abstract representation of the single-qubit Pauli matrices.

```haskell
-- A single 1-qubit Pauli operator.
data Pauli = I | X | Y | Z deriving (Eq, Ord)

instance Show Pauli where
  show I = "_"
  show X = "X"
  show Y = "Y"
  show Z = "Z"
```

Although Pauli strings can be represented as simple `[Pauli]`s, it is more efficient to use the symplectic (i.e. tableau-like) representation, with single bits being packed into 32-bit integers.

```haskell
-- The tensor product of a number of single-qubit Paulis. Will be implicitly
-- right-padded with identities as needed.
data PauliString = PauliString
  { len :: Int
  , x :: Vector Word32
  , z :: Vector Word32
  }

-- Create the identity on a number of qubits.
ident :: Int → PauliString
ident nqubits = PauliString { len = nqubits, x, z }
  where len = max 1 $ nqubits `quot` 32
        x = Vec.replicate len 0
        z = Vec.replicate len 0
```

Working in the symplectic representation requires some simple boilerplate to convert between it and more human-readable data.

```haskell
-- Get the (word, bit) coordinates of a given single Pauli position.
coord :: Int → (Int, Int)
coord k = (k `quot` 32, k `rem` 32)

-- Decode the symplectic representation for a single Pauli.
decode :: Word32 → Word32 → Pauli
decode 0 0 = I
decode _ 0 = X
decode 0 _ = Z
decode _ _ = Y

-- Get a single given bit.
get' :: Int → Vector Word32 → Word32
get' k v = (v ! k5) .&. bit k'
  where (k5, k') = coord k

-- Get the single-qubit Pauli in a given position.
get :: Int → PauliString → Maybe Pauli
get k paulis = if k < 0 || k ≥ paulis.len then Nothing else Just $ decode xk zk
  where (xk, zk) = (get' k paulis.x, get' k paulis.z)
```

```haskell
-- Set a single given bit.
set' :: Int → Bool → Vector Word32 → Vector Word32
set' k b v = v // [(k5, word')]
  where (k5, k') = coord k
        word' = if b then setBit (v ! k5) k' else clearBit (v ! k5) k'

-- Right-pad with as many words as it takes to have at least a number of bits.
rpad :: Int → Vector Word32 → Vector Word32
rpad len v = if wordpad > 0 then v Vec.++ Vec.replicate wordpad 0 else v
  where wordpad = (max 1 $ len `quot` 32) - Vec.length v

-- Set the single-qubit Pauli in a given position.
set :: Int → Pauli → PauliString → PauliString
set k p paulis = paulis { len = max paulis.len (k + 1), x = x', z = z' }
  where (x, z) = (rpad (k + 1) paulis.x, rpad (k + 1) paulis.z)
        (x', z') = case p of
                     I → (set' k False x, set' k False z)
                     X → (set' k True  x, set' k False z)
                     Y → (set' k True  x, set' k True  z)
                     Z → (set' k False x, set' k True  z)
```

```haskell
instance Show PauliString where
  show paulis = concatMap show singles
    where unwrap (Just x) = x
          unwrap Nothing  = error "unreachable"
          singles = map (unwrap . (`get` paulis)) [0 .. paulis.len - 1]

zipLongest :: [a] → [b] → [(Maybe a, Maybe b)]
zipLongest (ha : ta) (hb : tb) = (Just ha, Just hb) : zipLongest ta tb
zipLongest (ha : ta) []        = (Just ha, Nothing) : zipLongest ta []
zipLongest []        (hb : tb) = (Nothing, Just hb) : zipLongest [] tb
zipLongest []        []        = []

pauliMultiEq :: (Maybe Word32, Maybe Word32) → Bool
pauliMultiEq (l, r) = fromMaybe 0 l == fromMaybe 0 r

instance Eq PauliString where
  l == r =
    (all pauliMultiEq $ zipLongest (Vec.toList l.x) (Vec.toList r.x))
    && (all pauliMultiEq $ zipLongest (Vec.toList l.z) (Vec.toList r.z))

-- required for use with Map later
instance Ord PauliString where
  compare l r = if l == r then EQ else compare l.x r.x <> compare l.z r.z
```

Now the relevant algebraic and commutator properties of Pauli strings can be defined.

```haskell
-- Return True if two Pauli strings commute
commutes :: PauliString → PauliString → Bool
commutes p q = even $ Vec.sum commDotCount
  where dotCount x0 z0 x1 z1 = popCount $ (x0 .&. z1) `xor` (x1 .&. z0)
        commDotCount = Vec.zipWith4 dotCount p.x p.z q.x q.z
```

Multiplication of `PauliStrings` requires keeping track of an accumulated phase from individual product of single-qubit Paulis.

```haskell
-- Powers of i as ordinary integers: 0 ⇒ 1, 1 ⇒ i, 2 ⇒ -1, 3 ⇒ -i, etc.
type Phase = Int

-- Unpack all bits of a single word.
toBits :: Word32 → [Word32]
toBits word = map (\k → word .&. bit k) [0 .. 31]

-- Compute the "local" phase from multiplying two single-qubit Paulis.
localPhase :: Word32 → Word32 → Word32 → Word32 → Phase
localPhase x0k z0k x1k z1k
  | x0k /= 0 && z0k == 0 && x1k /= 0 && z1k /= 0 = 1
  | x0k /= 0 && z0k == 0 && x1k == 0 && z1k /= 0 = -1
  | x0k /= 0 && z0k /= 0 && x1k == 0 && z1k /= 0 = 1
  | x0k /= 0 && z0k /= 0 && x1k /= 0 && z1k == 0 = -1
  | x0k == 0 && z0k /= 0 && x1k /= 0 && z1k == 0 = 1
  | x0k == 0 && z0k /= 0 && x1k /= 0 && z1k /= 0 = -1
  | otherwise = 0

-- Compute the total phase from multiplying two multi-qubit Paulis.
wordPhase :: Word32 → Word32 → Word32 → Word32 → Phase
wordPhase x0 z0 x1 z1 = sum $ zipWith4 localPhase x0' z0' x1' z1'
  where (x0', z0', x1', z1') = (toBits x0, toBits z0, toBits x1, toBits z1)
```

```haskell
-- Modular reduction, constrained to positive values
remPos :: Integral a => a -> a -> a
remPos n m = if n' < 0 then n' + m else n'
  where n' = n `rem` m

-- Compute the total phase from multiplying two Pauli strings.
mulPhase :: PauliString -> PauliString -> Phase
mulPhase p q = r `remPos` 4
  where r = Vec.sum $ Vec.zipWith4 wordPhase p.x p.z q.x q.z

-- Compute the operator-only portion of the product of two Pauli strings.
mulOp :: PauliString -> PauliString -> PauliString
mulOp p q = PauliString { len, x, z }
  where len = max p.len q.len
        rpad' = rpad len
        (xp, zp, xq, zq) = (rpad' p.x, rpad' p.z, rpad' q.x, rpad' q.z)
        (x, z) = (Vec.zipWith xor xp xq, Vec.zipWith xor zp zq)

-- Compute the rescaled multiplication of two Pauli strings p and q, -i p q.
-- Returns Nothing if p and q commute.
(@) :: PauliString -> PauliString -> Maybe (Bool, PauliString)
p @ q = if p `commutes` q then Nothing else Just (sign, op)
  where r = mulPhase p q
        sign = odd $ r `quot` 2
        op = mulOp p q
```

Hence, sums over Pauli strings can be modeled.

```haskell
-- Pauli propagation simulator.
data PauliSum = PauliSum
  { terms :: Map PauliString Double
  , eps :: Double -- truncation threshold
  }

instance Show PauliSum where
  show paulis = concatMap showPair $ Map.toList paulis.terms
    where showPair (term, ampl) = printf "%s %f\n" (show term) ampl :: String

-- Create a new sum over a single Pauli string. The truncation threshold
-- defaults to -1, which disables truncation.
single :: PauliString -> Double -> Maybe Double -> PauliSum
single paulis ampl mbTrunc = PauliSum { terms, eps }
  where terms = Map.insert paulis ampl $ Map.empty
        eps = fromMaybe (-1.0) mbTrunc

-- Add a new term to the sum.
addTerm :: PauliString -> Double -> PauliSum -> PauliSum
addTerm term ampl paulis = paulis { terms = terms' }
  where terms' = Map.alter work term paulis.terms
        work Nothing = if abs ampl ≥ paulis.eps then Just ampl else Nothing
        work (Just prev) = if abs new ≥ paulis.eps then Just new else Nothing
          where new = prev + ampl

-- Get the Pauli string (and weight) whose weight has maximal absolute value.
maxWeight :: PauliSum -> Maybe PauliTerm
maxWeight paulis = if length paulis.terms > 0 then Just maxTerm else Nothing
  where maxTerm = maximumBy cmp $ Map.toList paulis.terms
        cmp (_, al) (_, ar) = compare al ar
```

Along with the all-important function to conjugate the sum by Pauli rotation gates.

```haskell
-- Conjugate a sum with a Pauli rotation gate about a given axis.
applyRot :: Double → [(Int, Pauli)] → PauliSum → PauliSum
applyRot _ [] paulis = paulis
applyRot angle ax paulis = removeZeros paulis''
  where nqubits = 1 + (maximum $ map fst ax)
        axString = foldr (\(k, p) acc → set k p acc) (ident nqubits) ax
        (newTerms, terms') =
          Map.mapAccumWithKey (applyRotWork angle axString) [] paulis.terms
        paulis' = paulis { terms = terms' }
        paulis'' = foldr (\(t, a) ts → addTerm t a ts) paulis' newTerms

applyRotWork
  :: Double                  -- angle
  → PauliString              -- rotation axis
  → [(PauliString, Double)] -- accumulator for new terms to add
  → PauliString              -- original term
  → Double                   -- amplitude
  → ([(PauliString, Double)], Double)
applyRotWork angle ax newTerms term ampl =
  case term @ ax of
    Nothing → (newTerms, ampl)
    Just (sign, newTerm) → (newTerms', ampl')
      where sign' = if sign then -1.0 else 1.0
            newAmpl = sign' * sin (-angle) * ampl -- negate the angle!
            newTerms' = (newTerm, newAmpl) : newTerms
            ampl' = cos angle * ampl
```

Note that, here, a minus sign is applied to the rotation angle to account for the fact that, per Equation 16, the Pauli strings must be conjugated by the *adjoints* of gates. Another essential ingredient is the ability to conjugate with H, S, $S^\dagger$, and CNOT gates, which can be taken directly from tableau simulation (see here for reference).

```haskell
type PauliTerm = (PauliString, Double)

termWise :: (PauliTerm → PauliTerm) → PauliSum → PauliSum
termWise f paulis = paulis { terms = terms' }
  where terms' = Map.fromList $ map f $ Map.toList paulis.terms

-- Flip a single given bit, right-padding with zeros as needed.
flip' :: Int → Vector Word32 → Vector Word32
flip' k v = v' // [(k5, (v' ! k5) `xor` bit k')]
  where (k5, k') = coord k
        v' = rpad (k + 1) v
```

```haskell
-- Conjugate a sum with a Hadamard gate.
applyH :: Int → PauliSum → PauliSum
applyH k paulis = termWise (applyHWork k) paulis

applyHWork :: Int → PauliTerm → PauliTerm
applyHWork k (term, ampl) =
  if xk == 0 && zk == 0 then (term, ampl) else (term', ampl')
    where (x, z) = (rpad (k + 1) term.x, rpad (k + 1) term.z)
          (xk, zk) = (get' k x, get' k z)
          x' = if (xk `xor` zk) /= 0 then flip' k x else x
          z' = if (xk `xor` zk) /= 0 then flip' k z else z
          term' = term { x = x', z = z' }
          ampl' = if xk /= 0 && zk /= 0 then -ampl else ampl
```

```haskell
-- Conjugate a sum with an S gate.
applyS :: Int → PauliSum → PauliSum
applyS k paulis = termWise (applySWork k) paulis

applySWork :: Int → PauliTerm → PauliTerm
applySWork k (term, ampl) = if xk == 0 then (term, ampl) else (term', ampl')
  where (x, z) = (rpad (k + 1) term.x, rpad (k + 1) term.z)
        (xk, zk) = (get' k x, get' k z)
        z' = if xk /= 0 then flip' k z else z
        term' = term { x, z = z' }
        ampl' = if xk /= 0 && zk /= 0 then -ampl else ampl
```

```haskell
-- Conjugate a sum with an Sdg gate.
applySdg :: Int → PauliSum → PauliSum
applySdg k paulis = termWise (applySdgWork k) paulis

applySdgWork :: Int → PauliTerm → PauliTerm
applySdgWork k (term, ampl) = if xk == 0 then (term, ampl) else (term', ampl')
  where (x, z) = (rpad (k + 1) term.x, rpad (k + 1) term.z)
        (xk, zk) = (get' k x, get' k z)
        z' = if xk /= 0 then flip' k z else z
        term' = term { x, z = z' }
        ampl' = if xk /= 0 && zk == 0 then negate ampl else ampl
```

```haskell
-- Conjugate a sum with a CX/CNOT gate. The control is the left qubit index.
applyCX :: Int → Int → PauliSum → PauliSum
applyCX c t paulis = termWise (applyCXWork c t) paulis

applyCXWork :: Int → Int → PauliTerm → PauliTerm
applyCXWork c t (term, ampl) =
  if c ≥ term.len then (term, ampl) else (term', ampl')
    where (x, z) = (rpad (max c t + 1) term.x, rpad (max c t + 1) term.z)
          (xc, zt) = (get' c x, get' t z)
          x' = if xc /= 0 then flip' t x else x
          z' = if zt /= 0 then flip' c z else z
          term' = term { x = x', z = z' }
          (xc', zc', xt', zt') = (get' c x', get' c z', get' t x', get' t z')
          ampl' = if xc' /= 0 && zt' /= 0 && not ((xt' /= 0) `xor` (zc' /= 0))
                  then -ampl else ampl
```

Which then allows controlled single-qubit rotations to be implemented.

```haskell
-- Conjugate a sum with a controlled RX gate.
applyCRX :: Double → Int → Int → PauliSum → PauliSum
applyCRX angle c t paulis =
  paulis & applyH t
         & applyCRZ angle c t
         & applyH t

-- Conjugate a sum with a controlled RY gate.
applyCRY :: Double → Int → Int → PauliSum → PauliSum
applyCRY angle c t paulis =
  paulis & applySdg t & applyH t
         & applyCRZ angle c t
         & applyH t & applyS t

-- Conjugate a sum with a controlled RZ gate.
applyCRZ :: Double → Int → Int → PauliSum → PauliSum
applyCRZ angle c t paulis =
  paulis & applyRot (angle / 2.0) [(t, Z)] & applyCX c t
         & applyRot (-angle / 2.0) [(t, Z)] & applyCX c t
```

Then, a simple gate interface can be constructed,

```haskell
-- A single gate in a circuit.
data Gate
  = H Int
  | S Int
  | Sdg Int
  | CX Int Int
  | RX Double Int
  | RY Double Int
  | RZ Double Int
  | CRX Double Int Int
  | CRY Double Int Int
  | CRZ Double Int Int
  deriving (Eq, Show)

-- Parse a QASM string into a number of qubits and a list of gates.
parseQasm :: String -> (Int, [Gate])
parseQasm = error "not implemented" -- omitted for brevity

-- Conjugate a sum with a generic Gate.
applyGate :: Gate -> PauliSum -> PauliSum
applyGate (H k) = applyH k
applyGate (S k) = applyS k
applyGate (Sdg k) = applySdg k
applyGate (CX c t) = applyCX c t
applyGate (RX ang k) = applyRot ang [(k, X)]
applyGate (RY ang k) = applyRot ang [(k, Y)]
applyGate (RZ ang k) = applyRot ang [(k, Z)]
applyGate (CRX ang c t) = applyCRZ ang c t
applyGate (CRY ang c t) = applyCRY ang c t
applyGate (CRZ ang c t) = applyCRZ ang c t

-- Conjugate a sum with a circuit.
applyCircuit :: [Gate] -> PauliSum -> PauliSum
applyCircuit gates paulis = foldl' (flip applyGate) paulis gates
```

other small steps in the actual solution defined,

```haskell
-- Round the weight of a PauliTerm to +/-1.
roundWeight :: PauliTerm -> PauliTerm
roundWeight (term, ampl) = (term, ampl')
  where ampl' = if ampl < 0.0 then -1.0 else 1.0

-- Return the sign of a weight.
weightSign :: Double -> Char
weightSign ampl = if ampl < 0.0 then '-' else '+'

-- Parse a QASM input string as a list of Gates.
parseQasm :: String -> [Gate]
parseQasm input = error "not implemented" -- omitted for brevity

-- Canonicalize a list of Pauli terms.
canonicalize :: [PauliTerm] -> [PauliTerm]
canonicalize = error "not implemented" -- see hstab technical description
```

and, finally, the problem solved.

```haskell
main :: IO ()
main = do
  qasmString ← readFile "input.qasm"
  let (nqubits, circuit) = parseQasm qasmString
  let trunc = Just 1e-5 -- relatively large truncations can be used
  let evolvedStabs =
        -- construct stabilizers for the all-zero state as PauliSums
        [0 .. nqubits - 1]
        & map (\k → single trunc 1.0 $ set k Z $ ident nqubits)
        -- apply the circuit and return the max-weight term
        & mapMaybe (maxWeight . applyCircuit circuit)
  let answer =
        -- canonicalize the final set
        evolvedStabs
        & canonicalize
        -- convert to a concatenated string
        & concatMap (\(term, ampl) → weightSign ampl : show term)
  putStrLn answer
```

References

[1] M. S. Rudolph, T. Jones, Y. Teng, A. Angrisani, and Z. Holmes, "Pauli Propagation: A Computational Framework for Simulating Quantum Systems." arXiv:2505.21606 (2025).

[2] H. Gharibyan, S. Hariprakash, M. Z. Mullath, and V. P. Su, "A Practical Guide to Pauli Path Simulators for Utility-Scale Quantum Experiments." arXiv:2507.10771 (2025).