```python

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

```

```
    def median(self):
        support = list(filter( lambda x : self.tensor[x] != 'I', range(WIRES)))
        if len(support) != 0:
            return support[int(len(support)/2)]
        return 0

    def seconed_wires(self):
        j = self.median()

        def find_nearset_nontrival(sign):


            gen = [ (i,v) for i,v in list(enumerate(self.tensor))[j+1:]] if sign ==1\
                 else [ (i,v) for i,v in list(enumerate(self.tensor))[j-1::-1]]

            for i,v in gen:
                if v != "I":
                    return i
            return j

        pos_wire = j
        neg_wire = j
        if j + 1 < WIRES:
            pos_wire = find_nearset_nontrival(1)
        if j - 1 >= 0:
            neg_wire = find_nearset_nontrival(-1)

        return neg_wire, pos_wire

def parser_line(line) -> local_Hamiltonian:
    line  = line.split()
    return local_Hamiltonian( list(line[-1]),
     { "-" : -1 , "+" : 1 }[line[0]] * np.float64(line[1]) / N )

def parser() -> None:
    hamiltonians = [ ]
    for line in open(PATHFILE).readlines():
        if len(line) > 1:
            hamiltonians.append(parser_line(line))
    return hamiltonians

def donothing(_):
    pass

def rotateY(cir):
    def _func(wire):
        cir.s(wire)
        cir.h(wire)
    return _func

def unrotateY(cir):
    def _func(wire):
        cir.h(wire)
        cir.sdg(wire)
    return _func

def MulByterm (circuit : QuantumCircuit, term ,next_terms = [], last_terms =  [],
 main_wire = WIRES-1) -> QuantumCircuit:

    def reqursive_manner(tensor, wire, weight, last_wire, _sign, first_not_trival=True):

        if wire < 0 or wire == WIRES:
            return QuantumCircuit(WIRES),QuantumCircuit(WIRES)

```

```
        compute =   {
            "X" : lambda cir : cir.h,
            "Y" : lambda cir : rotateY(cir),
            "Z" : lambda cir : donothing,
            "I" : lambda cir : donothing }


        uncompute = {
            "X" : lambda cir : cir.h,
            "Y" : lambda cir : unrotateY(cir),
            "Z" : lambda cir : donothing,
            "I" : lambda cir : donothing  }

        pauli = tensor[wire]
        if wire == main_wire:
            circuit_node = QuantumCircuit(WIRES)
            LU, RU = reqursive_manner(tensor, wire-1, weight, last_wire, -1, first_not_trival = True)
            LD, RD = reqursive_manner(tensor, wire+1, weight, last_wire,  1, first_not_trival = True)

            compute[pauli](circuit_node)(wire)
            for L in [ LD, LU]:
                circuit_node = circuit_node.compose(L)
            circuit_node.rz(2*weight, main_wire)
            for R in [ RD, RU]:
                circuit_node = circuit_node.compose(R)
            uncompute[pauli](circuit_node)(wire)
            return  circuit_node

        if pauli == 'I':


            # temp_wire = last_wire + _sign if first_not_trival else last_wire
            return reqursive_manner(tensor, wire + _sign, weight,
             last_wire, _sign, first_not_trival = first_not_trival)

        else:

            temp_wire = wire if first_not_trival else last_wire
            parity_collector = False
            if first_not_trival:
                parity_collector = True


            L, R = reqursive_manner(tensor, wire + _sign, weight, temp_wire, _sign, first_not_trival = False)
            circuit_left, circuit_right = QuantumCircuit(WIRES),QuantumCircuit(WIRES)

            if (parity_collector) or not (\
                 (last_terms[wire][0] == pauli) and\
                  (last_terms[wire][1][ { 1 : 1 , -1 : 0  }[_sign] ] == last_wire)):

                compute[pauli](circuit_left)(wire)
                circuit_left = circuit_left.compose(L)
                circuit_left.cx(wire, last_wire)

            else:
                circuit_left = L

            if  (parity_collector) or not (
                 (next_terms[wire][0] == pauli) and\
                  (next_terms[wire][1][ { 1 : 1 , -1 : 0  }[_sign] ] == last_wire)):

                circuit_right.cx(wire, last_wire)
                circuit_right = circuit_right.compose(R)
```

```

                uncompute[pauli](circuit_right)(wire)
            else:
                circuit_right = R
            return circuit_left, circuit_right

    circuit = circuit.compose(reqursive_manner(term.tensor, main_wire, term.weight, main_wire, 0))
    return circuit


def cutting(circuit : QuantumCircuit):
    '''Second optimization, cuts the gates which are follwed by their
    uncompute '''
    def filter_by_wire(wire):
        return list(filter( lambda item :\
            any( [register.index == wire for register in item[1][1]] ), enumerate(circuit.data) ))


    UNCOMPTE = {
        "h"  : "h",
        "cx" : "?",
        "rz" : "?" ,
        "s"  : "sdg",
        "sdg" : "s",
        "sxdg" :"?" }

    indices_todelete = []
    for wire in range(WIRES):
        operators = filter_by_wire(wire)

        j = 0
        while (j < len(operators) - 1 ):
            if ( UNCOMPTE[operators[j][1][0].name] == operators[j+1][1][0].name ):
                indices_todelete.append(  operators[j][0]  )
                indices_todelete.append(  operators[j+1][0]  )
            j += 1

    for index in reversed(sorted(indices_todelete)):
        circuit.data.pop(index)

    return circuit


def genreate_circut(terms = None):
    circuit = QuantumCircuit(WIRES)
    terms = parser() if terms == None else terms
    print(len(terms))
    for i, term in enumerate(terms):

        main_wire =  term.median()
        next_terms,last_terms = [],[]

        for _j in range(WIRES):
            found = False
            if i+1 < len(terms):
                for _term in terms[i+1:]:
                    if _term.tensor[_j] != 'I':
                        next_terms.append( (_term.tensor[_j], _term.seconed_wires() ))
                        found = True
                        break
            if not found:
                next_terms.append( ('I', OUTSIDE_OF_CIRCUIT ))

            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 = []
```

```python
    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]:
```

```python
                    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)

```

```python
        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

```

```
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()
```