

Efficiently Trotterizing Lithium Time Evolution Step

Experimental Project Classiq

David Ponarovsky

July 29, 2023

1 Preamble.

This document. On May-2022, Classiq, a pioneer quantum computing company, has launched the first (at least at that scale) competition in quantum circuits programming. The purpose of this document is to describe a proposal solution for the Lithium simulation problem. Hence it is important to notice that even that this paper has been written in L^AT_EX it is not an academic paper, that is, this paper might contain errors (and grammar mistakes for sure). Yet a lot of energy has been spent to bring that work to what it is, so the case that entire of it turned out to be completely nonsense probably won't happen.

The paper is organized in the following systematic 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 Hamiltonian terms), the naive solution, and how much improvement we could hope to make.

Then in the 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 to present each of the local terms as subgates.

In contrast, the third and the fourth sections review methods to determine an order which is clearly superior to the naive approach. The fourth section presents a concept of analyzing the "product graph" a concept which could be thought of as the second order analysis 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.

Naive approach. Before diving into the technical details let's review our competitor first. Consider the straightforward assignment Lie-Trotter [\[Tro59\]](#) just handle each of the Hamiltonian terms one by one. Assume the Hamming weight of the support of H_i is equal to d_i . So we pay two steps to rotate each wire into the parity basis and uncompute it in the end. Then we could apply the CX from each qubit in the support to a chosen one which will sum the wires' parity and finally will propagate the sum through the $RZ(\theta\Delta t)$ gate (rotation by the coefficient of the term and the step size). So in total we pay $(2+2d_i)$ for each term and therefore 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 Δt step of the Lie-Trotter formula [\[Tro59\]](#). In order to get a more solid feeling about what is a good solution, let's assume for the moment that $\mathbb{E}_{\sim i}[H] > 5$, in that case we obtain that the naive approach obtains a circuit at depth $\geq 2 \cdot 276 \cdot 6 \sim 3100$. In the end we will see that our solution yields a 1759-depth circuit.

Characterize our LiH. A native question that one might ask is whether exists a more efficient way to presents the Hamiltonian. So even we can't a sign that there is not, let's try to estimate how mach improvement can be achived by applying the Kitaev transformation. In one sentence, Kitaev has showed that the ladder and the annihilation operators has pauli representation such each operator touch at least $\log(n)$ of the qubits. By the fact that Hamiltonian in the energy base has "product" of four operators (i.e $a_i^\dagger b_j^\dagger a_n b_m$) we get that there are might be operators in the computing base which touch $4\log(n)$ qubits. In our case, $n = 10 \Rightarrow 4\log(n) > n$. Bottom line, It seems that the known techniques for reducing asymptotically the support of the terms don't work here. Hence, it's also an hint that 10-factor improvement is unlikely. Therefore we will aim to 2-factor of improvement over the naive.

2 Single Term Heuristics.

Main wire principle. From now on we will call to the wire which sum the parity of the state the main wire. For given term we have chose the main wire to be the median of the wire in it's support. For example consider the following term:

$$XXXXIII$$

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

Upper and lower summations. We summing the parity as follow, denote by $i_1, i_2, \dots, i_W, \dots, i_k \subset [n]$ the indices in the support of given term, such that i_W is the main wire. Then after rotate the wires into the phase base then we summing the parity of i_1, i_2, \dots, i_{W-1} into i_{W-1} , That it, We apply a $CX(i_j, i_{W-1})$ for $j < W - 1$.

As all the segements $[i_j, i_{W-1}]$ disjoint to any segements of the form $[i_{W+1}, i_j]$ for $j > W + 1$ then we could sum the parity of $i_{W+1}, i_{W+2}, \dots, i_k$ wires into i_{W+1} in parallel. It's easy to see that this heuristic cut by almost half the depth of single term.

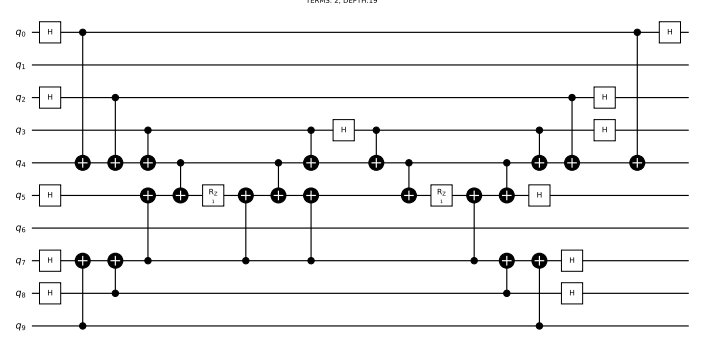


Figure 1: Demonstration of the above methods, applied over the following terms $XIXZZXIXXZ$ $XIXXZXIXXZ$. The fifth wire (qubit) is the main wire which sum the parity.

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 weight greater than w (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 be 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 imposed such they will be computed in parallel.

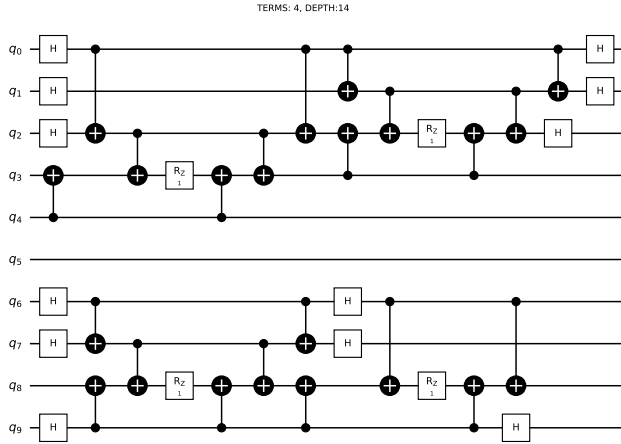


Figure 2: Example for a case in which chaining terms with high distance reduce the depth of the circuit. Here the terms are : $XIXZZIIII XXXZIIII IIIIIIXZX IIIIIIZIX$.

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:

$$\begin{aligned} E' &= \{ \{(u, v), (w, z)\} : (u, v), (w, z), (u, z), (w, v) \in \mathbf{SP} \} \\ w(e) &= w(\{(u, v), (w, z)\}) = \max w(u, w), w(v, z) \end{aligned}$$

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 generalized 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).

```

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_circut(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( "IIIXIIZIZX", 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

```