

# Efficiently Trotterizing Lithium Hydride Time Evolution Step

## Experimental Project Classiq

David Ponarovsky

July 29, 2023

### Abstract

In this paper, we review our experimental implementation of an algorithm for designing time evolution simulation circuits of molecules, given their Hamiltonian in the Pauli basis. Our algorithm employs several heuristics to minimize the circuit depth. We demonstrated this by taking the lithium hydride molecule as a use case and comparing the generated circuit to the one produced by the naive approach.

## 1 Preamble.

Simulation is definitely one of the core ingredients that our modern civilization relies on; the fact that one can check on their computer if their sketch for an electronic chip or plane will actually compute or fly without the need to construct and test a physical prototype both cheapens and accelerates the development process.

Though humanity has successfully enabled an efficient simulation in a variety of regimes, there are still areas in which we don't have any other choices than performing physical experiments. Development of materials, drugs, and medicines is a good example of a processes that suffer from the lack of simulation and can last for decades. What distinguishes those areas is the fact that the mechanics theory governing them is quantum mechanics, which we have good reason to believe cannot be simulated by classical computers. Yet, quantum computers are considered to be good candidates to overcome that barrier.

Unfortunately, our current quantum machines are extremely limited in terms of computing resources; the most advanced computers have access to no more than a few hundred qubits and suffer from a considerable amount of noise, which cause long computations resulting in nonsense. Therefore, any short-term application must be cost-efficient.

On May 2022, Classiq, a pioneering quantum computing company, launched the first (at least at that scale)

competition in quantum circuits programming. The initial purpose of this work was to solve the Lithium simulation problem. In short, the problem asked to come up with a quantum circuit, restricted to only a few qubits, that progress the lithium hydride molecule in time. The winner of the competition would be the circuit with the shortest depth. We mention that our algorithm is generic and the software we have written can easily be modified for any other Hamiltonian.

The paper organized in the follow systematically manner, first in the preamble we will present the problem and talk about the general concepts which describe our specific Hamiltonian, including the quantities regime (i.e number of qubits vs number of Hamiltonians terms), the naive solution, and how much improvement we could hope to make.

Then in next section we review all the techniques that we have used to improve a local gates assignment. Namely, this section deals with the question of how presenting each of the local term as subgate.

In contrast, the third and the forth sections reviews methods to determinate an order which clearly superior comparing to the naive approach. The forth section presenting an concept of analyzing the "product graph" a concept which could be thought as the second order analyses of the alternate path from the third section.

**The problem.** *Generate a circuit, using no more than 10 qubits, that approximates the unitary  $e^{-iH}$  where  $H$*

is the qubit Hamiltonian of a **LiH** (lithium hydride) molecule. The **LiH** Hamiltonian is composed of 276 Pauli strings, and can be found [HERE](#). The approximation error is defined in the next section, and should be less than 0.1. The circuit should be composed of the  $CX$  and single qubit gates only.

**Simulation in nutshell.** Let's begin by introducing notation, adhering to standard definitions; a quantum state  $|\psi\rangle$  is a vector in a Hilbert space, and a Hamiltonian is a Hermitian operator acting on that space. As physical fundamentals are not the main focus of the paper, we will merely mention, without providing justification, that a quantum state at time  $t$  in the presence of a Hamiltonian  $H$  is given by the relation  $|\psi(t)\rangle = e^{-iHt}|\psi(0)\rangle$ . Note that, due to the fact that  $H$  is a Hermitian operator,  $e^{-iHt}$  is a unitary operator.

For example, suppose that  $|\psi\rangle = |00\rangle$  and  $H = Z \otimes Z$ . Thus, the time evolution operator is  $e^{itZ \otimes Z}$ . As  $|00\rangle$  is an eigenstate of  $H$ , since  $Z \otimes Z |00\rangle = |00\rangle$ , we have that  $|\psi(t)\rangle = e^{-it} |00\rangle$ . However, if  $|\psi\rangle = |00\rangle + |10\rangle$  (ignoring normalization factors), then we would have that  $|\psi(t)\rangle = e^{-it} |00\rangle + e^{it} |10\rangle$ .

For simulating a general Hamiltonian, recall that the Pauli matrices, along with the identity, span the space of matrices. Thus, one can assume that the Hamiltonian is given as a linear combination of Paulis. We will denote it as  $H = \sum_i H_i$ , where each of the  $H_i$  is a Pauli product multiplied by a scalar factor.

Using the Lie-Trotter formula [\[Tro59\]](#)

$$e^{A+B} = \lim_{n \rightarrow \infty} (e^{A/n} e^{B/n})^n$$

for arbitrary matrices  $A, B$ , we obtain that one can apply the gates  $e^{\Delta t H_i}$  for  $1/(\Delta t)$  iterations and expect to converge to the desired product. This insight defines a circuit which we will refer to as the naive approach.

**Naive approach.** Before diving into the technical details, let's first review our competitor. Consider the straightforward assignment Lie-Trotter [\[Tro59\]](#), which handles each of the Hamiltonian terms one by one. Assume that the Hamming weight of the support of  $H_i$  is equal to  $d_i$ . Thus, we need two steps to rotate each wire into the parity base (relative to the Hamiltonian) and uncompute it in the end. Then, we can apply the  $CX$  from each qubit in the support to a (arbitrary) chosen

one, which will sum the wires' parity and finally propagate the sum through the  $RZ(\theta \Delta t)$  gate (rotation by the coefficient of the term and the step size). Therefore, in total we pay  $2 + 2d_i$  for each term and the whole circuit will require:

$$D^{\text{naive}}(n, m) = \sum_i 2 + 2d_i = m \sum_i \frac{2 + 2d_i}{m} = 2m \left( 1 + \mathbb{E}_{\sim i}[H] \right)$$

Where  $D^{\text{naive}}(n, m)$  is the depth of the circuit which computes a single  $\Delta t$  step of the Lie-Trotter formula [\[Tro59\]](#). To gain a better understanding of what constitutes a good solution, let us assume for the moment that  $\mathbb{E}_{\sim i}[H] > 5$ . Under this assumption, the naive approach yields a circuit of depth  $\geq 2 \cdot 276 \cdot 6 \sim 3100$ . Ultimately, we will see that our solution produces a 1759-depth circuit.

**Characterize our LiH.** A natural question to ask is whether there exists a more efficient way to present the Hamiltonian. Although we cannot be certain that there is not, let's try to estimate how much improvement can be achieved by applying the Bravyi-Kitaev transformation [\[BK02\]](#). In one sentence, Bravyi and Kitaev have shown that the ladder and annihilation operators have Pauli representations such that each operator touches at least  $\log(n)$  of the qubits. Since the Hamiltonian of molecule<sup>1</sup> in the energy base has a "product" of four operators (i.e  $a_i^\dagger a_j^\dagger a_n a_m$ ), we get that there may be operators in the computing base which touch  $4 \log(n)$  qubits. In our case,  $n = 10 \Rightarrow 4 \log(n) > n$ . In conclusion, it seems that the known techniques for reducing the support of the terms asymptotically do not work here. This is an indication that a 10-fold improvement is unlikely. Therefore, we will aim for a 2-fold improvement over the naive approach.

## 2 Single Term Heuristics.

**Main wire principle.** From now on, we will refer to the wire that sums the parity of the state as the main wire. For a given term, we have chosen the main wire to be the median of the wires in its support. For example, consider the following term:

$$XXXXIII$$

<sup>1</sup>Sometimes called electronic structure Hamiltonian

Then, as the second wire is above (and beneath) half of the wires, it will be the main wire in that case. The separation into upper and lower sides will enable us to guarantee that at least two  $CX$  will be added in parallel (using the next heuristic).

**Upper and lower summations.** We sum the parity of the given term as follows. Denote by  $i_1, i_2, \dots, i_W, \dots, i_k \subset [n]$  the indices in the support of the

given term, such that  $i_W$  is the main wire. After rotating the wires into the phase base, we sum the parity of  $i_1, i_2, \dots, i_{W-1}$  into  $i_{W-1}$  by applying a  $CX(i_j, i_{W-1})$  for  $j < W - 1$ .

Since all the segments  $[i_j, i_{W-1}]$  are disjoint from any segments of the form  $[i_{W+1}, i_j]$  for  $j > W + 1$ , we can sum the parity of  $i_{W+1}, i_{W+2}, \dots, i_k$  wires into  $i_{W+1}$  in parallel. This heuristic reduces the depth of a single term by almost half.



Figure 1: Demonstrating the above methods applied to the terms  $XIXZZXIXXZ$  and  $XIXXXZXIXXZ$ , the fifth wire (qubit) is the main wire that sums the parity.

### 3 Greedy Order Heuristics.

Next, we will review our methods to impose the gates order. Clearly the order does matter, to see that consider an hamiltonian pair which differ by only single coordinate i.e  $XXXX$  and  $XXXZ$ . In that case the contribute of the first tree qubits remain the same for both of them, and therefore there is no needed to uncompute them after the applying of the first gate.

**Sample Greedy Diameters.** Assume that we given a set of therms with a promise that it might has a lot of "closer" terms (relative to the Hamming distance of their supports). Our first simple heuristic goal is to chain them together in greedy manner such that the distance of each adjacent terms will be minimal.

---

**Algorithm 1:** Chain an Hamiltonian set
 

---

```

while  $G \neq \emptyset$  do
   $T \leftarrow$  Greedy Spanning Tree ( $G$ )
   $D_i \leftarrow$  Diameter ( $T$ )
   $G \leftarrow G/D_i$ 
end
return  $D_0 \dots D_l$ 
  
```

---

It's worth to note that replacing the Greedy Spanning Tree by MST (and the diameter by just the DFS scanning order of the MST) could be proving as 2-OPT in the terms of chaining. We think about the uncompute stage as the climbing back at the tree, and therefore  $\text{OPT} \leq 2w(\text{MST})$ . But also it easy to see that every path which contain all the vertices has weigh greater

than  $w(\text{MST})$ , Hence  $2w(\text{MST}) \leq 2\text{OPT}$ . Yet, By the fact that we have already improved the naive solution at factor which close to  $\frac{1}{2}$  and by the intuition that the optimal depth is not very far from the naive (don't expect that  $D^{\text{naive}} \geq 4\text{OPT}$  we decided to stick to the greedy construction, which in the next chapter will generalized to the product graph.

**Hyperplanes separation.** The cost of chaining pair of Hamiltonians isn't a monotone function of their Hamming distance, for example consider the case of two terms which have not overlapping at all, i.e  $XXXXIIII$  and  $IIIIXXXX$  then it's clear that those two can be impose such they will be computed in parallel.

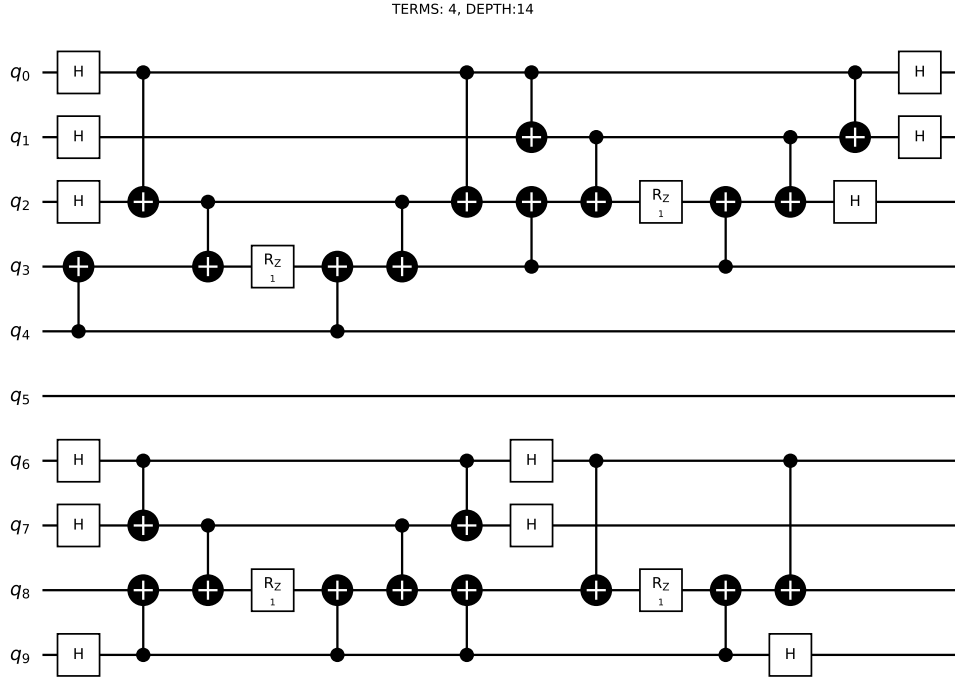


Figure 2: Example of a case in which chaining terms with high distance reduces the depth of the circuit. Here the terms are:  $XIXZZIIII$ ,  $XXXZIIII$ ,  $IIIIIXZX$ , and  $IIIIIZIX$ .

Technically we have separated those groups by follow, first we have sorted the term collection by the follow

weight functions  $f, g$  such that:

$$\begin{aligned}
 f(H_i) &= \min j : H_{j'} = I \text{ for } j' > j \\
 g(H_i) &= \max j : H_{j'} = I \text{ for } j' \leq j
 \end{aligned}$$

And then we have matched pairs till we get into intersection.

## 4 The Product Graph.

The main disadvantage of the last separation is that it doesn't take into account any relation between terms which classified in the same side of the plane. The suggested solution: We will say that the pair  $H_i, H_j$  share a "solid product" if  $f(H_i) \leq g(H_j)$  where  $f, g$  defined above. We denote such relation by  $(H_i, H_j) \in \mathbf{SP}$ . Define the graph  $G^2 = (V \times V, E')$  where  $V$  is the set of the terms, to be:

$$E' = \{ \{(u, v), (w, z)\} : (u, v), (w, z), (u, z), (w, v) \in \mathbf{SP} \}$$

$$w(e) = w(\{(u, v), (w, z)\}) = \max w(u, w), w(v, z)$$

Note, that in the  $G^2$  path which pass through all the vertices contain an  $|V|$  copies of each term. Hence looking for spanning trees makes no sense. Therefore our diameters sampler will be more appropriate for that task (when we ensure that each sampled tree contains at most one copy of each term).

Another advantage of this method that it can be easily generalize to scanning triples or higher orders by taking higher power of the graph, the main disadvantage is (of course) processing time.

The final submission is a mixture of all the methods we have reviewed above.

## References

- [Tro59] H. F. Trotter. "On the Product of Semi-Groups of Operators". In: *Proceedings of the American Mathematical Society* 10.4 (1959), pp. 545–551. ISSN: 00029939, 10886826. URL: <http://www.jstor.org/stable/2033649> (visited on 07/28/2023).
- [BK02] Sergey B. Bravyi and Alexei Yu. Kitaev. "Fermionic Quantum Computation". In: *Annals of Physics* 298.1 (May 2002), pp. 210–226. DOI: [10.1006/aphy.2002.6254](https://doi.org/10.1006/aphy.2002.6254). URL: <https://doi.org/10.1006/aphy.2002.6254>.

```

1
2 from importlib.resources import path
3 from math import ceil
4 from struct import unpack
5 from qiskit import QuantumCircuit
6 from qiskit.visualization import circuit_drawer
7 from matplotlib import pyplot as plt
8 import datetime
9 import numpy as np
10
11 N = 4
12 WIRES = 10
13 TEST = "NOT"
14 RANDOMSTACTICS = True
15 STRATEGY = "PRODUCT"
16 OUTSIDE_OF_CIRCUIT = WIRES + 1
17 PATHFILE = {
18     "NOT" : "./LiH",
19     "YES" : "./LiH_test",
20     "PRODUCT" : "./LiH_test_product_space"
21 }[TEST]
22
23
24 class local_Hamiltonian():
25     def __init__(self, tensor, weight) -> None:
26         self.tensor = tensor
27         self.weight = weight
28         self.parent = self
29
30     def tensorspace(self, other) -> bool:
31         for A,B in zip( list(self.tensor), list(other.tensor)):
32             if "I" not in [A, B]:
33                 return False
34         return True
35
36     def dis(self, other) -> int:
37         ret = 0
38         for A,B in zip(list(self.tensor), list(other.tensor)):
39             if A != B:
40                 ret += 1
41         return ret
42
43     def solid_product(self, other):
44         indices = []
45         for j in range(1,WIRES):
46             l= "X" * j + "I" * (WIRES-j)
47             r = "I" * j + "X" * (WIRES-j)
48             if local_Hamiltonian(l,0).tensorspace(self) and\
49                 local_Hamiltonian(r,0).tensorspace(other):
50                 indices.append(j)
51
52         if len(indices) > 0:
53             return True, indices
54         return False, []
55
56     def newbase(self, perm):
57         tensor = [ "" ] * len(self.tensor)
58         for i in range(len(perm)):
59             tensor[i] = self.tensor[perm[i]]
60         ret = local_Hamiltonian(tensor,self.weight)
61         ret.parent = self.parent
62         return ret
63

```

```

64 def median(self):
65     support = list(filter( lambda x : self.tensor[x] != 'I', range(WIRES)))
66     if len(support) != 0:
67         return support[int(len(support)/2)]
68     return 0
69
70 def seconded_wires(self):
71     j = self.median()
72
73     def find_nearset_nontrival(sign):
74
75
76         gen = [ (i,v) for i,v in list(enumerate(self.tensor))[j+1:]] if sign ==1\
77                 else [ (i,v) for i,v in list(enumerate(self.tensor))[j-1::-1]]
78
79         for i,v in gen:
80             if v != "I":
81                 return i
82         return j
83
84     pos_wire = j
85     neg_wire = j
86     if j + 1 < WIRES:
87         pos_wire = find_nearset_nontrival(1)
88     if j - 1 >= 0:
89         neg_wire = find_nearset_nontrival(-1)
90
91     return neg_wire, pos_wire
92
93 def parser_line(line) -> local_Hamiltonian:
94     line = line.split()
95     return local_Hamiltonian( list(line[-1]),
96                               { "-" : -1 , "+" : 1 }[line[0]] * np.float64(line[1]) / N )
97
98 def parser() -> None:
99     hamiltonians = [ ]
100     for line in open(PATHFILE).readlines():
101         if len(line) > 1:
102             hamiltonians.append(parser_line(line))
103     return hamiltonians
104
105 def donothing(_):
106     pass
107
108 def rotateY(cir):
109     def _func(wire):
110         cir.s(wire)
111         cir.h(wire)
112     return _func
113
114 def unrotateY(cir):
115     def _func(wire):
116         cir.h(wire)
117         cir.sdg(wire)
118     return _func
119
120 def MulByterm (circuit : QuantumCircuit, term ,next_terms = [], last_terms = [],
121 main_wire = WIRES-1) -> QuantumCircuit:
122
123     def reursive_manner(tensor, wire, weight, last_wire, _sign, first_not_trival=True):
124
125         if wire < 0 or wire == WIRES:
126             return QuantumCircuit(WIRES),QuantumCircuit(WIRES)
127

```

```

128     compute = {
129         "X" : lambda cir : cir.h,
130         "Y" : lambda cir : rotateY(cir),
131         "Z" : lambda cir : donothing,
132         "I" : lambda cir : donothing }
133
134
135     uncompute = {
136         "X" : lambda cir : cir.h,
137         "Y" : lambda cir : unrotateY(cir),
138         "Z" : lambda cir : donothing,
139         "I" : lambda cir : donothing }
140
141     pauli = tensor[wire]
142     if wire == main_wire:
143         circuit_node = QuantumCircuit(WIRES)
144         LU, RU = reursive_manner(tensor, wire-1, weight, last_wire, -1, first_not_trival = True)
145         LD, RD = reursive_manner(tensor, wire+1, weight, last_wire, 1, first_not_trival = True)
146
147         compute[pauli](circuit_node)(wire)
148         for L in [ LD, LU]:
149             circuit_node = circuit_node.compose(L)
150         circuit_node.rz(2*weight, main_wire)
151         for R in [ RD, RU]:
152             circuit_node = circuit_node.compose(R)
153         uncompute[pauli](circuit_node)(wire)
154         return circuit_node
155
156     if pauli == 'I':
157
158
159         # temp_wire = last_wire + _sign if first_not_trival else last_wire
160         return reursive_manner(tensor, wire + _sign, weight,
161             last_wire, _sign, first_not_trival = first_not_trival)
162
163     else:
164
165         temp_wire = wire if first_not_trival else last_wire
166         parity_collector = False
167         if first_not_trival:
168             parity_collector = True
169
170
171         L, R = reursive_manner(tensor, wire + _sign, weight, temp_wire, _sign, first_not_trival =
False)
172         circuit_left, circuit_right = QuantumCircuit(WIRES), QuantumCircuit(WIRES)
173
174         if (parity_collector) or not (
175             (last_terms[wire][0] == pauli) and\
176             (last_terms[wire][1][ { 1 : 1 , -1 : 0 }[_sign] ] == last_wire)):
177
178             compute[pauli](circuit_left)(wire)
179             circuit_left = circuit_left.compose(L)
180             circuit_left.cx(wire, last_wire)
181
182         else:
183             circuit_left = L
184
185         if (parity_collector) or not (
186             (next_terms[wire][0] == pauli) and\
187             (next_terms[wire][1][ { 1 : 1 , -1 : 0 }[_sign] ] == last_wire)):
188
189             circuit_right.cx(wire, last_wire)
190             circuit_right = circuit_right.compose(R)

```



```

191         uncompute[pauli](circuit_right)(wire)
192     else:
193         circuit_right = R
194     return circuit_left, circuit_right
195
196
197 circuit = circuit.compose(reursive_manner(term.tensor, main_wire, term.weight, main_wire, 0))
198 return circuit
199
200
201 def cutting(circuit : QuantumCircuit):
202     '''Second optimization, cuts the gates which are followed by their
203     uncompute '''
204     def filter_by_wire(wire):
205         return list(filter( lambda item :\
206             any( [register.index == wire for register in item[1][1]] ), enumerate(circuit.data) ))
207
208
209 UNCOMPTE = {
210     "h" : "h",
211     "cx" : "?",
212     "rz" : "?",
213     "s" : "sdg",
214     "sdg" : "s",
215     "sxdg" : "?" }
216
217 indices_todelete = []
218 for wire in range(WIRES):
219     operators = filter_by_wire(wire)
220
221     j = 0
222     while (j < len(operators) - 1 ):
223         if ( UNCOMPTE[operators[j][1][0].name] == operators[j+1][1][0].name ):
224             indices_todelete.append( operators[j][0] )
225             indices_todelete.append( operators[j+1][0] )
226         j += 1
227
228 for index in reversed(sorted(indices_todelete)):
229     circuit.data.pop(index)
230
231 return circuit
232
233
234 def genreate_circuit(terms = None):
235     circuit = QuantumCircuit(WIRES)
236     terms = parser() if terms == None else terms
237     print(len(terms))
238     for i, term in enumerate(terms):
239
240         main_wire = term.median()
241         next_terms,last_terms = [],[]
242
243         for _j in range(WIRES):
244             found = False
245             if i+1 < len(terms):
246                 for _term in terms[i+1:]:
247                     if _term.tensor[_j] != 'I':
248                         next_terms.append( (_term.tensor[_j], _term.seconed_wires() ))
249                         found = True
250                         break
251             if not found:
252                 next_terms.append( ('I', OUTSIDE_OF_CIRCUIT ) )
253
254         found = False

```

```

255         if i > 0:
256             for _term in terms[i-1::-1]:
257                 if _term.tensor[_j] != 'I':
258                     last_terms.append( (_term.tensor[_j], _term.seconed_wires() ))
259                     found = True
260                     break
261             if not found:
262                 last_terms.append( ('I', OUTSIDE_OF_CIRCUIT ))
263
264         circuit = MulByterm(circuit, term, main_wire=main_wire,
265                             next_terms=next_terms, last_terms=last_terms )
266
267     return circuit
268
269 def genreate_optimized_circut(circuit, terms, svg =False, entire = False):
270     circuit = cutting(cutting(circuit))
271
272     if entire:
273         for _ in range(ceil(np.log(N))):
274             circuit = circuit.compose(circuit)
275             circuit = cutting(circuit)
276
277     print(f"TERMS: {len(terms)}, DEPTH:{circuit.depth()}")
278
279     if svg:
280         circuit_drawer(circuit, output='mpl',style="bw", fold=-1)
281         plt.title( f"TERMS: {len(terms)}, DEPTH:{circuit.depth()}")
282         plt.tight_layout()
283         plt.savefig(f'Ham_{STRATEGY}-{datetime.datetime.now()}.svg')
284
285     if entire:
286         open(f"Ham_{STRATEGY}-{datetime.datetime.now()}.qasm", "w+").write(circuit.qasm())
287
288     return circuit.depth()
289
290
291

```

```

1 from copy import deepcopy
2 import networkx as nx
3 from Hamiltonian_parser import WIRES, parser, local_Hamiltonian, genreate_circuit, genreate_optimized_circuit
4 from itertools import permutations, product, combinations
5 from random import choice
6 import pickle as pkl
7
8 from matplotlib import pyplot as plt
9
10
11 class Permutation_Base():
12     def __init__(self, arr) -> None:
13         self.arr = arr
14         self.parent = self
15
16
17 def generated_the_product_graph(terms = parser()):
18     def generated_the_product_graph_by_base(_terms, number_premu=0):
19
20         G = nx.Graph()
21         Gproduct = nx.Graph()
22
23         edges_set = set()
24
25         for (H1, H2) in product(_terms, _terms):
26             if H1.tensorspace(H2):
27                 G.add_edge(H1, H2)
28                 edges_set.add( (H1,H2) )
29                 G.edges[(H1, H2)]['weight'] = H1.dis(H2)
30                 G.edges[(H1, H2)]['solid'] = True
31                 G.edges[(H1, H2)]['permutation'] = j
32
33         for e in G.edges():
34             H1,H2 = e
35             for H3,H4 in \
36                 product(list(G.adj[H1]),list(G.adj[H2])):
37                 if (H1,H4) in edges_set and\
38                     (H3,H4) in edges_set and\
39                     (H2,H3) in edges_set:
40                     Gproduct.add_edge((H1,H2), (H3, H4))
41                     Gproduct.edges[(H1, H2), (H3, H4)]['weight'] = max(H1.dis(H3), H2.dis(H4))
42                     Gproduct.edges[(H1, H2), (H3, H4)]['solid'] = True
43                     Gproduct.edges[(H1, H2), (H3, H4)]['permutation'] = j
44
45         for (H1, H2) in product(_terms, _terms):
46             if (H1,H2) not in edges_set:
47                 G.add_edge(H1, H2)
48                 G.edges[(H1, H2)]['weight'] = H1.dis(H2)
49                 G.edges[(H1, H2)]['solid'] = False
50                 G.edges[(H1, H2)]['permutation'] = j
51
52         print(f"vertices:{Gproduct.number_of_nodes()}\t edges: ~{Gproduct.number_of_edges()}")
53         return Gproduct, G, _terms
54
55     return pkl.load( open(f"mainG-276-1.pkl", "br"))
56
57 permutations = list(map(lambda x: Permutation_Base(x) , [
58     # [0,2,4,6,8,1,3,5,7,9],
59     [0,1,2,3,4,5,6,7,8,9]
60     # [0,7,4,6,8,9,3,5,2,1]
61 ]))
62 graphs = []
63 perm_terms = []

```

```

64     mainG = nx.Graph()
65     mainProductG = nx.Graph()
66     for j, permutation in enumerate(permutations):
67         perm_terms = list(map( lambda x : x.newbase(permutation.arr), terms))
68         productG, G, _ = generated_the_product_graph_by_base(perm_terms, number_premu=j)
69
70         if mainG.number_of_nodes() == 0:
71             mainG = nx.compose(mainG, G)
72             mainProductG = nx.compose(mainProductG, productG)
73
74     pickle.dump((mainG, mainProductG, terms ,permutations), open(f"mainG-{len(terms)}-
{len(permutations)}.pkl", "bw+"))
75     return mainG, mainProductG, terms, permutations
76
77 def select(_list, v, G, flag = True):
78     minimal = min(_list, key =lambda u : G.edges[v,u]['weight'] )
79     return choice( [r for r in _list if G.edges[v,r]['weight'] == \
80         G.edges[v,minimal]['weight'] ])
81
82 def notcolorized(node, _set):
83     if isinstance(node, tuple):
84         term1, term2 = node
85         if (term1.parent in _set) or (term2.parent in _set):
86             return False
87         return True
88     else:
89         return node.parent not in _set
90
91 def colirize(node, _set):
92     if isinstance(node, tuple):
93         term1, term2 = node
94
95         if (term1.parent in _set) or (term2.parent in _set):
96             return False, _set, 1
97         _set.add(term1.parent)
98         _set.add(term2.parent)
99         return True
100    else:
101        if node.parent in _set:
102            return False
103        else:
104            _set.add(node.parent)
105            return True
106
107
108 #randomized DFS.
109 def sample_path(G, terms) -> tuple((nx.Graph, set)):
110     print("sample_path")
111     color = set()
112
113     def DFS(v, T, _color, sign=0, flag=True):
114
115         if not colirize(v, _color):
116             return _color, sign
117
118         can_packed = [None]
119         while len(can_packed) > 0 :
120             can_packed = list(filter(lambda x :\
121                 notcolorized(x, _color), G.adj[v]))
122
123             if len(can_packed) > 0:
124                 u = select( can_packed, v, G, flag=flag)
125                 T.add_edge(v,u)
126                 if 'permutation' in G.edges[v,u]:

```

```

127         T.edges[v,u]['permutation'] = G.edges[v,u]['permutation']
128         T.edges[v,u]['sign'] = sign
129         _color, sign = DFS(u, T, _color, sign=sign+1, flag=flag)
130     return _color, sign
131
132 T = nx.DiGraph()
133 l = []
134
135 for v in G.nodes():
136     T.add_node(v)
137     l.append(v)
138
139 _sign = 0
140 for v in G.nodes():
141     color, _sign = DFS(v, T, color, sign=_sign, flag=False)
142
143 return T, color
144
145 def get_Diameter(Tree: nx.Graph) -> tuple((tuple([], int)), tuple([], int))) :
146
147     def DFS_tree_depth(G, v):
148
149         if len(list(G.adj[v])) == 0 :
150             return (([v],1),([v],1))
151
152         branches = []
153         maxinnerpath, maxinnerdepth = [],0
154         for u in list(G.adj[v]):
155             ((temppath, tempdepth), \
156              (tempinnerpath, tempinnerdepth)) = DFS_tree_depth(G,u)
157
158             if maxinnerdepth < tempinnerdepth:
159                 maxinnerpath, maxinnerdepth = tempinnerpath, tempinnerdepth
160
161             branches.append((temppath, tempdepth))
162
163         maxpath, maxdepth = [],0
164
165         for (b1, d1),(b2, d2) in combinations(branches, r=2):
166             if 1 + d1 + d2 > maxinnerdepth:
167                 maxinnerpath = b1 + [ v ] + b2
168                 maxinnerdepth = 1 + d1 + d2
169
170         for b,d in branches:
171             if 1 + d > maxdepth:
172                 maxpath = [v] + b
173                 maxdepth = 1 + d
174         return ((maxpath,maxdepth), (maxinnerpath, maxinnerdepth))
175
176     ((maxpath,maxdepth), (maxinnerpath, maxinnerdepth)) = DFS_tree_depth(Tree, list(Tree.nodes)[0])
177     print(maxdepth, maxinnerdepth)
178     return maxpath if maxdepth > maxinnerdepth else maxinnerpath
179
180
181 def generate_simple_graph(_terms):
182     G = nx.Graph()
183     for (H1, H2) in product(_terms, _terms):
184         G.add_edge(H1, H2)
185         G.edges[(H1, H2)]['weight'] = H1.dis(H2)
186     return G
187
188 def greedy_path( terms ):
189     G = generate_simple_graph(terms )
190

```

```

191 def reursive_form( _terms ):
192     if len(_terms) < 3:
193         return _terms
194     else:
195
196         T, _ = sample_path(G, _terms)
197
198         Q = get_Diameter(T)
199         color = set()
200
201         ret = []
202         for H in Q:
203             if H.parent not in color:
204                 color.add(H.parent)
205                 ret.append(H.parent)
206                 G.remove_node(H)
207
208         remain_terms = [ term for term in _terms if term not in color ]
209         return ret + reursive_form(remain_terms)
210 return reursive_form(terms)
211
212 def Hamiltonian_sorting(hamiltonians):
213     groups = [[] for _ in product(range(WIRES), range(WIRES))]
214     for term in hamiltonians:
215         x,y = term.seconed_wires()
216         groups[x + WIRES*y].append(term)
217
218     ret = [ ]
219     for group in groups:
220         group = greedy_path(group)
221         ret += group
222     return ret
223
224 def enforce_seapration(hamiltonians):
225
226     def seperate(terms):
227
228         def sort_tensor_by_geometrical_support(tensor, up = True):
229
230
231             for j,pauli in enumerate( { True: tensor, False: reversed(tensor) }[up] ):
232                 if pauli != "I":
233                     if up:
234                         return 10 - j
235                     else:
236                         return 10 - j
237             else:
238                 return 10
239
240         above = sorted(terms,\
241             key=lambda x : sort_tensor_by_geometrical_support(x.tensor, up=True))
242         beneath = sorted(terms,\
243             key=lambda x : sort_tensor_by_geometrical_support(x.tensor, up=False))
244
245         contact_point = min( range(len(terms)),\
246             key = lambda i : above[i].tensorspace(beneath[i]))
247
248         print(f"contact point: {contact_point}")
249         path = []
250         for x,y in zip( sorted(above[:contact_point], key = lambda z : z.tensor),\
251             sorted(beneath[:contact_point], key = lambda z : z.tensor)):
252             path.append(x)
253             path.append(y)
254

```

```

255     new_terms = []
256     for x in above[contact_point:]:
257         if x not in beneath[:contact_point]:
258             new_terms.append(x)
259     return new_terms, path
260 path = [ ]
261 terms ,temppath = seperate(deepcopy(hamiltonians))
262 path += temppath
263
264 print(f"terms:{len(terms)}")
265 return path, terms
266
267 def alternate_path_v2(mG : nx.Graph, G : nx.Graph,\
268     terms, permutations, single_iteration = False):
269     other_color = set()
270
271     T, _ = sample_path(G, terms)
272     Q = get_Diameter(T)
273     ret = []
274
275     for (u,v) in Q:
276         for H in [u,v]:
277             if H not in other_color:
278                 other_color.add(H)
279                 ret.append(H)
280
281     made_progress = True
282     _single_iteration = True
283     while _single_iteration and (len(Q) > 2 and made_progress):
284         made_progress = False
285         for u,v in Q:
286             for H in [u,v]:
287                 if H.parent not in other_color:
288                     ret.append(H)
289                     other_color.add(H.parent)
290                     for w in list(G.nodes()):
291                         if H in w:
292                             G.remove_node(w)
293                             made_progress = True
294
295                     if mG.has_node(H):
296                         mG.remove_node(H)
297
298     T, _ = sample_path(G, terms)
299     if T.number_of_edges() > 1:
300         Q = get_Diameter(T)
301     else:
302         break
303     _single_iteration = not single_iteration
304
305     if not single_iteration:
306         for term in terms:
307             if term.parent not in other_color:
308                 ret.append(term)
309
310     return ret, terms, other_color
311
312 def main_enforce(hamiltonians):
313     path, terms = enforce_seapration(hamiltonians)
314     path += Hamiltonian_sorting(terms)
315     circuit = genreate_circuit(path)
316     depth = genreate_optimized_circuit(circuit, hamiltonians, svg=False, entire=False)
317     return circuit
318

```

```

319 def compose_alternate_enforce():
320     G, mainProductG, terms, permus = generated_the_product_graph()
321     candidates = [ ]
322     for _ in range(5):
323         path, terms, color = alternate_path_v2( deepcopy(G), deepcopy(mainProductG), terms, permus,
single_iteration=True)
324         circuit = genreate_circuit(path)
325         genreate_optimized_circuit(circuit, terms, svg=False, entire=False)
326         remain_terms = [ term for term in terms if term not in color ]
327         circuit = circuit.compose( main_enforce( remain_terms ) )
328         depth = genreate_optimized_circuit(circuit, terms, svg = False, entire=False )
329         candidates.append( (depth, circuit) )
330         print(f"DEPTH: {depth}")
331     depth, circuit = min( candidates, key = lambda x : x[0] )
332     depth = genreate_optimized_circuit(circuit, terms, svg = False, entire=True)
333     # depth = genreate_optimized_circuit(circuit, terms, entire=True)
334     return circuit,terms
335
336
337
338
339 def demonstrate_fig( ):
340     path = [ local_Hamiltonian( "XIXZZIIIII", 0.5 ),
341             local_Hamiltonian( "XXXZIIIII", 0.5 ),
342             local_Hamiltonian( "IIIIIXXZX", 0.5 ),
343             local_Hamiltonian( "IIIIIZIZX", 0.5 ),
344             local_Hamiltonian( "IIIXIIZIX", 0.5 ) ]
345
346     path, terms = enforce_seapration(path)
347     circuit = genreate_circuit(path)
348     genreate_optimized_circuit(circuit ,path, svg=True, entire=False)
349
350
351 if __name__ == "__main__":
352
353     circuit, terms = compose_alternate_enforce()
354
355
356

```



1 + 0.003034656830204855 \* IIIYIIIIYY  
2 + 0.003034656830204855 \* IIXXIIIIYY  
3 + 0.003034656830204855 \* IIIYIIIIXX  
4 + 0.003034656830204855 \* IIXXIIIIXX  
5 - 0.008373361424264817 \* YZZYIIIIYY  
6 - 0.008373361424264817 \* XZZXIIIIYY  
7 - 0.008373361424264817 \* YZZYIIIIXX  
8 - 0.008373361424264817 \* XZZXIIIIXX  
9 + 0.00211113766859809 \* YZZYIIIIYY  
10 + 0.00211113766859809 \* XZZXIIIIYY  
11 + 0.00211113766859809 \* YZZYIIIIXX  
12 + 0.00211113766859809 \* XZZXIIIIXX  
13 - 0.00491756976241806 \* IIIIIIIIIYY  
14 - 0.00491756976241806 \* IIIIIIIIIXX  
15 + 0.010540187409026488 \* ZIIIIIIIIYY  
16 + 0.010540187409026488 \* ZIIIIIIIIXX  
17 - 0.0011822832324725804 \* IZIIIIIIYY  
18 - 0.0011822832324725804 \* IZIIIIIIXX  
19 - 0.0011822832324725804 \* IIZIIIIIIYY  
20 - 0.0011822832324725804 \* IIZIIIIIIXX  
21 - 0.00154067008970742 \* IIIZIIIIYY  
22 - 0.00154067008970742 \* IIIZIIIIXX  
23 + 0.011733623912074194 \* IIIIZIIIIYY  
24 + 0.011733623912074194 \* IIIIZIIIIXX  
25 + 0.0027757462269049522 \* IIIIIZIIYY  
26 + 0.0027757462269049522 \* IIIIIZIIXX  
27 + 0.0036202487558837123 \* IIIIIZIYY  
28 + 0.0036202487558837123 \* IIIIIZIXX  
29 + 0.0036202487558837123 \* IIIIIZIYY  
30 + 0.0036202487558837123 \* IIIIIZIXX  
31 + 0.005996760849734561 \* IIVZYIIVZY  
32 + 0.005996760849734561 \* IIXZXIIVZY  
33 + 0.005996760849734561 \* IIVZYIIXZX  
34 + 0.005996760849734561 \* IIXZXIIXZX  
35 + 0.004802531988356293 \* IIVYIIIVZY  
36 + 0.004802531988356293 \* IIXXIIIVZY  
37 + 0.004802531988356293 \* IIVYIIIXZX  
38 + 0.004802531988356293 \* IIXXIIIXZX  
39 - 0.004879740484191497 \* YZYIIIVZY  
40 - 0.004879740484191497 \* XZXIIIVZY  
41 - 0.004879740484191497 \* YZYIIIXZX  
42 - 0.004879740484191497 \* XZXIIIXZX  
43 + 0.005996760849734561 \* IYZZYIYZZY  
44 + 0.005996760849734561 \* IXZZXIYZZY  
45 + 0.005996760849734561 \* IYZZYIXZZX  
46 + 0.005996760849734561 \* IXZZXIXZZX  
47 + 0.004802531988356293 \* IYZYIIVZZY  
48 + 0.004802531988356293 \* IXZXIIVZZY  
49 + 0.004802531988356293 \* IYZYIIXZZX  
50 + 0.004802531988356293 \* IIXXIIIXZZX  
51 - 0.004879740484191497 \* YYIIIIYZZY  
52 - 0.004879740484191497 \* XXIIIIYZZY  
53 - 0.004879740484191497 \* YYIIIIIXZZX  
54 - 0.004879740484191497 \* XXIIIIIXZZX  
55 - 0.008373361424264817 \* IIIYYYZZZY  
56 - 0.008373361424264817 \* IIXXYZZZY  
57 - 0.008373361424264817 \* IIIYXXZZZX  
58 - 0.008373361424264817 \* IIXXXZZZX  
59 + 0.0307383271773138 \* YZZZYZZZY  
60 + 0.0307383271773138 \* XZZXZZZY  
61 + 0.0307383271773138 \* YZZYXZZZX  
62 + 0.0307383271773138 \* XZZXXZZZX  
63 - 0.0077644411821215335 \* YZZYIYZZZY

64 - 0.0077644411821215335 \* XZZXIYZZZY  
65 - 0.0077644411821215335 \* YZZYIXZZZX  
66 - 0.0077644411821215335 \* XZZXIXZZZX  
67 - 0.005949019975734247 \* IIIIIYZZZY  
68 - 0.005949019975734247 \* IIIIIXZZZX  
69 - 0.0351167704024114 \* ZIIIIYZZZY  
70 - 0.0351167704024114 \* ZIIIIIXZZX  
71 + 0.0027298828353264134 \* IZIIIIYZZY  
72 + 0.0027298828353264134 \* IZIIIXZZZX  
73 + 0.0027298828353264134 \* IIZIIYZZZY  
74 + 0.0027298828353264134 \* IIZIIXZZZX  
75 + 0.0023679368995844726 \* IIIZIIYZZY  
76 + 0.0023679368995844726 \* IIIZIXZZZX  
77 - 0.03305872858775587 \* IIIIZYZZZY  
78 - 0.03305872858775587 \* IIIIZXZZZX  
79 - 0.0021498576488650843 \* IIIIIYIZZY  
80 - 0.0021498576488650843 \* IIIIIXIZZX  
81 - 0.0021498576488650843 \* IIIIIYZIZY  
82 - 0.0021498576488650843 \* IIIIIXZIZX  
83 + 0.004479074568182561 \* IIIIIYZZIY  
84 + 0.004479074568182561 \* IIIIIXZZIX  
85 + 0.004802531988356293 \* IIVZYIIYYI  
86 + 0.004802531988356293 \* IIXZXIIYYI  
87 + 0.004802531988356293 \* IIVZYIIXXI  
88 + 0.004802531988356293 \* IIXZXIIXXI  
89 + 0.010328819322301622 \* IIVYIIIIYYI  
90 + 0.010328819322301622 \* IIXXIIIIYYI  
91 + 0.010328819322301622 \* IIVYIIIXXI  
92 + 0.010328819322301622 \* IIXXIIIXXI  
93 - 0.003466391848475337 \* YZYIIIIYYI  
94 - 0.003466391848475337 \* XZXIIIIYYI  
95 - 0.003466391848475337 \* YZYIIIIXXI  
96 - 0.003466391848475337 \* XZXIIIIXXI  
97 + 0.004802531988356293 \* IYZZYIYZYI  
98 + 0.004802531988356293 \* IXZZXIYZYI  
99 + 0.004802531988356293 \* IYZZYIXZXI  
100 + 0.004802531988356293 \* IXZZXIXZXI  
101 + 0.010328819322301622 \* IYZYIIYZYI  
102 + 0.010328819322301622 \* IXZXIIYZYI  
103 + 0.010328819322301622 \* IYZYIIXZXI  
104 + 0.010328819322301622 \* IXZXIIXZXI  
105 - 0.003466391848475337 \* YYIIIIYZYI  
106 - 0.003466391848475337 \* XXIIIIYZYI  
107 - 0.003466391848475337 \* YYIIIIIXXI  
108 - 0.003466391848475337 \* XXIIIIIXXI  
109 + 0.00211113766859809 \* IIIYVYZZYI  
110 + 0.00211113766859809 \* IIIXXYZZYI  
111 + 0.00211113766859809 \* IIIYVXZZXI  
112 + 0.00211113766859809 \* IIIXXXZZXI  
113 - 0.0077644411821215335 \* YZZZYZZZYI  
114 - 0.0077644411821215335 \* XZZZXYZZYI  
115 - 0.0077644411821215335 \* YZZZYXZZXI  
116 - 0.0077644411821215335 \* XZZZXZZXI  
117 + 0.006575744899182541 \* YZZYIYZZYI  
118 + 0.006575744899182541 \* XZZXIYZZYI  
119 + 0.006575744899182541 \* YZZYIXZZXI  
120 + 0.006575744899182541 \* XZZXIXZZXI  
121 + 0.023557442395837284 \* IIIIIYZZYI  
122 + 0.023557442395837284 \* IIIIIXZZXI  
123 + 0.010889407716094479 \* ZIIIIYZZYI  
124 + 0.010889407716094479 \* ZIIIIIXZZI  
125 - 0.0003518893528389501 \* IZIIIIYZZYI  
126 - 0.0003518893528389501 \* IZIIIXZZXI  
127 - 0.0003518893528389501 \* IIZIIYZZYI

128 - 0.0003518893528389501 \* IIZIIXZZXI  
129 - 0.00901204279263803 \* IIIZIYZZYI  
130 - 0.00901204279263803 \* IIIZIXZZXI  
131 + 0.0127339139792953 \* IIIIZYZZYI  
132 + 0.0127339139792953 \* IIIIZXZZXI  
133 - 0.003818281201314288 \* IIIIIYIZYI  
134 - 0.003818281201314288 \* IIIIIXIZXI  
135 - 0.003818281201314288 \* IIIIIYZIYI  
136 - 0.003818281201314288 \* IIIIIXZIXI  
137 + 0.004217284878422759 \* IYVIIIVYII  
138 + 0.004217284878422759 \* IXXIIIVYII  
139 + 0.004217284878422759 \* IYVIIIXXII  
140 + 0.004217284878422759 \* IXXIIIXXII  
141 - 0.004879740484191498 \* IIVZYYZYII  
142 - 0.004879740484191498 \* IIXZXYZYII  
143 - 0.004879740484191498 \* IIVZYXZXII  
144 - 0.004879740484191498 \* IIXZXXZXII  
145 - 0.003466391848475337 \* IIVYIYZYII  
146 - 0.003466391848475337 \* IIXXIYZYII  
147 - 0.003466391848475337 \* IIVYIXZXII  
148 - 0.003466391848475337 \* IIXXIXZXII  
149 + 0.004868302545087521 \* YZYIIVZYII  
150 + 0.004868302545087521 \* XZXIIVZYII  
151 + 0.004868302545087521 \* YZYIIXZXII  
152 + 0.004868302545087521 \* XZXIIXZXII  
153 - 0.004879740484191498 \* IYZZYVYIII  
154 - 0.004879740484191498 \* IXZZXYVYIII  
155 - 0.004879740484191498 \* IYZZYXXIII  
156 - 0.004879740484191498 \* IXZZXXXIII  
157 - 0.003466391848475337 \* IYZYIVYIII  
158 - 0.003466391848475337 \* IXZXIVYIII  
159 - 0.003466391848475337 \* IYZYIXXIII  
160 - 0.003466391848475337 \* IXZXIXXIII  
161 + 0.004868302545087521 \* VYIIIVYIII  
162 + 0.004868302545087521 \* XXIIIVYIII  
163 + 0.004868302545087521 \* VYIIIXXIII  
164 + 0.004868302545087521 \* XXIIIXXIII  
165 - 0.004917569762418068 \* IIIYVYIIII  
166 - 0.004917569762418068 \* IIIXXIIII  
167 + 0.0027757462269049522 \* ZIIYVYIIII  
168 + 0.0027757462269049522 \* ZIIXXIIII  
169 + 0.0036202487558837123 \* IZIVYVYIIII  
170 + 0.0036202487558837123 \* IZIXXIIII  
171 + 0.0036202487558837123 \* IIZYVYIIII  
172 + 0.0036202487558837123 \* IIZXXIIII  
173 - 0.005949019975734285 \* YZZZYIIII  
174 - 0.005949019975734285 \* XZZZXIIII  
175 - 0.0021498576488650843 \* YIZZYIIII  
176 - 0.0021498576488650843 \* XIZZXIIII  
177 - 0.0021498576488650843 \* YZIZYIIII  
178 - 0.0021498576488650843 \* XZIZXIIII  
179 + 0.004479074568182561 \* YZZIVYIIII  
180 + 0.004479074568182561 \* XZZIXIIII  
181 + 0.02355744239583729 \* YZZYVYIIII  
182 + 0.02355744239583729 \* XZZXVYIIII  
183 - 0.003818281201314288 \* YIZYVYIIII  
184 - 0.003818281201314288 \* XIZYVYIIII  
185 - 0.003818281201314288 \* YZIVYVYIIII  
186 - 0.003818281201314288 \* XZIXVYIIII  
187 + 1.0709274663656798 \* IIIIIIIII  
188 - 0.5772920990654371 \* ZIIIIIIII  
189 - 0.4244817531727133 \* IZIIIIIIII  
190 + 0.06245512523136934 \* ZZIIIIIIII  
191 - 0.4244817531727134 \* IIZIIIIIIII

192 + 0.06245512523136934 \* ZIZIIIIIII  
193 + 0.06558452315458405 \* IZZIIIIIII  
194 - 0.3899177647415215 \* IIIZIIIIIII  
195 + 0.053929860773588405 \* ZIIZIIIIIII  
196 + 0.06022550139954594 \* IZIZIIIIIII  
197 + 0.06022550139954594 \* IIZZIIIIIII  
198 + 0.004360552555030484 \* YZZYZIIIII  
199 + 0.004360552555030484 \* XZZXZIIIII  
200 - 0.30101532158947975 \* IIIIZIIIII  
201 + 0.08360121967246183 \* ZIIIZIIIII  
202 + 0.06278876343471208 \* IZIIZIIIII  
203 + 0.06278876343471208 \* IIZIZIIIII  
204 + 0.053621410722614865 \* IIZZZIIIII  
205 + 0.010540187409026488 \* IIIYYZIIIII  
206 + 0.010540187409026488 \* IIXXZIIIII  
207 - 0.0351167704024114 \* YZZYZIIIII  
208 - 0.0351167704024114 \* XZZXZIIIII  
209 + 0.010889407716094479 \* YZZYIZIIIII  
210 + 0.010889407716094479 \* XZZXIZIIIII  
211 - 0.5772920990654372 \* IIIIIZIIIII  
212 + 0.11409163501020725 \* ZIIIIIZIIIII  
213 + 0.06732342777645686 \* IZIIIZIIIII  
214 + 0.06732342777645686 \* IIZIIZIIIII  
215 + 0.060505605672770954 \* IIIIZIZIIIII  
216 + 0.11433954684977561 \* IIIIZZIIIII  
217 - 0.0011822832324725804 \* IIIYYIZIII  
218 - 0.0011822832324725804 \* IIXXIZIII  
219 + 0.0027298828353264134 \* YZZYZIIZIII  
220 + 0.0027298828353264134 \* XZZXIZIIZIII  
221 - 0.0003518893528389501 \* YZZYIIZIIZIII  
222 - 0.0003518893528389501 \* XZZXIIZIIZIII  
223 - 0.42448175317271336 \* IIIIIIZIIZIII  
224 + 0.06732342777645686 \* ZIIIIIZIIZIII  
225 + 0.0782363777898523 \* IZIIIIIZIIZIII  
226 + 0.06980180803300681 \* IIZIIIIIZIIZIII  
227 + 0.07055432072184756 \* IIIZIIIZIIZIII  
228 + 0.06878552428444665 \* IIIIZIZIIZIII  
229 + 0.06245512523136934 \* IIIIIZZIIZIII  
230 - 0.0011822832324725804 \* IIIYYIIZIIZIII  
231 - 0.0011822832324725804 \* IIXXIIZIIZIII  
232 + 0.0027298828353264134 \* YZZYIIZIIZIII  
233 + 0.0027298828353264134 \* XZZXIIZIIZIII  
234 - 0.0003518893528389501 \* YZZYIIIZIIZIII  
235 - 0.0003518893528389501 \* XZZXIIIZIIZIII  
236 - 0.42448175317271336 \* IIIIIIIIZIIZIII  
237 + 0.06732342777645686 \* ZIIIIIIIZIIZIII  
238 + 0.06980180803300681 \* IZIIIIIIIZIIZIII  
239 + 0.0782363777898523 \* IIZIIIIIIIZIIZIII  
240 + 0.07055432072184756 \* IIIZIIIIIZIIZIII  
241 + 0.06878552428444665 \* IIIIIZIIZIIZIII  
242 + 0.06245512523136934 \* IIIIIZIZIIZIII  
243 + 0.06558452315458405 \* IIIIIIZZIIZIII  
244 - 0.00154067008970742 \* IIIYYIIIZIIZIII  
245 - 0.00154067008970742 \* IIXXIIIZIIZIII  
246 + 0.0023679368995844726 \* YZZYIIIZIIZIII  
247 + 0.0023679368995844726 \* XZZXIIIZIIZIII  
248 - 0.00901204279263803 \* YZZYIIIIIZIIZIII  
249 - 0.00901204279263803 \* XZZXIIIIIZIIZIII  
250 - 0.38991776474152134 \* IIIIIIIIZIIZIII  
251 + 0.060505605672770954 \* ZIIIIIIIZIIZIII  
252 + 0.07055432072184756 \* IZIIIIIIIZIIZIII  
253 + 0.07055432072184756 \* IIZIIIIIIIZIIZIII  
254 + 0.08470391802239534 \* IIIZIIIIIZIIZIII  
255 + 0.05665606755281972 \* IIIIIZIIZIIZIII

256 + 0.053929860773588405	*	IIIIIZIIZI
257 + 0.06022550139954594	*	IIIIIIIZIZI
258 + 0.06022550139954594	*	IIIIIIIZZI
259 + 0.004360552555030484	*	IIIIYZZYZ
260 + 0.004360552555030484	*	IIIIIXZZXZ
261 + 0.011733623912074194	*	IIYYIIIIIZ
262 + 0.011733623912074194	*	IIXXIIIIIZ
263 - 0.03305872858775587	*	YZZYIIIIIZ
264 - 0.03305872858775587	*	XZZXIIIIIZ
265 + 0.0127339139792953	*	YZZYIIIIIZ
266 + 0.0127339139792953	*	XZXIIIIIZ
267 - 0.30101532158947975	*	IIIIIIIIIZ
268 + 0.11433954684977561	*	ZIIIIIIIZ
269 + 0.06878552428444665	*	IZIIIIIIIZ
270 + 0.06878552428444665	*	IIZIIIIIZ
271 + 0.05665606755281972	*	IIIZIIIIIZ
272 + 0.12357087224898464	*	IIIIZIIIIIZ
273 + 0.08360121967246183	*	IIIIIZIIIZ
274 + 0.06278876343471208	*	IIIIIZIIZ
275 + 0.06278876343471208	*	IIIIIIIZIZ
276 + 0.053621410722614865	*	IIIIIIIZZ