Efficiently Trotterizing Lithium Time Eveloution Step Experimential 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 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 choice than to perform 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 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 Hamiltoinan, including the quantities regime (i.e number of qubits vs number of hamiltonians terms), the naive solution, and how mach 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.

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 Hamiltonian terms one by one. Assume the the hamming weight of the support of H_i equal to d_i. so we pay two steps to rotate each wire into the parity base 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 propgate the sum trough 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 compute a single Δt step of Lie-Trotter formula[Tro59]. In order to getting a more solid feeling about what is good solution, let's assume for the moment that $\mathbb{E}_{\sim i}[H] > 5$, in that case we obtain that the naive approach obtain 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 $i_1, i_2, ...i_W, ...i_k \subset [n]$ 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 $i_1, i_2, ...i_{W-1}$ 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 segments of the form $[i_{W+1}, i_j]$ for $j > W+1$ then we could sum the parity of $i_{W+1}, i_{W+2}..i_k$ wires into i_{W+1} in parallel. It's easy to see that this heuristic cut by almost half the depth of single term.

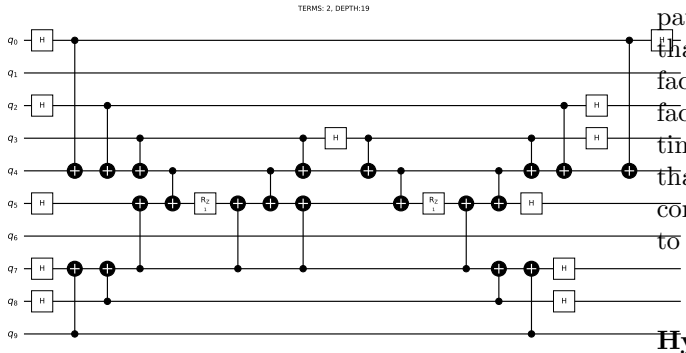

Figure 1: Demonstration of the above methods, applied over the following terms $XIXZZXIXXZ$ $XIXXZXIXXZ$. The fifth wire (qubit) is the main wire which sum the parity.

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 $G \neq$ **do**
  $T \leftarrow$ Greedy Spanning Tree (G)
  $D_i \leftarrow$ Diameter (T)
  $G \leftarrow G/Di$
end
return $D_0 ... D_l$

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 $\mathbf{OPT} \leq 2w\,(\text{MST})$. But also it easy to see that every path which contain all the vertices has weigth greater than $w\,(\text{MST})$, Hence $2w\,(\text{MST}) \leq 2\mathbf{OPT}$. 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 $D^{\text{naive}} \geq 4\mathbf{OPT}$ 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.

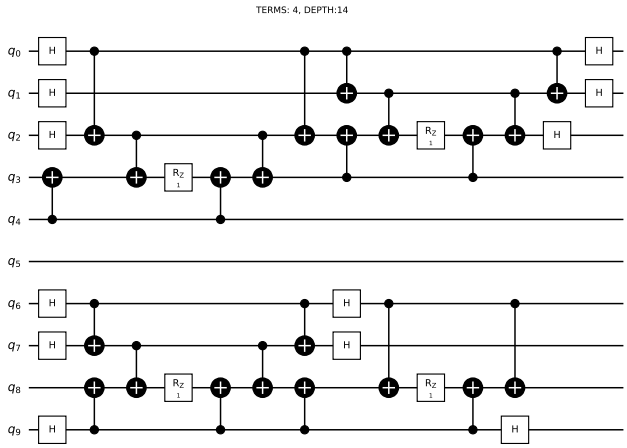

Figure 2: Example for a case in which chaining terms with high distance reduce the depth of the circuit. Here the terms are : $XIXZZIIII$ $XXXZIIIII$ $IIIIIXXZX$ $IIIIIIZIZX$.

Technically we have separated those groups by follow, first we have sorted the term collection by the follow weight functions f, g such that:

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

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 a count 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).

```

from importlib.resources import path
from math import ceil
from struct import unpack
from qiskit import  QuantumCircuit
from qiskit.visualization import circuit_drawer
from matplotlib import pyplot as plt
import datetime
import numpy as np

N                       = 4
WIRES                   = 10
TEST                    = "NOT"
RANDOMSTACTICS          = True
STRATEGY                =  "PRODUCT"
OUTSIDE_OF_CIRCUIT      = WIRES + 1
PATHFILE                =  {
                            "NOT" : "./LiH",
                            "YES" : "./LiH_test",
                            "PRODUCT" : "./LiH_test_product_space"
                            }[TEST]


class local_Hamiltonian():
    def __init__(self, tensor, weight) -> None:
        self.tensor = tensor
        self.weight = weight
        self.parent = self

    def tensorspace(self, other) -> bool:
        for A,B in zip( list(self.tensor), list(other.tensor)):
            if "I" not in [A, B]:
                return False
        return True

    def dis(self, other) -> int:
        ret = 0
        for A,B in zip(list(self.tensor), list(other.tensor)):
            if A != B:
                ret += 1
        return ret

    def solid_product(self, other):
        indices = []
        for j in range(1,WIRES):
            l= "X" * j + "I" * (WIRES-j)
            r = "I" * j + "X" * (WIRES-j)
            if local_Hamiltonian(l,0).tensorspace(self) and\
                local_Hamiltonian(r,0).tensorspace(other):
                indices.append(j)

        if len(indices) > 0:
            return True, indices
        return False, []

    def newbase(self, perm):
        tensor = [ "" ] * len(self.tensor)
        for i in range(len(perm)):
            tensor[i] = self.tensor[perm[i]]
        ret = local_Hamiltonian(tensor,self.weight)
        ret.parent = self.parent
        return ret

```

```
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 seconed_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 reqursive_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 = reqursive_manner(tensor, wire-1, weight, last_wire, -1, first_not_trival = True)
145             LD, RD = reqursive_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 reqursive_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 = reqursive_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
192                 uncompute[pauli](circuit_right)(wire)
193             else:
194                 circuit_right = R
195             return circuit_left, circuit_right
196
197     circuit = circuit.compose(reqursive_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 follwed 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
```

```
            if i > 0:
                for _term in terms[i-1::-1]:
                    if _term.tensor[_j] != 'I':
                        last_terms.append( (_term.tensor[_j], _term.seconed_wires() ))
                        found = True
                        break
            if not found:
                last_terms.append( ('I', OUTSIDE_OF_CIRCUIT  ))

        circuit = MulByterm(circuit, term, main_wire=main_wire,
         next_terms=next_terms, last_terms=last_terms )

    return  circuit

def genreate_optimzed_circut(circuit, terms, svg =False, entire = False):
    circuit = cutting(cutting(circuit))

    if entire:
        for _ in range(ceil(np.log(N))):
            circuit = circuit.compose(circuit)
        circuit = cutting(circuit)

    print(f"TERMS: {len(terms)}, DEPTH:{circuit.depth()}")

    if svg:
        circuit_drawer(circuit, output='mpl',style="bw", fold=-1)
        plt.title( f"TERMS: {len(terms)}, DEPTH:{circuit.depth()}")
        plt.tight_layout()
        plt.savefig(f'Ham_{STRATEGY}-{datetime.datetime.now()}.svg')

    if entire:
        open(f"Ham_{STRATEGY}-{datetime.datetime.now()}.qasm", "w+").write(circuit.qasm())

    return circuit.depth()
```

```python
from copy import deepcopy
import networkx as nx
from Hamiltonian_parser import WIRES, parser, local_Hamiltonian,genreate_circut,genreate_optimzed_circut
from itertools import permutations, product, combinations
from random import choice
import pickle as pkl

from matplotlib import pyplot as plt


class Permutation_Base():
    def __init__(self, arr) -> None:
        self.arr = arr
        self.parent = self


def generated_the_product_graph(terms = parser()):
    def generated_the_product_graph_by_base(_terms, number_premu=0):

        G        = nx.Graph()
        Gproduct = nx.Graph()

        edges_set = set()

        for (H1, H2) in product(_terms, _terms):
            if  H1.tensorspace(H2):
                G.add_edge(H1, H2)
                edges_set.add( (H1,H2) )
                G.edges[(H1, H2)]['weight'] = H1.dis(H2)
                G.edges[(H1, H2)]['solid'] = True
                G.edges[(H1, H2)]["permutation"] = j

        for e in G.edges():
            H1,H2 = e
            for H3,H4 in \
                product(list(G.adj[H1]),list(G.adj[H2])):
                if (H1,H4) in edges_set and\
                    (H3,H4) in edges_set and\
                        (H2,H3) in edges_set:
                    Gproduct.add_edge((H1,H2), (H3, H4))
                    Gproduct.edges[(H1, H2), (H3, H4)]['weight'] =  max(H1.dis(H3), H2.dis(H4))
                    Gproduct.edges[(H1, H2), (H3, H4)]['solid'] = True
                    Gproduct.edges[(H1, H2), (H3, H4)]["permutation"] = j

        for (H1, H2) in product(_terms, _terms):
            if (H1,H2) not in edges_set:
                G.add_edge(H1, H2)
                G.edges[(H1, H2)]['weight'] = H1.dis(H2)
                G.edges[(H1, H2)]['solid'] = False
                G.edges[(H1, H2)]["permutation"] = j

        print(f"vertices:{Gproduct.number_of_nodes()}\t edges: ~{Gproduct.number_of_edges()}")
        return Gproduct, G,  _terms

    return pkl.load( open(f"mainG-276-1.pkl", "br"))

    permutations = list(map(lambda x: Permutation_Base(x) , [
        # [0,2,4,6,8,1,3,5,7,9],
        [0,1,2,3,4,5,6,7,8,9]
        # [0,7,4,6,8,9,3,5,2,1]
    ]))
    graphs = []
    perm_terms = []
```

```
    mainG = nx.Graph()
    mainProductG = nx.Graph()
    for j, permutation in enumerate(permutations):
        perm_terms = list(map( lambda x : x.newbase(permutation.arr), terms))
        productG, G, _ = generated_the_product_graph_by_base(perm_terms, number_premu=j)

        if mainG.number_of_nodes() == 0:
            mainG = nx.compose(mainG, G)
        mainProductG = nx.compose(mainProductG, productG)

    pkl.dump((mainG, mainProductG, terms ,permutations), open(f"mainG-{len(terms)}-
{len(permutations)}.pkl", "bw+"))
    return mainG, mainProductG, terms, permutations

def select(_list, v, G, flag = True):
    minimal = min(_list, key =lambda u : G.edges[v,u]['weight'] )
    return choice( [r for r in _list if G.edges[v,r]['weight'] == \
         G.edges[v,minimal]['weight'] ])

def notcolorized(node, _set):
    if isinstance(node, tuple):
        term1, term2 = node
        if (term1.parent in _set) or (term2.parent in _set):
            return False
        return True
    else:
        return node.parent not in _set

def colirize(node, _set):
    if isinstance(node, tuple):
        term1, term2 = node

        if (term1.parent in _set) or (term2.parent in _set):
            return  False, _set, 1
        _set.add(term1.parent)
        _set.add(term2.parent)
        return True
    else:
        if node.parent in _set:
            return False
        else:
            _set.add(node.parent)
            return True


#randomized DFS.
def sample_path(G, terms) -> tuple((nx.Graph, set)):
    print("sample_path")
    color = set()

    def DFS(v, T, _color, sign=0, flag =True):

        if not colirize(v, _color):
            return _color, sign

        can_packed = [None]
        while len(can_packed) > 0 :
            can_packed = list(filter(lambda x :\
                notcolorized(x, _color), G.adj[v]))

            if len(can_packed) > 0:
                u =  select( can_packed, v, G, flag=flag)
                T.add_edge(v,u)
                if 'permutation' in G.edges[v,u]:
```

```
                    T.edges[v,u]['permutation'] = G.edges[v,u]['permutation']
                T.edges[v,u]['sign'] = sign
                _color, sign = DFS(u, T, _color, sign=sign+1, flag=flag)
        return _color, sign

    T = nx.DiGraph()
    l = []

    for v in G.nodes():
        T.add_node(v)
        l.append(v)

    _sign = 0
    for v in G.nodes():
        color, _sign = DFS(v, T, color, sign=_sign, flag=False)

    return T, color

def get_Diameter(Tree: nx.Graph) -> tuple((tuple(([], int)),tuple(([], int)))) :

    def DFS_tree_depth(G, v):

        if len(list(G.adj[v])) == 0 :
            return (([v],1),([v],1))

        branches = []
        maxinnerpath, maxinnerdepth = [],0
        for u in list(G.adj[v]):
            ((temppath, tempdepth), \
                (tempinnerpath, tempinnerdepth)) = DFS_tree_depth(G,u)

            if maxinnerdepth <  tempinnerdepth:
                maxinnerpath, maxinnerdepth = tempinnerpath, tempinnerdepth

            branches.append((temppath, tempdepth))

        maxpath, maxdepth = [],0

        for (b1, d1),(b2, d2) in combinations(branches, r=2):
            if 1 + d1 + d2 > maxinnerdepth:
                maxinnerpath    =  b1 + [ v ] + b2
                maxinnerdepth   = 1 + d1 + d2

        for b,d in branches:
            if 1 + d > maxdepth:
                maxpath = [v] + b
                maxdepth = 1 + d
        return ((maxpath,maxdepth), (maxinnerpath, maxinnerdepth))

    ((maxpath,maxdepth), (maxinnerpath, maxinnerdepth)) = DFS_tree_depth(Tree, list(Tree.nodes)[0])
    print(maxdepth, maxinnerdepth)
    return maxpath if maxdepth > maxinnerdepth else maxinnerpath


def generate_simple_graph(_terms):
    G = nx.Graph()
    for (H1, H2) in product(_terms, _terms):
        G.add_edge(H1, H2)
        G.edges[(H1, H2)]['weight'] = H1.dis(H2)
    return G

def greedy_path( terms ):
    G = generate_simple_graph(terms )

```

```
    def reqursive_form( _terms ):
        if len(_terms) < 3:
            return _terms
        else:

            T, _ = sample_path(G, _terms)

            Q = get_Diameter(T)
            color = set()

            ret = []
            for H in Q:
                if H.parent not in color:
                    color.add(H.parent)
                    ret.append(H.parent)
                    G.remove_node(H)

            remain_terms = [ term for term in _terms if term not in color ]
            return ret + reqursive_form(remain_terms)
    return reqursive_form(terms)

def Hamiltonian_sorting(hamiltonians):
    groups = [[] for _ in product(range(WIRES),range(WIRES))]
    for term in hamiltonians:
        x,y = term.seconed_wires()
        groups[x + WIRES*y].append(term)

    ret = [ ]
    for group in groups:
        group =  greedy_path(group)
        ret += group
    return ret

def enforce_seapration(hamiltonians):

    def seperate(terms):

        def sort_tensor_by_geometrical_support(tensor, up = True):


            for j,pauli in enumerate( { True: tensor, False: reversed(tensor) }[up] ):
                if pauli != "I":
                    if up:
                        return 10 - j
                    else:
                        return 10 - j
                else:
                    return 10

        above   = sorted(terms,\
             key=lambda x : sort_tensor_by_geometrical_support(x.tensor, up=True))
        beneath = sorted(terms,\
             key=lambda x : sort_tensor_by_geometrical_support(x.tensor, up=False))

        contact_point = min( range(len(terms)),\
            key = lambda i : above[i].tensorspace(beneath[i]))

        print(f"contact point: {contact_point}")
        path = []
        for x,y in zip( sorted(above[:contact_point], key = lambda z : z.tensor),\
            sorted(beneath[:contact_point], key = lambda z : z.tensor)):
            path.append(x)
            path.append(y)

```

```
        new_terms = []
        for x in above[contact_point:]:
            if x not in beneath[:contact_point]:
                new_terms.append(x)
        return new_terms, path
    path = [ ]
    terms ,temppath =  seperate(deepcopy(hamiltonians))
    path += temppath

    print(f"terms:{len(terms)}")
    return path, terms

def alternate_path_v2(mG : nx.Graph, G : nx.Graph,\
     terms, permutations, single_iteration = False):
    other_color = set()

    T, _ = sample_path(G, terms)
    Q = get_Diameter(T)
    ret = []

    for (u,v) in Q:
        for H in [u,v]:
            if H not in other_color:
                other_color.add(H)
                ret.append(H)

    made_progress = True
    _single_iteration = True
    while _single_iteration and (len(Q) > 2 and made_progress):
        made_progress = False
        for u,v in Q:
            for H in [u,v]:
                if H.parent not in other_color:
                    ret.append(H)
                    other_color.add(H.parent)
                for w in list(G.nodes()):
                    if H in w:
                        G.remove_node(w)
                        made_progress = True

                if mG.has_node(H):
                    mG.remove_node(H)

        T, _ = sample_path(G, terms)
        if T.number_of_edges() > 1:
            Q = get_Diameter(T)
        else:
            break
        _single_iteration = not single_iteration

    if not single_iteration:
        for term in terms:
            if term.parent not in other_color:
                ret.append(term)

    return ret, terms, other_color

def main_enforce(hamiltonians):
    path, terms = enforce_seapration(hamiltonians)
    path +=  Hamiltonian_sorting(terms)
    circuit = genreate_circut(path)
    depth = genreate_optimzed_circut(circuit, hamiltonians, svg=False, entire=False)
    return circuit

```

```python
def compose_alternate_enforce():
        G, mainProductG, terms, permus = generated_the_product_graph()
        canidates = [ ]
        for _ in range(5):
            path, terms, color = alternate_path_v2( deepcopy(G), deepcopy(mainProductG), terms, permus,
single_iteration=True)
            circuit = genreate_circut(path)
            genreate_optimzed_circut(circuit, terms, svg=False, entire=False)
            remain_terms = [  term for term in terms if term not in color  ]
            circuit = circuit.compose(  main_enforce( remain_terms ) )
            depth = genreate_optimzed_circut(circuit, terms, svg = False, entire=False )
            canidates.append( (depth, circuit) )
            print(f"DEPTH: {depth}")
        depth, circuit =  min( canidates, key = lambda x : x[0] )
        depth = genreate_optimzed_circut(circuit, terms, svg = False, entire=True)
        # depth = genreate_optimzed_circut(circuit, terms, entire=True)
        return circuit,terms



def demonstrate_fig( ):
    path = [ local_Hamiltonian( "XIXZZIIIII", 0.5 ),
      local_Hamiltonian( "XXXZIIIIII", 0.5 ),
      local_Hamiltonian( "IIIIIIXXZX", 0.5 ),
      local_Hamiltonian( "IIIIIIZIZX", 0.5 ),
      local_Hamiltonian( "IIIXIIZIZX", 0.5 ) ]

    path, terms = enforce_seapration(path)
    circuit =  genreate_circut(path)
    genreate_optimzed_circut(circuit ,path, svg=True, entire=False)


if __name__ == "__main__":

    circuit, terms  = compose_alternate_enforce()
```