Procedurally Optimised ZX-Diagram Cutting

This Jupyter notebook is supplementary to the paper on Procedurally Optimised ZX-Diagram Cutting (2024), by Matthew Sutcliffe and Aleks Kissinger

Initialisation

In [1]:
```
#pip install pyzx
```

In [2]:
```
import pyzx as zx

import sys, os, math
import random
import sympy as sym
from fractions import Fraction
from pyzx import print_matrix
from pyzx.basicrules import *
```

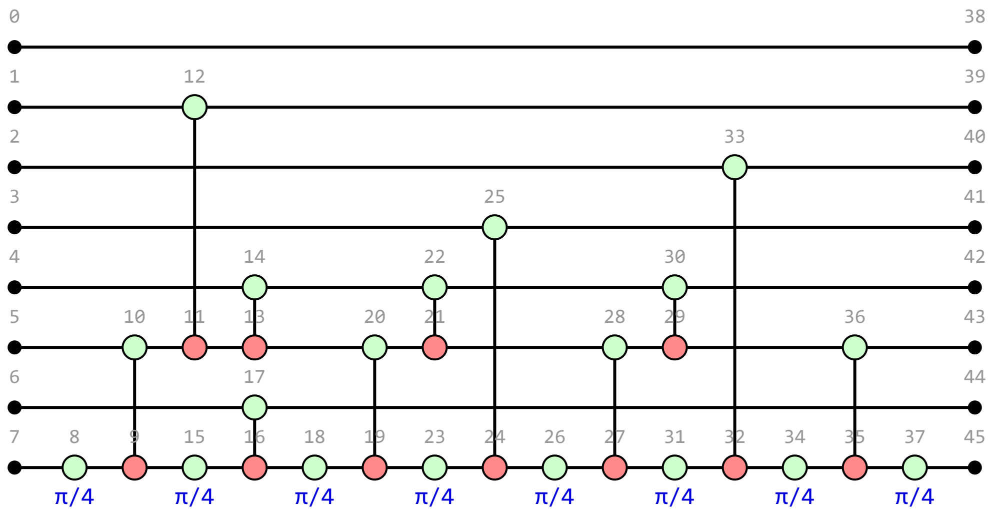
Introduction with example circuits

Example 1

Let's generate an example circuit and partially simplify it, while keeping its structure graph-like...

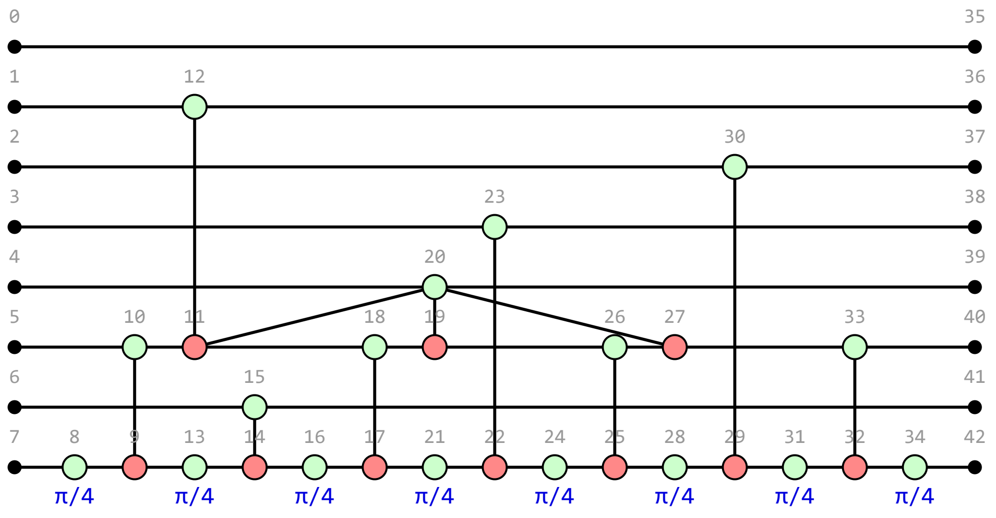
In [3]:
```
strCirc = """
qreg q[8];
rz(0.25*pi) q[7];
cx q[5], q[7];
cx q[1], q[5];
cx q[4], q[5];
rz(0.25*pi) q[7];
cx q[6], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];
cx q[4], q[5];
rz(0.25*pi) q[7];
cx q[3], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];
cx q[4], q[5];
rz(0.25*pi) q[7];
cx q[2], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];
rz(0.25*pi) q[7];
"""

c = zx.qasm(strCirc)
g = c.to_graph()
#g.normalize()
```

```
zx.draw(g, labels=True,scale=30)
print("T-count = ", zx.tcount(g))
```



```
T-count =  8
```

In [4]:
```
zx.basicrules.fuse(g,22,14)
zx.basicrules.fuse(g,22,30)

zx.basicrules.fuse(g,11,13)

# This just re-numbers the vertex indexes to remove any gaps:
h = g.copy()
g = h.copy()

zx.draw(g, labels=True,scale=30)
print("T-count = ", zx.tcount(g))
```



```
T-count =  8
```

As outlined in the paper, we can define a procedure to compute the weights of the vertices (at the base tier)...

In [5]:
```
def compFirstTier(g,showWeights=True):
    vweights = [0]*(max(g.vertices())+1)
    vDoorsteps = [set() for i in range(max(g.vertices())+1)] # Doorstep nodes are T
    vCtrls = [set() for i in range(max(g.vertices())+1)] # This keeps track of the
    for v in g.vertices(): # Be careful as g.vertices isn't normalised
        vweights[v] = 0
        #if g.phase(v) in (0.25, 0.75, 1.25, 1.75): vweights[v] = 1 # If the vertex

    for v in g.vertices():
        if g.type(v) == 2 and g.phase(v) in {0,1}: # If red vertex of n*pi-phase
            for neigh in g.neighbors(v):
                if g.qubit(neigh) != g.qubit(v): # If the neighbour is not on the s
                    vCtrls[v].add(neigh) # Keep track of the CNOT's control spiders

    for v in g.vertices():
        if g.type(v) == 2 and g.phase(v) in {0,1}: # If red vertex of n*pi-phase
            Tneighs = 0
            countedNeighs = set() # Keeps track of which T-gates have already been
            for neigh in g.neighbors(v):
                if g.qubit(neigh) == g.qubit(v): # If the neighbour is on the same
                    if g.phase(neigh) in (0.25, 0.75, 1.25, 1.75): # If the neighbo
                        if not(neigh in countedNeighs): # If the neighbour hasn't c
                            Tneighs += 1
                            countedNeighs.add(neigh)

            #print()
            #print(v, "\t", Tneighs)
            #print(countedNeighs)

            if Tneighs == 2 and len(vCtrls[v])>0: # the second check is a temporary
                #if (len(vCtrls[v])==0): print("\n\nERROR at vertex",v); zx.draw(g,
                w = 2/(len(vCtrls[v])) # If this gives div-by-0 error, then partial
                for ctrl in vCtrls[v]:

                    isNew = True
                    for i in countedNeighs:
                        if (i in vDoorsteps[ctrl]): isNew = False

                    if isNew:
                        for i in countedNeighs: vDoorsteps[ctrl].add(i)
                        vweights[ctrl] += w

    #-----

    tier = 1
    vtiers = newvweights = [0]*(max(g.vertices())+1)

    # Weightings for lowest tier:
    if (showWeights): print("--== TIER 1 ==--\nvertex \t weight\t children")
    for i in range(len(vweights)):
        if (len(vDoorsteps[i]) > 0):
            w = vweights[i]
            #if (g.phase(i) in [0.25,0.75,1.25,1.75]): w += 1 # +1 if the vertex it
            if (showWeights): print(i, "\t", w, "\t", vDoorsteps[i])
            vtiers[i] = tier
    if (showWeights): print()

    vweights_0 = vweights.copy()
    vDoorsteps_0 = vDoorsteps.copy()

    vweights_max = vweights.copy()
    #vDoorsteps_max = vDoorsteps.copy()

    return [tier,vweights_max,vweights,vtiers,vDoorsteps]

tierData     = compFirstTier(g)
tier         = tierData[0]
vweights_max = tierData[1]
```

```
vweights     = tierData[2]
vtiers       = tierData[3]
vDoorsteps   = tierData[4]
```

```
--== TIER 1 ==--
vertex   weight   children
10       2.0      {8, 13}
15       2.0      {16, 13}
18       2.0      {16, 21}
23       2.0      {24, 21}
26       2.0      {24, 28}
30       2.0      {28, 31}
33       2.0      {34, 31}
```

Here, we observe 7 vertices with weights of 2.0. These are the vertices which, if cut, would enable 2 T-like gates to reduce to Clifford (hence reducing the T-count by 2). (We note also which T-like vertices they are preventing from reducing in this way.)

When looking through the scope of the NEXT tier, we're considering cuts as prerequesites to those identified in the base tier. In other words, we're considering vertices which, rather than blocking T-like children from fusing, are blocking tier-1 WEIGHTED vertices from fusing. So, we're looking for vertices which, when cut (along with its subsequently fused children then also being cut), are ultimately likely to be worthwhile in reducing T-count efficiently. And, now that we're requiring a cut on a vertex AND on its to-be fused children, the existing weights AS SEEN BY THE NEXT TIER are halved (though capped to 1.0, meaning an almost certain worthwhile cut). So, the above weightings as seen by the next tier are...

In [6]:
```
# Weightings as seen by next tier:
for i in range(len(vweights)):
    if (len(vDoorsteps[i]) > 0):
        w = min(vweights[i]/2,1.0)
        print(i, "\t", w, "\t", vDoorsteps[i])
```

```
10       1.0      {8, 13}
15       1.0      {16, 13}
18       1.0      {16, 21}
23       1.0      {24, 21}
26       1.0      {24, 28}
30       1.0      {28, 31}
33       1.0      {34, 31}
```

So now let's compute the next tier, similar to the way we did the first (excpet looking for fusing opportunities of weighted vertices rather than T-like vertices)...

In [7]:
```
def compNextTier(g,showWeights,tier,vweights_max,vweights,vtiers):
    isEmptyTier = False
    while (isEmptyTier == False):
        tier += 1
        newvweights = [0]*(max(g.vertices())+1)
        vDoorsteps = [set() for i in range(max(g.vertices())+1)] # Doorstep nodes o
        vCtrls = [set() for i in range(max(g.vertices())+1)] # This keeps track of

        for v in g.vertices(): # Be careful as g.vertices isn't normalised
            vweights[v] = min(vweights_max[v]/2,1.0) # Update lower tiers' values f
            newvweights[v] = 0
            #if g.phase(v) in (0.25, 0.75, 1.25, 1.75): vweights[v] = 1 # If the ve

        for v in g.vertices():
            if g.type(v) == 2 and g.phase(v) in {0,1}: # If red vertex of n*pi-phas
                for neigh in g.neighbors(v):
                    if g.qubit(neigh) != g.qubit(v): # If the neighbour is not on t
                        vCtrls[v].add(neigh) # Keep track of the CNOT's control spi
        for v in g.vertices():
            if g.type(v) == 2 and g.phase(v) in {0,1}: # If red vertex of n*pi-phas
                Tneighs = 0
                countedNeighs = set() # Keeps track of which weighted nodes have al
                for neigh in g.neighbors(v):
                    if g.qubit(neigh) == g.qubit(v): # If the neighbour is on the s
                        if vweights[neigh] > 0: # If the neighbour also is weighted
                            if not(neigh in countedNeighs): # If the neighbour hasn
                                Tneighs += 1
                                countedNeighs.add(neigh)

                #print()
                #print(v, "\t", Tneighs)
                #print(countedNeighs)

                if Tneighs == 2 and len(vCtrls[v])>0: # the second check is a tempo
                    #if (len(vCtrls[v])==0): print("\n\nERROR at vertex",v); zx.dra
                    w = 2/(len(vCtrls[v])) # If this gives div-by-0 error, then par
                    for ctrl in vCtrls[v]:
                        isNew = True

                        for i in countedNeighs: # avoid double-counting
                            if (i in vDoorsteps[ctrl]): isNew = False

                        isPrevTier = False
                        for i in countedNeighs: # at least one must be of the immed
                            if (vtiers[i] == tier-1): isPrevTier = True
                        if (isPrevTier==False): isNew = False

                        if isNew:
                            for i in countedNeighs: vDoorsteps[ctrl].add(i)
                            newvweights[ctrl] += w


        #-----

        if (showWeights): print("--== TIER",tier,"==--")

        isEmptyTier = True
        for i in range(len(vweights)):
            vweights[i] = newvweights[i] # Update vweights from buffer

            if (vweights[i] > vweights_max[i]): vweights_max[i] = vweights[i] # Upd

            if (vweights[i]>0): #if (len(vDoorsteps[i]) > 0):
                if (showWeights): print(i, "\t", vweights[i], "\t", vDoorsteps[i])
                vtiers[i] = tier
                isEmptyTier = False

        if (showWeights):
            if (isEmptyTier): print("(empty)")
            print()
            return [tier,vweights_max,vweights,vtiers,vDoorsteps,False] # If no mor

        return [tier,vweights_max,vweights,vtiers,vDoorsteps,True]

tierData     = compNextTier(g,True,tier,vweights_max,vweights,vtiers)
tier         = tierData[0]
vweights_max = tierData[1]
vweights     = tierData[2]
vtiers       = tierData[3]
vDoorsteps   = tierData[4]
```

```
--== TIER 2 ==--
12      1.0     {10, 18}
20      3.0     {26, 33, 10, 18}
```

If we were to check the NEXT tier (tier 3), we find - in this case - no tier-3 vertex weightings, and so we stop here...

In [8]:
```
compNextTier(g,True,tier,vweights_max,vweights,vtiers);
```

```
--== TIER 3 ==--
(empty)
```

Looking at the total data we have so far, we see...

In [9]:
```
# Total weightings...

def dispWeights(tier,vweights_max,vweights,vtiers):
    print("vertex\t weight\t\t tier")

    for i in range(len(vweights)):
        if (vweights_max[i]>0): #if (len(vDoorsteps[i]) > 0):
            w = vweights_max[i]
            b = 0
            if (g.phase(i) in [0.25,0.75,1.25,1.75]): b+=1 # Add 1 to weight if the
            strB = "(+" + str(b) + ")" #strB = " (" + str(w+b) + ")"
            print(i, "\t", w,strB, "\t", vtiers[i])

dispWeights(tier,vweights_max,vweights,vtiers)
```

```
vertex   weight          tier
10       2.0 (+0)        1
12       1.0 (+0)        2
15       2.0 (+0)        1
18       2.0 (+0)        1
20       3.0 (+0)        2
23       2.0 (+0)        1
26       2.0 (+0)        1
30       2.0 (+0)        1
33       2.0 (+0)        1
```

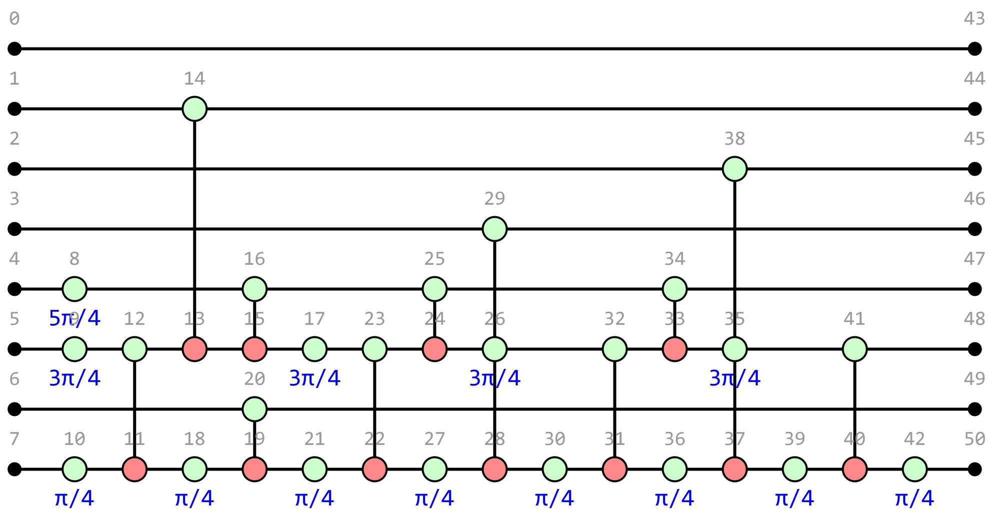
Here, the bonus (+0)/(+1) on the weights are determined by whether that vertex is itself T-like. This is not seen by other vertices when considering the weights of their neighbours (as fusing a couple of tiered-vertices where one is T-like has no extra benefit than if neither were T-like), but is only seen when considering which weight is best to cut first (as cutting a T-like tiered-vertex will remove 1 extra T-gate than if it had not been T-like).

Higher tier vertices are always considered preferentially to lower tier vertices when deciding which to cut next. So, in this example, the initial best cut is on vertex 20, as it is on tier 2 and has the max weight there of 3.0 (even though this cut itself does not reduce the T-count at all).

Before we proceed to the next step, let's look at another (more complex) example, and see if you can work through the how its results are arrived at...

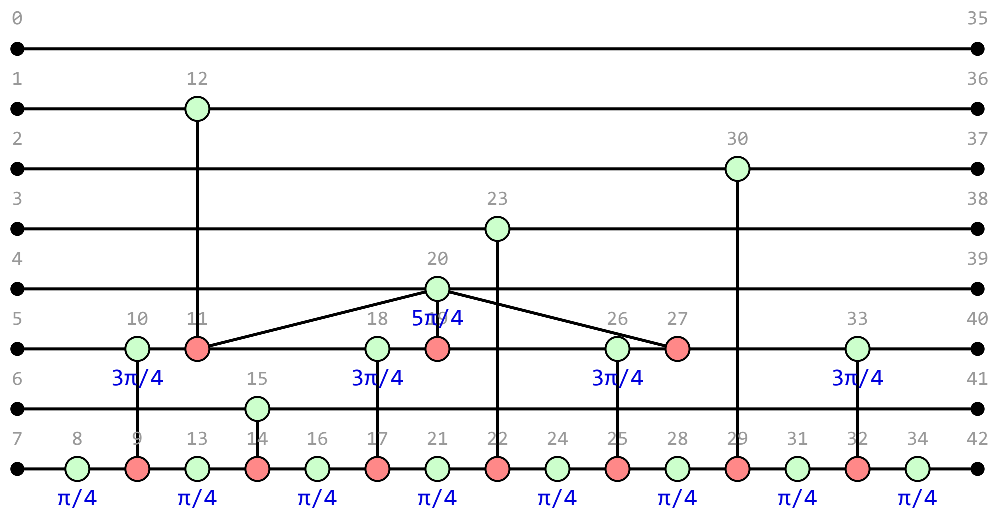
Example 2

In [10]:
```
strCirc = """
qreg q[8];
rz(1.25*pi) q[4];
rz(0.75*pi) q[5];
rz(0.25*pi) q[7];
cx q[5], q[7];

cx q[1], q[5];

cx q[4], q[5];
rz(0.75*pi) q[5];

rz(0.25*pi) q[7];
cx q[6], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];

cx q[4], q[5];
rz(0.75*pi) q[5];

rz(0.25*pi) q[7];
cx q[3], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];

cx q[4], q[5];
rz(0.75*pi) q[5];

rz(0.25*pi) q[7];
cx q[2], q[7];
rz(0.25*pi) q[7];
cx q[5], q[7];
rz(0.25*pi) q[7];
"""

c = zx.qasm(strCirc)
g = c.to_graph()
#g.normalize()
zx.draw(g, labels=True,scale=30)
print("T-count = ", zx.tcount(g))
```



```
T-count =  13
```

In [11]:
```
xx = 2

zx.basicrules.fuse(g,23+xx,14+xx)
zx.basicrules.fuse(g,23+xx,8)
zx.basicrules.fuse(g,23+xx,32+xx)
zx.basicrules.fuse(g,12,9)
zx.basicrules.fuse(g,21+xx,15+xx)
zx.basicrules.fuse(g,30+xx,24+xx)
zx.basicrules.fuse(g,39+xx,33+xx)

zx.basicrules.fuse(g,13,15)

# This just re-numbers the vertex indexes to remove any gaps:
h = g.copy()
g = h.copy()
gOrig = g.copy()
gEx2 = g.copy()

zx.draw(g, labels=True,scale=30)
print("T-count = ", zx.tcount(g))
```



```
T-count =  13
```

In [12]:
```
def bestCut(g,tier,vweights_max,vweights,vtiers):
    maxTier = max(vtiers)
    maxW    = -1.0
    bestV   = -1

    for i in range(len(vweights)):
        if (vweights_max[i]>0 and vtiers[i] == maxTier): #if (len(vDoorsteps[i]) >
            w = vweights_max[i]
            if (g.phase(i) in [0.25,0.75,1.25,1.75]): w+=1 # Add 1 to weight if the
            #print(i, "\t", w, "\t", vtiers[i])
            if (w > maxW):
                maxW  = w
                bestV = i
    return bestV

def compWeights(g,showWeights=True):
    tierData     = compFirstTier(g,showWeights)
    tier         = tierData[0]
    vweights_max = tierData[1]
    vweights     = tierData[2]
    vtiers       = tierData[3]

    #isNextTier   = True
    #while (isNextTier):
    tierData     = compNextTier(g,showWeights,tier,vweights_max,vweights,vtiers)
    tier         = tierData[0]
    vweights_max = tierData[1]
    vweights      = tierData[2]
    vtiers        = tierData[3]
    #isNextTier    = tierData[5]

    vBest         = bestCut(g,tier,vweights_max,vweights,vtiers)
    return [vBest,tier,vweights_max,vweights,vtiers]
```

In [13]:
```
print("BEST CUT: vertex",compWeights(g)[0])
```

```
--== TIER 1 ==--
vertex   weight  children
10       2.0     {8, 13}
12       1.0     {10, 18}
15       2.0     {16, 13}
18       2.0     {16, 21}
20       3.0     {26, 33, 10, 18}
23       2.0     {24, 21}
26       2.0     {24, 28}
30       2.0     {28, 31}
33       2.0     {34, 31}

--== TIER 2 ==--
12       1.0     {10, 18}
20       3.0     {26, 33, 10, 18}

BEST CUT: vertex 20
```

In [14]:
```
dispWeights(tier,vweights_max,vweights,vtiers)
```

```
vertex   weight          tier
10       2.0 (+1)        1
12       1.0 (+0)        2
15       2.0 (+0)        1
18       2.0 (+1)        1
20       3.0 (+1)        2
23       2.0 (+0)        1
26       2.0 (+1)        1
30       2.0 (+0)        1
33       2.0 (+1)        1
```

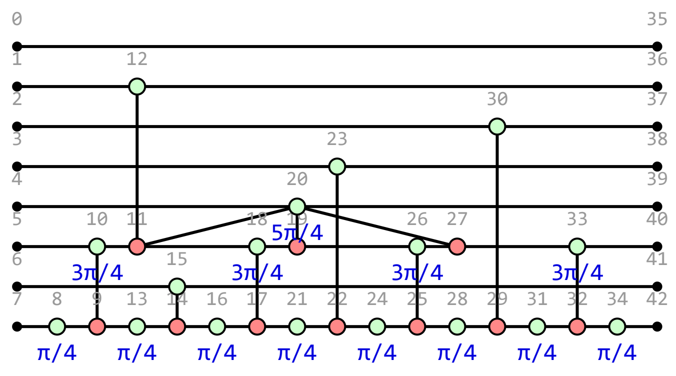
The weightings are a little different, but we again find that vertex 20 is the best initial cut. So, let's make the cut and decompose our graph into 2 new graphs accordingly...

In [15]:
```
def cut(v): #TEMP
    x = g.row(v)
    y = g.qubit(v)

    for i in g.neighbors(v):
        newVert = g.add_vertex(2,y,x)
        g.add_edge((i,newVert), 1)

    g.remove_vertex(v)

    zx.draw(g, labels=True)
    #zx.simplify.full_reduce(g)
    #zx.draw(g, labels=True)

#----------

def cut(v): #TEMP
    #for i in g.neighbors(v): g.add_to_phase(25,1,{}) # Add pi to each neighbour
    g.remove_vertex(v)

#----------
```

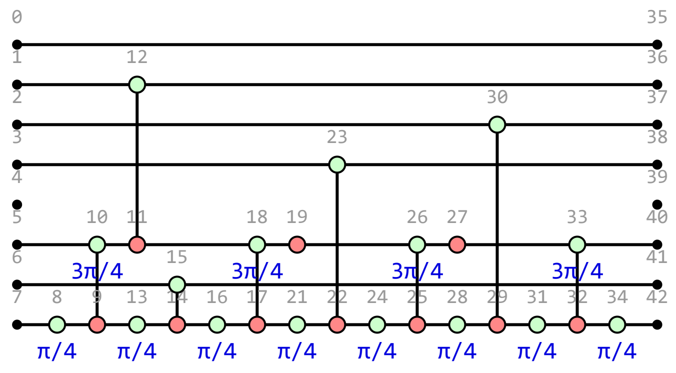
```python
def pcut(v,p): #TEMP
    for i in g.neighbors(v): g.add_to_phase(i,0,[p]) # Add param to each neighbour
    g.remove_vertex(v)

#----------

def apply_cut(g,v): # Return left and right branches of a vertex cut
    gLeft = g.copy()
    gRight = g.copy()
    gLeft.remove_vertex(v)
    gRight.remove_vertex(v)

    for i in g.neighbors(v):
        if g.type(i) == 2: # if red
            gRight.set_phase(i,gRight.phase(i)+1)

    return [gLeft,gRight]

#----------

def singleFusion(g): # TEMP - this is a very inefficient method
    fusionFound = False
    for i in g.vertices():
        for j in g.neighbors(i):
            if g.type(i) > 0 and g.type(i) == g.type(j) and g.edge_type((i,j)) == 1
                fusionFound = True
                break
        else: continue
        break
    if fusionFound:
        zx.basicrules.fuse(g,i,j)
    return [g,fusionFound]

def fullFusion(g):
    fullyFused = False
    while fullyFused == False:
        data = singleFusion(g)
        g = data[0]
        fullyFused = not data[1]
    return g

def idRemoval(g):
    deadSpiders = set() # Spiders marked for removal

    for i in g.vertices():
        if g.phase(i) == 0 and len(g.neighbors(i)) == 2: # Identity removal
            deadSpiders.add(i)

    for i in deadSpiders:
        neighs = list()
        for neigh in g.neighbors(i): # necessarily only has 2 neighbours here
            neighs.append(neigh)

        g.add_edge((neighs[0],neighs[1]),1)
        g.remove_vertex(i)

    return g
```

In [16]:
```python
zx.draw(g, labels=True, scale=20)
print("=> DECOMPOSES TO...")

cutChoice = 20 # the choice of vertex to cut
```

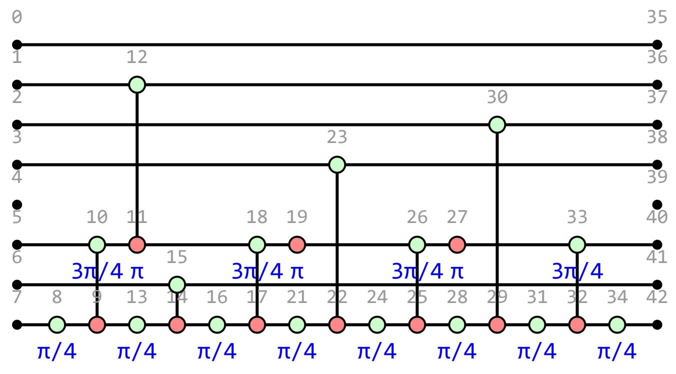
```
gList = apply_cut(g,cutChoice)
zx.draw(gList[0],labels=True,scale=20)
print("+")
zx.draw(gList[1],labels=True,scale=20)
```



```
=> DECOMPOSES TO...
```



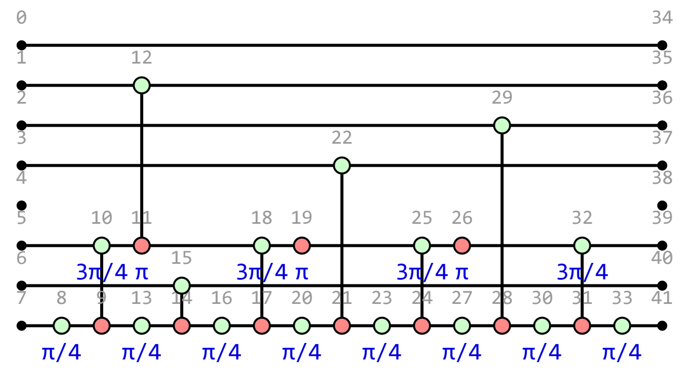
```
+
```



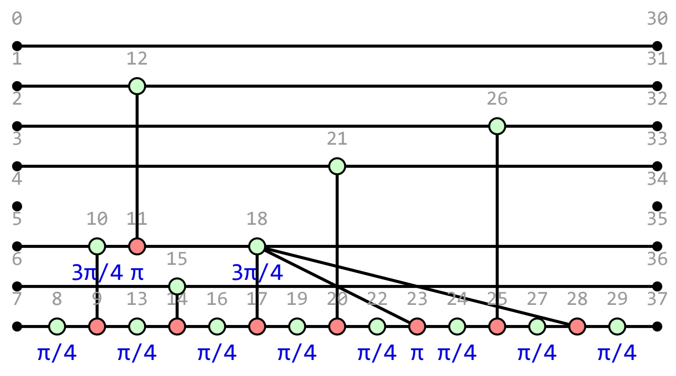
Having applied the cut (and thus decomposed our graph into 2 graphs), let's now partially simplify each while maintining structure. Specifically, we'll look closely at the second one to see how it simplifies...

In [17]:
```
#TEMP...
g = gList[1].copy()
#g.apply_state("0"*8)  #TEMP
#g.apply_effect("0"*8) #TEMP
zx.draw(g,labels=True,scale=20)

def pi_commute_Z(g,v): #zx.basicrules.pi_commute_Z(g,v)
    g.set_phase(v, -g.phase(v))
    ns = g.neighbors(v)
    for w in ns:
        e = g.edge(v, w)
        et = g.edge_type(e)
        if ((g.type(w) == 1 and et == 2) or
            (g.type(w) == 2 and et == 1)):
```

```
            g.add_to_phase(w, 1,[])
        else:
            g.remove_edge(e)
            c = g.add_vertex(2,
                    qubit=0.5*(g.qubit(v) + g.qubit(w)),
                    row=0.5*(g.row(v) + g.row(w)))
            g.add_edge(g.edge(v, c))
            g.add_edge(g.edge(c, w), edgetype=et)
    return g

def picom(g): # Push any 2-legged pi-phase Z-spiders through into a neighbouring (l
    applied = False
    for v in g.vertices():
        if (g.type(v)==2 and len(g.neighbors(v))==2 and g.phase(v)==1): # if red, 2
            #for i in g.neighbors(v): if (g.type(i)!=1): #Verify its neighbours are
            g = pi_commute_Z(g,min(g.neighbors(v)))
            g = idRemoval(g)
            g = fullFusion(g)
            applied = True
            break
    return [g,applied]

def partialSimp(g): # TODO - this function should be made a bit more robust
    g = fullFusion(g)
    g = idRemoval(g)
    g = fullFusion(g)

    picomApplies = True
    while (picomApplies == True):
        data = picom(g)
        g = data[0]
        picomApplies = data[1]

    g = idRemoval(g)
    return g.copy()

print("...partially simplifies to...")
g = partialSimp(g)
zx.draw(g,labels=True,scale=20)
```



```
...partially simplifies to...
```



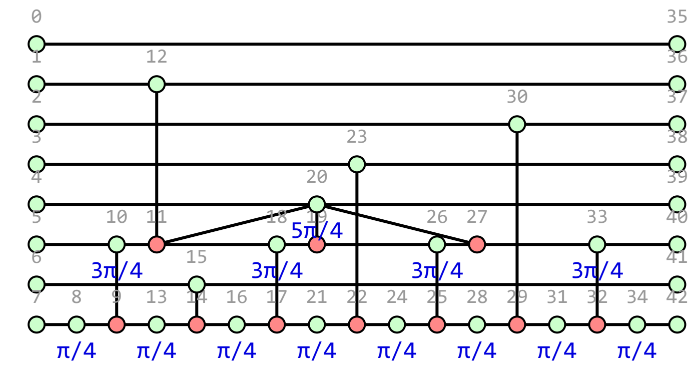
Now we have two graphs. We can update the weightings on each in a cleverer way, by merging the weights of fused vertices and only recalculating where a change has occurred etc., but since we're not too concerned with super-efficient implementation here, we can simply re-use the functions we have and recompute the weightings (still very quickly/efficiently) on each of these new graphs. And then we just repeat this process and boom! - we have ourselves a decomposition strategy!

Let's go back to our initial (example 2) circuit, make it scalar (i.e. plug all its inputs and outputs with something), and see how it fares...

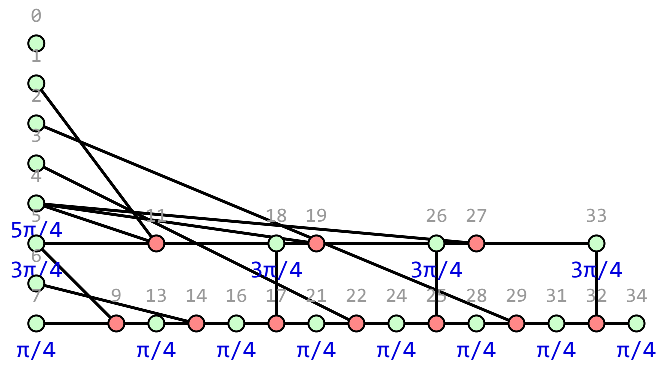
In [18]:
```
def cutGraphsOld(gList,debug): # Old version (does not ignore redundant, i.e. 0-sca
    hList = []
    for i in gList:
        g = i.copy()
        g = partialSimp(g)
        if (debug): zx.draw(g,labels=True,scale=20)
        cutChoice = compWeights(g,debug)[0]
        if (debug): print("     CUT",cutChoice)
        if (cutChoice > -1):
            split = apply_cut(g,cutChoice)
            hList.append(split[0])
            hList.append(split[1])
        else:
            hList.append(g)
    gList = hList.copy()
    return gList

def cutGraphs(gList,cliffCount,scalar,debug):
    hList = []
    for i in gList:
        g = i.copy()
        g = partialSimp(g)
        if (debug): zx.draw(g,labels=True,scale=20)
        cutChoice = compWeights(g,debug)[0]
        if (debug): print("     CUT",cutChoice)
        if (cutChoice > -1):
            split = apply_cut(g,cutChoice)

            gTemp0 = split[0].copy(); gTemp0 = partialSimp(gTemp0); zx.simplify.ful
            if (gTemp0.scalar.is_zero == False and zx.tcount(gTemp0) > 0): hList.ap
            else: cliffCount+=1; scalar+=gTemp0.scalar.to_number() # ADD TERM TO GL

            gTemp1 = split[1].copy(); gTemp1 = partialSimp(gTemp1); zx.simplify.ful
            if (gTemp1.scalar.is_zero == False and zx.tcount(gTemp1) > 0): hList.ap
            else: cliffCount+=1; scalar+=gTemp1.scalar.to_number() # ADD TERM TO GL
        else:
            hList.append(g) # If there are no more cuts found then don't cut (this
    gList = hList.copy()
    return [gList,cliffCount,scalar]
```

In [19]:
```
g = gOrig.copy()
print("Plug with some states and effects...")
g.apply_state("+"*8)
g.apply_effect("+"*8)
zx.draw(g,scale=20,labels=True)
partialSimp(g)
print("And partially simplify...")
zx.draw(g,scale=20,labels=True)
print("T-count = ", zx.tcount(g))
h = g.copy()
```

```
Plug with some states and effects...
```

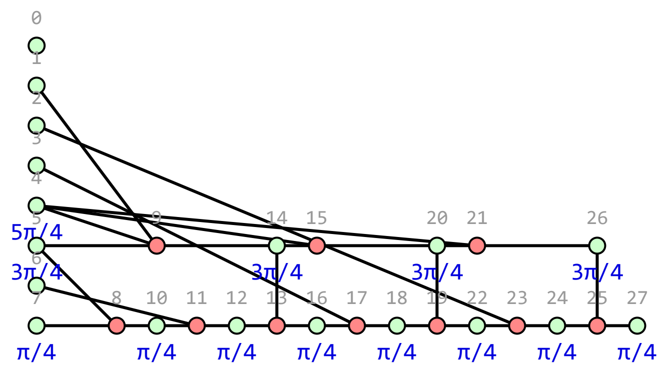


```
And partially simplify...
```



```
T-count =  13
```

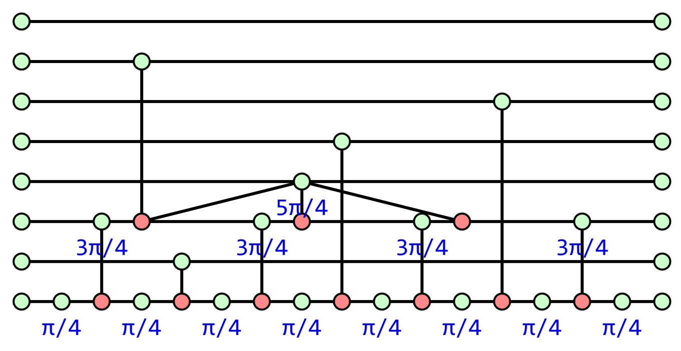
In [20]:
```
debug = False
softDebug = True

cliffCount = 0 # number of clifford terms (i.e. number of leaves of the cutting tre
scalar     = 0 # total scalar (= sum of the clifford terms)
treeDepth  = 0 # The depth of the cutting tree (i.e. max terms = 2^depth)

g = h.copy()
gTemp = g.copy(); zx.simplify.full_reduce(gTemp)

zx.draw(g, labels=True, scale=20)
print("Initial T-count = ", zx.tcount(g))
print("Reduced T-count = ", zx.tcount(gTemp))

#gList = apply_cut(11)
#zx.draw(gList[0],labels=True,scale=20)
#zx.draw(gList[1],labels=True,scale=20)

gList = [g]
if (debug): print("\nleaves \t buffer\t |  T-counts")
elif (softDebug): print("\nleaves \t buffer")

while(len(gList)>0):
    if (debug): print("##########")
    treeDepth += 1
    data       = cutGraphs(gList,cliffCount,scalar,debug)
    gList      = data[0]
    cliffCount = data[1]
    scalar     = data[2]

    if (debug and len(gList) < 10):
        strLine = str(cliffCount) + " \t " + str(len(gList)) + " \t |  "
        for i in range(len(gList)):
            gTemp = gList[i].copy()
            zx.simplify.full_reduce(gTemp)
            strLine += str(zx.tcount(gTemp)) + "  "
            #zx.draw(gTemp,scale=20,labels=True) #TEMP
        print(strLine)
    elif (softDebug): print(cliffCount,"\t",len(gList))

print("\nNUMBER OF CLIFFORD TERMS:",cliffCount)
print("( tree depth:",treeDepth,")")
```



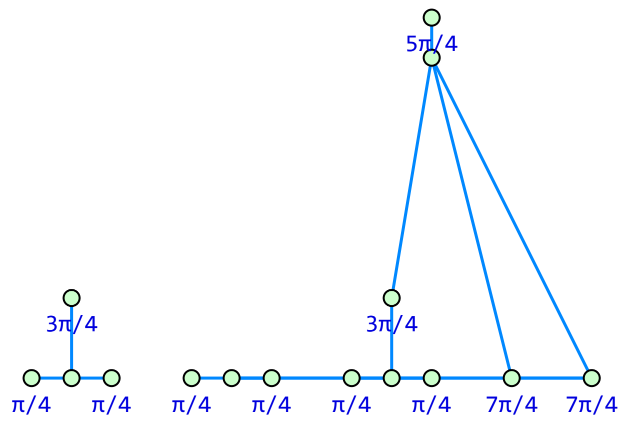
```
Initial T-count =  13
Reduced T-count =  11

leaves   buffer
0        2
0        4
8        0

NUMBER OF CLIFFORD TERMS: 8
( tree depth: 3 )
```

In this simple case, our initial graph was decomposed into 2 graphs, and each of those was then decomposed into another 2, and each of THOSE into another 2. So, we decomposed our 13 T-count graph into 8 Clifford graph terms, hence alpha ~= 0.23? Well, this is a bit misleading, since our original graph could have been reduced to T-count 11 with Clifford simplification...

In [21]:
```
g = gOrig.copy()
g.apply_state("+"*8)
g.apply_effect("+"*8)
zx.draw(g,scale=20)
zx.simplify.full_reduce(g)
print("...reduces via Clifford simp to...")
zx.draw(g,scale=20)
h = g.copy()
```



```
...reduces via Clifford simp to...
```



So really, applied to this circuit we observe an effective alpha of log_2(8)/11 = 0.27. Still a great result! And much better than the estimate of alpha=0.47 that the BSS decomposition achieves. In fact, let's compare to how well the BSS would have done on this particular circuit...

In [22]:
```
gDecomp = zx.simulate.find_stabilizer_decomp(g)
print("No. of terms via BSS:",len(gDecomp))
```

```
No. of terms via BSS: 11
```

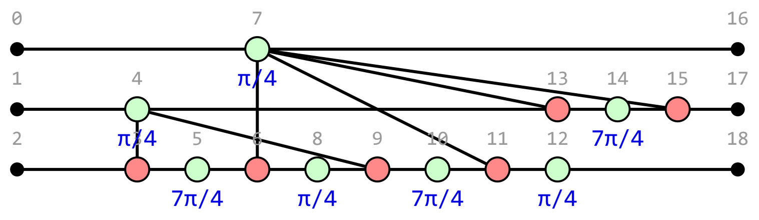
The BSS method (with inter-step Clifford simplification) would have reduced our circuit to 11 stabiliser terms (alpha=0.31). Our method achieved 8 (alpha=0.23)! And this was just a trivial simple example - later we'll see how well it works on much larger T-count circuits. But first, let's check how often our procedure finds the MOST optimal set of vertex cuts (rather than simply AN optimal set)...

Benchmarking 1

Let's generate a very simple and small example circuit...

In [23]:
```
strCirc = """
qreg q[3];
ccx q[0], q[1], q[2];
"""

c = zx.qasm(strCirc)
g = c.to_graph()
#zx.draw(g, labels=True)

partialSimp(g)
gOrig = g.copy()
```

```
zx.draw(gOrig, labels=True,scale=30)
print("T-count = ", zx.tcount(gOrig))
```



```
T-count =  7
```

This is small enough such that we can try out every possible combination of vertex cuts (among its 7 Z-spiders). In each case, we try to simplify as much as possible after the cuts and if any T-spiders still remain then we assume that we can remove them with alpha=0.47 (i.e. falling back on the BSS decomposition)...

In [24]:
```
debug = False
softDebug = True
testLimit = 500 # set to -1 for no limit

spiders = []
aBest = 999
termsBest = 999999999999

tOrig = zx.tcount(gOrig)
inps = gOrig.inputs()
outs = gOrig.outputs()

for v in gOrig.vertices():
    if (gOrig.type(v) == 1 and not(v in inps) and not(v in outs)): # Only consider
        spiders.append(v)

for i in range(2**len(spiders)):
    #i = 17954 #TEMP
    g = gOrig.copy()

    b = str(bin(i)[2:]).rjust(len(spiders), '0')
    n_cuts = b.count("1") # number of cuts (i.e. 2^n = no. of terms)

    for j in range(len(spiders)):
        if(b[j]=="1"): g.remove_vertex(spiders[j])

    for j in inps:
        if(len(g.neighbors(j))<1): g.remove_vertex(j)
    for j in outs:
        if(len(g.neighbors(j))<1): g.remove_vertex(j)

    zx.simplify.full_reduce(g)

    t = zx.tcount(g)
    tDiff = tOrig - t                # tDiff := no. of T-gates that were reduced
    n_terms = 2**n_cuts
    if (t>0): n_terms *= 7**(t/6)  # If there are remaining T-gates, assume these a
    a = math.log(n_terms,2)/tOrig   # a := the decomposition efficiency, i.e. alpha

    if (debug): zx.draw(g, scale=20, labels=True)
    if (softDebug): print(i, "("+b+")\t|  cuts:", n_cuts, "\t|  ", "Terms:", math.c
    if (a < aBest):
        aBest = a
        termsBest = n_terms
    if (testLimit > -1 and i > testLimit): break #TEMP
```

```
print("\n\nBEST ALPHA:",aBest) # The best result for alpha
print("NUM. TERMS:",termsBest) # The best result for no. of terms (i.e. 2^(aBest)t)
```

```
0 (0000000)    |  cuts: 0    |   Terms: 10   |   ALPHA:  0.4678924870096007
1 (0000001)    |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
2 (0000010)    |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
3 (0000011)    |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
4 (0000100)    |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
5 (0000101)    |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
6 (0000110)    |  cuts: 2    |   Terms: 6    |   ALPHA:  0.3525560695728001
7 (0000111)    |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
8 (0001000)    |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
9 (0001001)    |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
10 (0001010)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
11 (0001011)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
12 (0001100)   |  cuts: 2    |   Terms: 11   |   ALPHA:  0.4862396372898289
13 (0001101)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
14 (0001110)   |  cuts: 3    |   Terms: 16   |   ALPHA:  0.5622549962884573
15 (0001111)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
16 (0010000)   |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
17 (0010001)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
18 (0010010)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
19 (0010011)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
20 (0010100)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
21 (0010101)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
22 (0010110)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
23 (0010111)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
24 (0011000)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
25 (0011001)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
26 (0011010)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
27 (0011011)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
28 (0011100)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
29 (0011101)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
30 (0011110)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
31 (0011111)   |  cuts: 5    |   Terms: 45   |   ALPHA:  0.7811274981442287
32 (0100000)   |  cuts: 1    |   Terms: 2    |   ALPHA:  0.14285714285714285
33 (0100001)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
34 (0100010)   |  cuts: 2    |   Terms: 11   |   ALPHA:  0.4862396372898289
35 (0100011)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
36 (0100100)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
37 (0100101)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
38 (0100110)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
39 (0100111)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
40 (0101000)   |  cuts: 2    |   Terms: 6    |   ALPHA:  0.3525560695728001
41 (0101001)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
42 (0101010)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
43 (0101011)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
44 (0101100)   |  cuts: 3    |   Terms: 16   |   ALPHA:  0.5622549962884573
45 (0101101)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
46 (0101110)   |  cuts: 4    |   Terms: 43   |   ALPHA:  0.7719539230041146
47 (0101111)   |  cuts: 5    |   Terms: 32   |   ALPHA:  0.7142857142857143
48 (0110000)   |  cuts: 2    |   Terms: 4    |   ALPHA:  0.2857142857142857
49 (0110001)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
50 (0110010)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
51 (0110011)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
52 (0110100)   |  cuts: 3    |   Terms: 12   |   ALPHA:  0.49541321242994296
53 (0110101)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
54 (0110110)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
55 (0110111)   |  cuts: 5    |   Terms: 32   |   ALPHA:  0.7142857142857143
56 (0111000)   |  cuts: 3    |   Terms: 8    |   ALPHA:  0.42857142857142855
57 (0111001)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
58 (0111010)   |  cuts: 4    |   Terms: 23   |   ALPHA:  0.6382703552870858
59 (0111011)   |  cuts: 5    |   Terms: 32   |   ALPHA:  0.7142857142857143
60 (0111100)   |  cuts: 4    |   Terms: 16   |   ALPHA:  0.5714285714285714
61 (0111101)   |  cuts: 5    |   Terms: 45   |   ALPHA:  0.7811274981442287
62 (0111110)   |  cuts: 5    |   Terms: 32   |   ALPHA:  0.7142857142857143
63 (0111111)   |  cuts: 6    |   Terms: 89   |   ALPHA:  0.9239846410013717
```

```
64 (1000000)    |  cuts: 1     |  Terms: 2    |  ALPHA:  0.14285714285714285
65 (1000001)    |  cuts: 2     |  Terms: 6    |  ALPHA:  0.3525560695728001
66 (1000010)    |  cuts: 2     |  Terms: 4    |  ALPHA:  0.2857142857142857
67 (1000011)    |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
68 (1000100)    |  cuts: 2     |  Terms: 4    |  ALPHA:  0.2857142857142857
69 (1000101)    |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
70 (1000110)    |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
71 (1000111)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
72 (1001000)    |  cuts: 2     |  Terms: 4    |  ALPHA:  0.2857142857142857
73 (1001001)    |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
74 (1001010)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
75 (1001011)    |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
76 (1001100)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
77 (1001101)    |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
78 (1001110)    |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
79 (1001111)    |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
80 (1010000)    |  cuts: 2     |  Terms: 4    |  ALPHA:  0.2857142857142857
81 (1010001)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
82 (1010010)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
83 (1010011)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
84 (1010100)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
85 (1010101)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
86 (1010110)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
87 (1010111)    |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
88 (1011000)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
89 (1011001)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
90 (1011010)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
91 (1011011)    |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
92 (1011100)    |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
93 (1011101)    |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
94 (1011110)    |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
95 (1011111)    |  cuts: 6     |  Terms: 64   |  ALPHA:  0.8571428571428571
96 (1100000)    |  cuts: 2     |  Terms: 4    |  ALPHA:  0.2857142857142857
97 (1100001)    |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
98 (1100010)    |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
99 (1100011)    |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
100 (1100100)   |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
101 (1100101)   |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
102 (1100110)   |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
103 (1100111)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
104 (1101000)   |  cuts: 3     |  Terms: 12   |  ALPHA:  0.49541321242994296
105 (1101001)   |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
106 (1101010)   |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
107 (1101011)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
108 (1101100)   |  cuts: 4     |  Terms: 23   |  ALPHA:  0.6382703552870858
109 (1101101)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
110 (1101110)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
111 (1101111)   |  cuts: 6     |  Terms: 89   |  ALPHA:  0.9239846410013717
112 (1110000)   |  cuts: 3     |  Terms: 8    |  ALPHA:  0.42857142857142855
113 (1110001)   |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
114 (1110010)   |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
115 (1110011)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
116 (1110100)   |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
117 (1110101)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
118 (1110110)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
119 (1110111)   |  cuts: 6     |  Terms: 64   |  ALPHA:  0.8571428571428571
120 (1111000)   |  cuts: 4     |  Terms: 16   |  ALPHA:  0.5714285714285714
121 (1111001)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
122 (1111010)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
123 (1111011)   |  cuts: 6     |  Terms: 64   |  ALPHA:  0.8571428571428571
124 (1111100)   |  cuts: 5     |  Terms: 32   |  ALPHA:  0.7142857142857143
125 (1111101)   |  cuts: 6     |  Terms: 64   |  ALPHA:  0.8571428571428571
126 (1111110)   |  cuts: 6     |  Terms: 64   |  ALPHA:  0.8571428571428571
127 (1111111)   |  cuts: 7     |  Terms: 128  |  ALPHA:  1.0
```

```
BEST ALPHA: 0.14285714285714285
NUM. TERMS: 2
```

In [25]:
```
spiders # The vertices liable for cutting (i.e. 0000011 above means we cut vertices
```

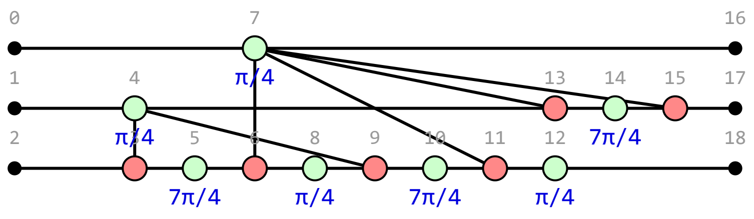
Out[25]:
```
[4, 5, 7, 8, 10, 12, 14]
```

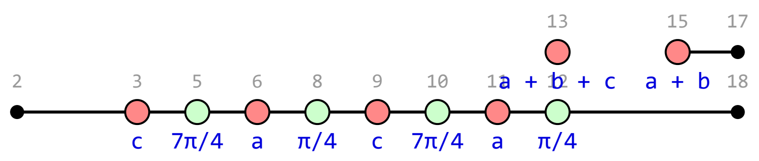
Having tried all 128 possible combinations of vertex cuts, we see that the best possible solution(s) removes all the T-gates at the cost of just 2 stabiliser terms (hence alpha = 0.14). For instance, option '1 (0000001)' achieves this result (which cuts only vertex 14)

[OPTIONAL] If we install the parameter-supported version of PyZX from https://github.com/mjsutcliffe99/ParamZX, then we can improve the above verifications by including "cut order correction" as outlined in the paper, to correct for any suboptimality in the cut ordering...

In [26]:
```
def orderCorrect(gg):
    pRedund = [] # track redundant params
    for pvars in gg.scalar.phasenodevars:
        for var in pvars:
            if not(var in pRedund):
                pRedund.append(var)
                break
    return pRedund
```

In [27]:
```
# Demonstrate order correction...

g = gOrig.copy()
zx.draw(g, labels=True,scale=30)

chars = 'abcdefghijklmnopqrstuvwxyz'
pnum = 0

pcut(7,chars[pnum]);  pnum+=1
pcut(14,chars[pnum]); pnum+=1
pcut(4,chars[pnum]);  pnum+=1

for j in inps:
    if(len(g.neighbors(j))<1): g.remove_vertex(j)
for j in outs:
    if(len(g.neighbors(j))<1): g.remove_vertex(j)

zx.draw(g, labels=True,scale=30)

zx.simplify.full_reduce(g)
orderCorrect(g)
```

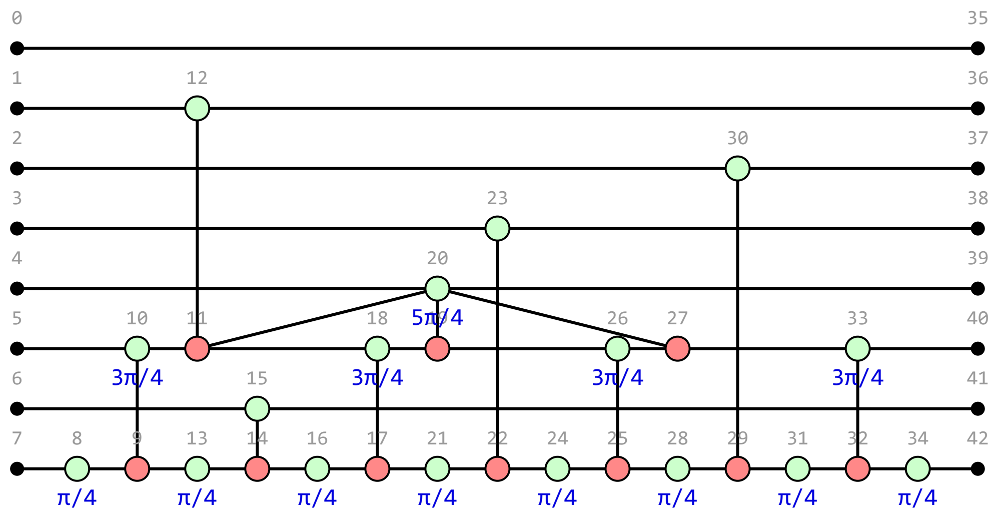


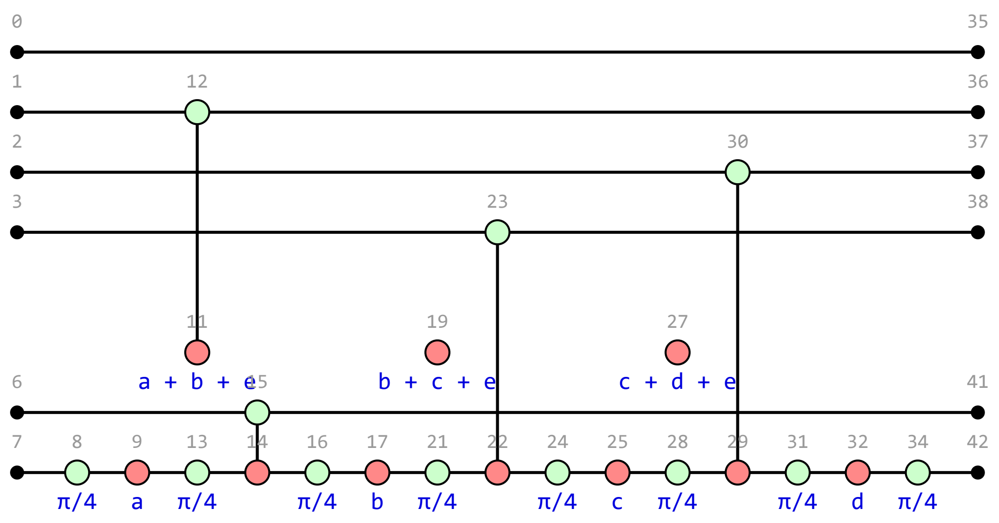


Out[27]:
```
['c']
```

Above we've considered the option of cutting all 7 Z-spiders in our example graph (which naively results in 2^7 = 128 stabiliser terms). However, with some parameter analysis, we can infer that we have one redundant parameter ('b') and so (ignoring the 0-terms) this would actually result in 2^6 = 64 stabiliser terms. Let's look at another example...

In [28]:
```
# Demonstrate order correction...

g = gEx2.copy()
zx.draw(g, labels=True, scale=30)

spiders = []
t = zx.tcount(g)
inps = g.inputs()
outs = g.outputs()

chars = 'abcdefghijklmnopqrstuvwxyz'
pnum = 0

pcut(10,chars[pnum]); pnum+=1
pcut(18,chars[pnum]); pnum+=1
pcut(26,chars[pnum]); pnum+=1
pcut(33,chars[pnum]); pnum+=1
#pcut(12,chars[pnum]); pnum+=1
pcut(20,chars[pnum]); pnum+=1

for j in inps:
    if(len(g.neighbors(j))<1): g.remove_vertex(j)
for j in outs:
    if(len(g.neighbors(j))<1): g.remove_vertex(j)

zx.draw(g, labels=True, scale=30)

zx.simplify.full_reduce(g)
orderCorrect(g)
```





```
Out[28]: ['c', 'd']
```

In this graph, if we were to cut vertices 10, 18, 26, 33, and 20, we'd naively conclude this results in 2^5 = 32 terms. However, with some cut order correction we see that we actually have 2 redundant parameters here ('b' and 'c') and hence this would actually reduce to 2^3 = 7 terms (as if we had cut vertex 20 first and THEN fused the others and cut them as one).

So, let's redo our verification experiment from above, but with this cut order correction...

```
In [29]:
# PARAM-SUPPORTED VERSION...

debug = False
softDebug = True
doOrderCorrect = True # Use parameter analysis to correct for any suboptimal cut or
testLimit = 500 # set to -1 for no limit

spiders = []
aBest = 999
termsBest = 999999999999
chars = 'abcdefghijklmnopqrstuvwxyz'

tOrig = zx.tcount(gOrig)
inps = gOrig.inputs()
outs = gOrig.outputs()

for v in gOrig.vertices():
    if (gOrig.type(v) == 1 and not(v in inps) and not(v in outs)): # Only consider
        spiders.append(v)

for i in range(2**len(spiders)):
    #i = 17954 #TEMP
    g = gOrig.copy()
    pnum = 0

    b = str(bin(i)[2:]).rjust(len(spiders), '0')
    n_cuts = b.count("1") # number of cuts (i.e. 2^n = no. of terms)

    for j in range(len(spiders)):
        if(b[j]=="1"):
            pcut(spiders[j],chars[pnum]) #g.remove_vertex(spiders[j])
            pnum += 1

    for j in inps:
        if(len(g.neighbors(j))<1): g.remove_vertex(j)
    for j in outs:
        if(len(g.neighbors(j))<1): g.remove_vertex(j)

    zx.simplify.full_reduce(g)
    for v in g.vertices(): g.set_phase(v,g.phase(v)) # Reduce all params -> 0

    if (doOrderCorrect):
        n_cuts -= len(orderCorrect(g)) # Correct for suboptimal cut order
        #print("cut correct",orderCorrect(g)) #TEMP
    zx.simplify.full_reduce(g)

    t = zx.tcount(g)
    tDiff = tOrig - t               # tDiff := no. of T-gates that were reduced
    n_terms = 2**n_cuts
    if (t>0): n_terms *= 7**(t/6)  # If there are remaining T-gates, assume these c
    a = math.log(n_terms,2)/tOrig   # a := the decomposition efficiency, i.e. alpha

    if (debug): zx.draw(g, scale=20, labels=True)
    if (softDebug): print(i, "("+b+")\t|  cuts:", n_cuts, "\t|  ", "Terms:", math.c
    if (a < aBest):
        aBest = a
        termsBest = n_terms
    if (testLimit > -1 and i > testLimit): break #TEMP

print("\n\nBEST ALPHA:",aBest) # The best result for alpha
print("NUM. TERMS:",termsBest) # The best result for no. of terms (i.e. 2^(aBest)t)
```

```
0 (0000000)     |  cuts: 0     |  Terms: 10   |   ALPHA:  0.4678924870096007
1 (0000001)     |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
2 (0000010)     |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
3 (0000011)     |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
4 (0000100)     |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
5 (0000101)     |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
6 (0000110)     |  cuts: 2     |  Terms: 6    |   ALPHA:  0.3525560695728001
7 (0000111)     |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
8 (0001000)     |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
9 (0001001)     |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
10 (0001010)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
11 (0001011)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
12 (0001100)    |  cuts: 2     |  Terms: 11   |   ALPHA:  0.4862396372898289
13 (0001101)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
14 (0001110)    |  cuts: 3     |  Terms: 16   |   ALPHA:  0.5622549962884573
15 (0001111)    |  cuts: 4     |  Terms: 16   |   ALPHA:  0.5714285714285714
16 (0010000)    |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
17 (0010001)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
18 (0010010)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
19 (0010011)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
20 (0010100)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
21 (0010101)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
22 (0010110)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
23 (0010111)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
24 (0011000)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
25 (0011001)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
26 (0011010)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
27 (0011011)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
28 (0011100)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
29 (0011101)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
30 (0011110)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
31 (0011111)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
32 (0100000)    |  cuts: 1     |  Terms: 2    |   ALPHA:  0.14285714285714285
33 (0100001)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
34 (0100010)    |  cuts: 2     |  Terms: 6    |   ALPHA:  0.3525560695728001
35 (0100011)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
36 (0100100)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
37 (0100101)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
38 (0100110)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
39 (0100111)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
40 (0101000)    |  cuts: 2     |  Terms: 6    |   ALPHA:  0.3525560695728001
41 (0101001)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
42 (0101010)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
43 (0101011)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
44 (0101100)    |  cuts: 3     |  Terms: 16   |   ALPHA:  0.5622549962884573
45 (0101101)    |  cuts: 4     |  Terms: 16   |   ALPHA:  0.5714285714285714
46 (0101110)    |  cuts: 4     |  Terms: 43   |   ALPHA:  0.7719539230041146
47 (0101111)    |  cuts: 5     |  Terms: 45   |   ALPHA:  0.7811274981442287
48 (0110000)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
49 (0110001)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
50 (0110010)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
51 (0110011)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
52 (0110100)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
53 (0110101)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
54 (0110110)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
55 (0110111)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
56 (0111000)    |  cuts: 2     |  Terms: 4    |   ALPHA:  0.2857142857142857
57 (0111001)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
58 (0111010)    |  cuts: 3     |  Terms: 12   |   ALPHA:  0.49541321242994296
59 (0111011)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
60 (0111100)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
61 (0111101)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
62 (0111110)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
63 (0111111)    |  cuts: 3     |  Terms: 8    |   ALPHA:  0.42857142857142855
```

```
64 (1000000)     |  cuts: 1      |  Terms: 2    |  ALPHA:  0.14285714285714285
65 (1000001)     |  cuts: 2      |  Terms: 6    |  ALPHA:  0.3525560695728001
66 (1000010)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
67 (1000011)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
68 (1000100)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
69 (1000101)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
70 (1000110)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
71 (1000111)     |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
72 (1001000)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
73 (1001001)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
74 (1001010)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
75 (1001011)     |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
76 (1001100)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
77 (1001101)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
78 (1001110)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
79 (1001111)     |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
80 (1010000)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
81 (1010001)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
82 (1010010)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
83 (1010011)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
84 (1010100)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
85 (1010101)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
86 (1010110)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
87 (1010111)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
88 (1011000)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
89 (1011001)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
90 (1011010)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
91 (1011011)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
92 (1011100)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
93 (1011101)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
94 (1011110)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
95 (1011111)     |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
96 (1100000)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
97 (1100001)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
98 (1100010)     |  cuts: 2      |  Terms: 4    |  ALPHA:  0.2857142857142857
99 (1100011)     |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
100 (1100100)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
101 (1100101)    |  cuts: 4      |  Terms: 23   |  ALPHA:  0.6382703552870858
102 (1100110)    |  cuts: 4      |  Terms: 23   |  ALPHA:  0.6382703552870858
103 (1100111)    |  cuts: 5      |  Terms: 32   |  ALPHA:  0.7142857142857143
104 (1101000)    |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
105 (1101001)    |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
106 (1101010)    |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
107 (1101011)    |  cuts: 5      |  Terms: 45   |  ALPHA:  0.7811274981442287
108 (1101100)    |  cuts: 3      |  Terms: 12   |  ALPHA:  0.49541321242994296
109 (1101101)    |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
110 (1101110)    |  cuts: 4      |  Terms: 16   |  ALPHA:  0.5714285714285714
111 (1101111)    |  cuts: 6      |  Terms: 89   |  ALPHA:  0.9239846410013717
112 (1110000)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
113 (1110001)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
114 (1110010)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
115 (1110011)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
116 (1110100)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
117 (1110101)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
118 (1110110)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
119 (1110111)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
120 (1111000)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
121 (1111001)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
122 (1111010)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
123 (1111011)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
124 (1111100)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
125 (1111101)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
126 (1111110)    |  cuts: 3      |  Terms: 8    |  ALPHA:  0.42857142857142855
127 (1111111)    |  cuts: 7      |  Terms: 128  |  ALPHA:  1.0
```

```
BEST ALPHA: 0.14285714285714285
NUM. TERMS: 2
```

The final "most optimal" result is the same in this case, but looking at the results for specific combinations we see that this actually results in much fewer terms in most cases (as we're essentially ignoring redundant branches when cutting).

We can replace the example circuit above with any other, or generate a bunch of random examples to benchmark our method and see how often our approach finds the most optimal result

Benchmarking 2

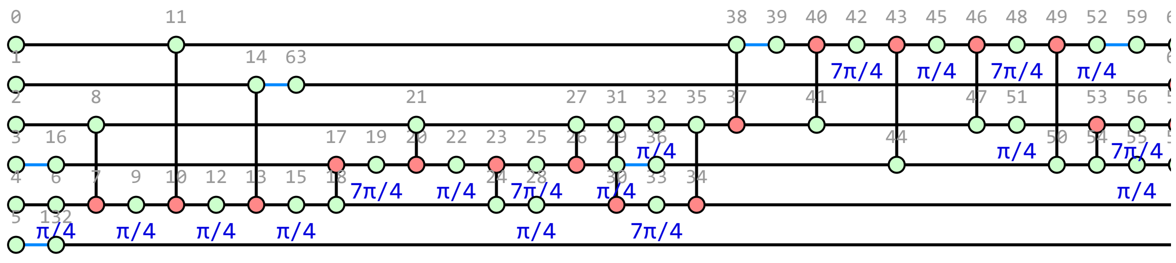
Now let's consider some larger random circuits (which are too big to verify with brute force as above). Instead, we want to compare our results to the method of Kissinger and van de Wetering (which relies primarily on the BSS decomposition). So, first let's generate a large random circuit that has some local structural elements...

[Note: set useExample = False here if you want to generate new random examples, rather than using the one generated in advance. But note that the text ahead will refer to the results from this particular example circuit]

In [30]:
```
useExample = True # FEEL FREE TO TOGGLE THIS
```

In [31]:
```
# GENERATE RANDOM (NON-TRIVIALLY) STRUCTURED CIRCUITS...

includeToffHads = True # Include the Hadamards in the Toffoli?

def rq(avoidA=-1, avoidB=-1): # select a random qubit (avoiding, if desired, specif
    if (avoidA < 0 and avoidB < 0): return random.randrange(1,NQ+1)-1
    rNum = avoidA
    while (rNum in [avoidA, avoidB]): rNum = random.randrange(1,NQ+1)-1
    return rNum

def addGate(gate, b=-1, a=-1, c=-1): # Add a gate on qubits a,b,c (or randomise eit
    if (b<0): b=rq()
    if (a<0): a=rq(b)
    if (c<0): c=rq(a,b)
    strGate = "\n"
    match gate:
        case "t":    strGate += "t q[" + str(b) + "];"
        case "cnot": strGate += "cx q[" + str(a) + "], q[" + str(b) + "];"
        case "toff":
            if (includeToffHads): strGate += "h q[" + str(c) + "];\n"
            strGate += "ccz q[" + str(a) + "], q[" + str(b) + "], q[" + str(c) + "]
            if (includeToffHads): strGate += "\nh q[" + str(c) + "];"
        case "rz":
            ph = random.randrange(1,8) # random Clifford+T phase (no point in inclu
            if (b==1): ph = random.randrange(1,4)*2 # if Clifford
            ph = ph/4
            strGate += "rz(" + str(ph) + "*pi) q[" + str(a) + "];"
        case "rx":
            ph = random.randrange(1,8) # random Clifford+T phase (no point in inclu
            if (b==1): ph = random.randrange(1,4)*2 # if Clifford
            ph = ph/4
            strGate += "rx(" + str(ph) + "*pi) q[" + str(a) + "];"
    qasmLines.append(strGate)
    return [a,b,c]

def tSandwich(t=2): # Generate a T-CNOT-T-CNOT-T-... sandwich of length t (minimum
    q = addGate("t")[1]
    for i in range(t-1):
        addGate("cnot",q)
        addGate("t",q)

def toff(n=1,cnots=False): # Generate a chain of randomly placed Toffoli gates ('cr
    for i in range(n):
        q = addGate("toff")[0]
        if(cnots): addGate("cnot", q)

def rz(cliff=True): addGate("rz",int(cliff))
def rx(cliff=True): addGate("rx",int(cliff))
```

In [32]:
```
# Example (feel free to change this up)...

NQ = 6 # No. of qubits

strCirc = "\nqreg q[" + str(NQ) + "];"
qasmLines = list()

tSandwich(4)
toff(2,True)
rz(True)
toff(1,True)
rz(False)
toff(1,True)
rz(False)
rz(False)

#----------

for line in qasmLines: strCirc += line
if (useExample): strCirc = "\nqreg q[6];\nt q[4];\ncx q[2], q[4];\nt q[4];\ncx q[0]
c = zx.qasm(strCirc)
g = c.to_graph()

g.apply_state("+"*NQ)  #TEMP
g.apply_effect("+"*NQ) #TEMP

#g.normalize()
zx.draw(g, labels=True, scale=20)
print("T-count = ", zx.tcount(g))
```

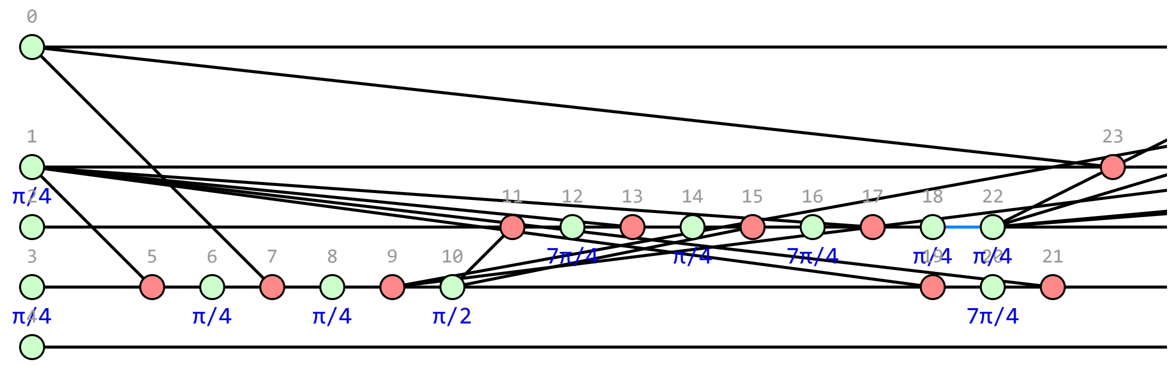


```
T-count =  49
```

Here we've generated a random circuit that has a T-count of 49 (unless you generated a new random circuit). With some partial simplification (but keeping the graph structure), this reduces a little to 47 in this case...

In [33]:
```
partialSimp(g)
h = g.copy(); g = h.copy() # This just re-numbers the vertex indexes to remove any
```

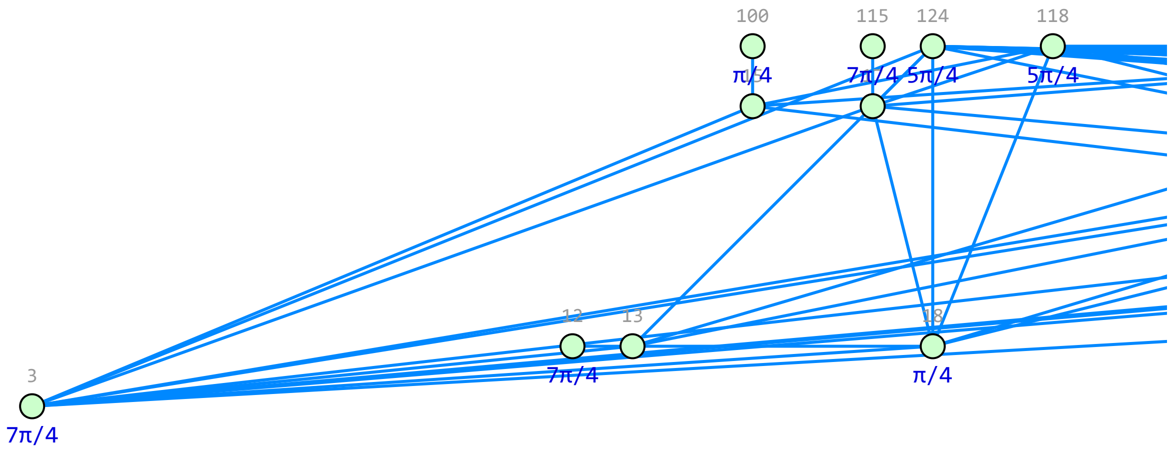
```
zx.draw(g, labels=True, scale=30)
print("T-count = ", zx.tcount(g))
```



```
T-count =  47
```

Even though we'll be starting from this above circuit (without doing a full Clifford simplification first - as this compromises the structure), we need to know what the TRUE initial T-count is (after applying full Clifford simp.)...

In [34]:
```
zx.simplify.full_reduce(g)
zx.draw(g, labels=True, scale=30)
print("T-count = ", zx.tcount(g))
g = h.copy()
```



```
T-count =  22
```

22. So we'll take note of this as our initial T-count (the best we can get before needing to decompose something)

Now, let's go back to our graph-like version and apply the procedure to it...

In [35]:
```
def killnullverts(gg): # This removes any free (legless) nodes of null-type (i.e. t
    nullverts = list()
    for v in gg.vertices():
        if (g.type(v)==0 and len(g.neighbors(v))==0): nullverts.append(v)
    for v in nullverts:
        gg.remove_vertex(v)
```

In [36]:
```
vData = compWeights(g)

vBest        = vData[0]
tier         = vData[1]
vweights_max = vData[2]
vweights     = vData[3]
vtiers       = vData[4]
```

```
--== TIER 1 ==--
vertex   weight  children
0        4.0     {8, 1, 6, 25}
1        8.0     {3, 6, 41, 12, 14, 16, 18, 20}
10       2.0     {16, 14}
22       6.0     {32, 34, 25, 26, 28, 30}
25       2.0     {28, 30}
36       2.0     {42, 44}
38       5.0     {52, 53, 22, 57, 59, 61}
41       6.0     {36, 39, 42, 44, 46, 48}
50       2.0     {41, 52}
52       3.0     {57, 53, 22, 55}
59       2.0     {80, 68}
64       2.0     {69, 71}
68       6.0     {64, 66, 69, 71, 73, 75}
77       6.0     {80, 81, 83, 85, 87, 89}
80       2.0     {83, 85}
93       2.0     {97, 95}

--== TIER 2 ==--
0        2.0     {1, 25}
50       2.0     {41, 52}
59       2.0     {80, 68}
```

In [37]:
```
dispWeights(tier,vweights_max,vweights,vtiers)
print("\nBEST CUT:",vBest)
```

```
vertex   weight          tier
0        4.0 (+0)        2
1        8.0 (+1)        1
10       2.0 (+0)        1
22       6.0 (+1)        1
25       2.0 (+1)        1
36       2.0 (+1)        1
38       5.0 (+1)        1
41       6.0 (+1)        1
50       2.0 (+0)        2
52       3.0 (+1)        1
59       2.0 (+1)        2
64       2.0 (+1)        1
68       6.0 (+1)        1
77       6.0 (+1)        1
80       2.0 (+1)        1
93       2.0 (+0)        1

BEST CUT: 0
```

In [38]:
```
def getTierCutOrder(tier,vweights_max,vweights,vtiers): # Returns the cut order by
    VS = []
    WS = []
    for i in range(len(vweights)):
        if (vtiers[i] == tier and vweights_max[i]>0): #if (len(vDoorsteps[i]) > 0):
            w = vweights_max[i]
            b = 0
            if (g.phase(i) in [0.25,0.75,1.25,1.75]): b+=1 # Add 1 to weight if the
            VS.append(i)
            WS.append(w+b)
    return [VS,WS]

def getCutOrder(vweights_max,vweights,vtiers): # Returns ALL weighted vertices, ord
    cutOrders = []
    for t in range(max(vtiers),0,-1):
        data = getTierCutOrder(t,vweights_max,vweights,vtiers)
        VS = data[0]
        WS = data[1]
        cutOrder = [x for _,x in sorted(zip(WS,VS))]
        cutOrder.reverse()
        cutOrders += cutOrder
    return cutOrders
```

In [39]:
```
cutOrder = getCutOrder(vweights_max,vweights,vtiers)
print(cutOrder)
```

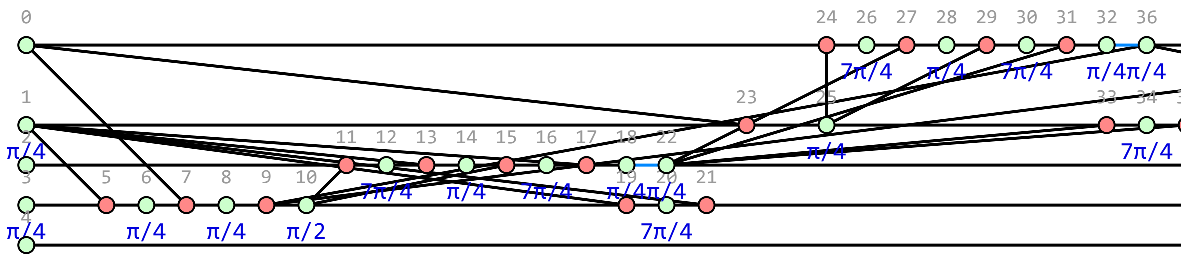
```
[0, 59, 50, 1, 77, 68, 41, 22, 38, 52, 80, 64, 36, 25, 93, 10]
```

In the paper, we present a method of efficiently/quickly estimating the number of terms produced by this method, utilising parameterisation. Here, we offer a (slightly less automatic) alternative estimation approach which does not require the modified PyZX package (but rather can run on the standard version of PyZX)...

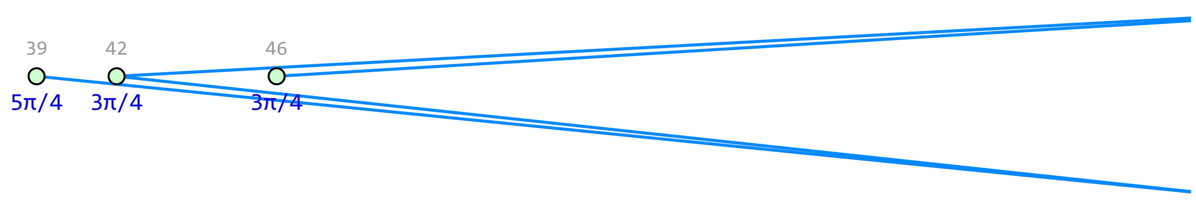
The above (getCutOrder) function gives us a list of the weighted vertices in order of cutting priority. For a quick ESTIMATION of the number of stabiliser terms we're likely to get, we'll assume this list does not change after the cuts, or vary between branches (as in reality it would). So, let's decide how many we want to cut (starting from the highest priority down)...

In [40]:
```
cutcount = 4 # FEEL FREE TO CHANGE THIS
```

In [41]:
```
# CUT ANY NUMBER OF VERTICES, BY ORDER OF CUTTING PREFERENCE (AS DETERMINED BY THE

g = h.copy()
zx.draw(g, labels=True, scale=20)
print("T-count = ", zx.tcount(g), "\n\n")

cutOrder = getCutOrder(vweights_max,vweights,vtiers)

for i in range(cutcount):
    g.remove_vertex(cutOrder[i]); killnullverts(g);

#partialSimp()
zx.simplify.full_reduce(g)

print("cutcount:",cutcount,"      |   ",2**(cutcount))
zx.draw(g, labels=True, scale=20)
print("T-count = ", zx.tcount(g))
```



```
T-count =  47


cutcount: 4      |   16
```



We can use a bit of trial and error to find that cutcount = 4 seems to be the best choice - i.e. the fewest number of cuts that reduces our circuit to a trivially small T-count (at T-count 6 we can use BSS).

In [42]:
```
#TODO...
# BSS DECOMP. WHAT REMAINS...

gT = zx.tcount(g)
gDecomp = zx.simulate.find_stabilizer_decomp(g)

print("CUT CONTRIBUTION...")
print("Exact:   ",2**cutcount)

print("\nBSS CONTRIBUTION...")
print("Exact:   ",len(gDecomp))
print("Estimate:",7**(gT/6))

print("\nTOTAL...")
print("Exact:   ",(2**cutcount)*len(gDecomp))
print("Estimate:",(2**cutcount)*(7**(gT/6)))
```

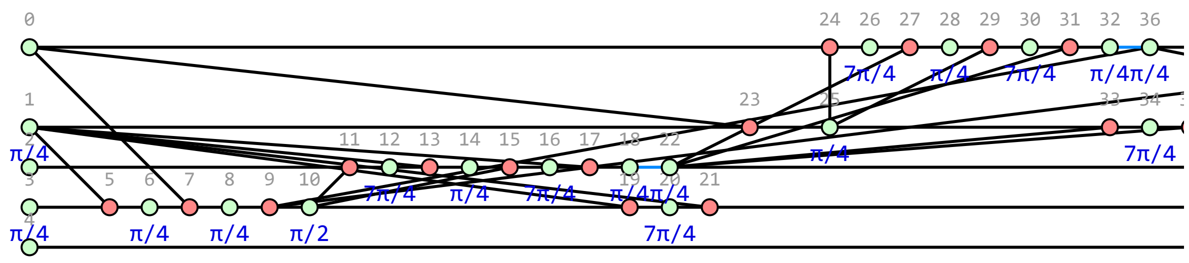
```
CUT CONTRIBUTION...
Exact:    16

BSS CONTRIBUTION...
Exact:    4
Estimate: 7.0

TOTAL...
Exact:    64
Estimate: 112.0
```

So, we made 4 cuts (hence 2^4 = 16 terms), followed by one instance of the BSS (which split each into another 4). So, we estimate that we end up with 64 stabiliser terms. Recall that this was to reduce a circuit of T-count 22, hence this achieves an alpha = 0.27.

This was our estimate (which we could compute very rapidly, regardless of circuit depth). Let's actually try it out for real and see what we get...

In [43]:
```
#Tthreshold = 7 # if a term has <= this many T-gates remaining then we fall back or

debug = True
softDebug = True

cliffCount  = 0 # number of clifford terms (i.e. number of leaves of the cutting tre
scalar      = 0 # total scalar (= sum of the clifford terms)
treeDepth   = 0 # The depth of the cutting tree (i.e. max terms = 2^depth)
```

```
g = h.copy()
gTemp = g.copy(); zx.simplify.full_reduce(gTemp)

zx.draw(g, labels=True, scale=20)
print("Initial T-count = ", zx.tcount(g))
print("Reduced T-count = ", zx.tcount(gTemp))

#gList = apply_cut(11)
#zx.draw(gList[0],labels=True,scale=20)
#zx.draw(gList[1],labels=True,scale=20)

gList = [g]
if (debug): print("\nleaves \t buffer\t |  T-counts")
elif (softDebug): print("\nleaves \t buffer")

while(len(gList)>0):
    treeDepth += 1
    data       = cutGraphs(gList,cliffCount,scalar,False)
    gList      = data[0]
    cliffCount = data[1]
    scalar     = data[2]

    if (debug and len(gList) < 10):
        strLine = str(cliffCount) + " \t " + str(len(gList)) + " \t |  "
        for i in range(len(gList)):
            gTemp = gList[i].copy()
            zx.simplify.full_reduce(gTemp)
            strLine += str(zx.tcount(gTemp)) + "  "
            #zx.draw(gTemp,scale=20,labels=True) #TEMP
        print(strLine)
    elif (softDebug): print(cliffCount,"\t",len(gList))

print("\nNUMBER OF CLIFFORD TERMS:",cliffCount)
print("( tree depth:",treeDepth,")")
```

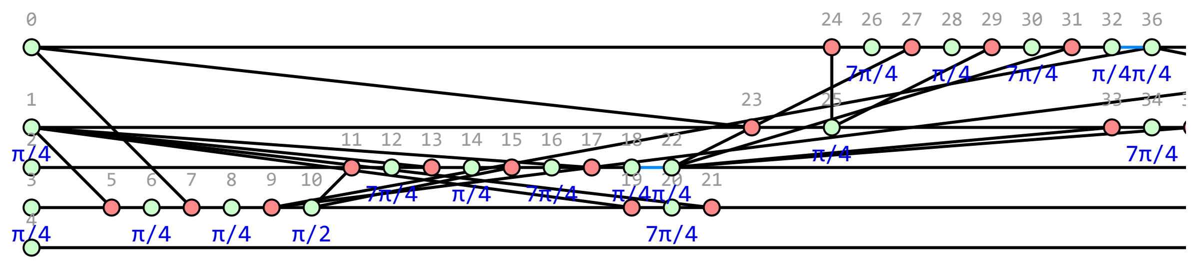


```
Initial T-count =  47
Reduced T-count =  22

leaves   buffer  |  T-counts
0        2       |  21  21
0        4       |  18  18  18  18
0        8       |  15  15  15  15  15  15  15  15
1        15
31       0       |

NUMBER OF CLIFFORD TERMS: 31
( tree depth: 5 )
```

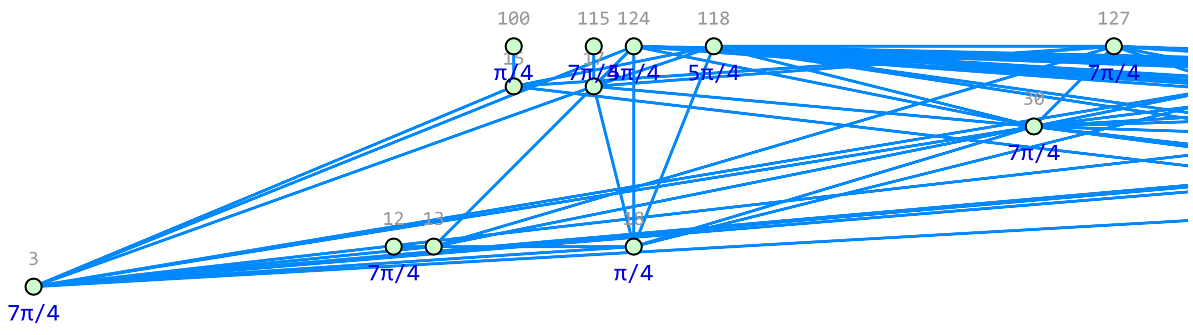
And there we have it! We ended up with 31 stabiliser terms! In reducing a circuit of T-count 22, this achieves an alpha = 0.225 - very good!

And for comparison, let's see what the Kissinger/Wetering (BSS-based) approach would have achieved for this circuit...

In [44]:
```
%%time
g = h.copy()

#g.apply_state("+"*6)  ## TEMP
#g.apply_effect("+"*6) ## TEMP

zx.draw(g, scale=20, labels=True)
print("T-count = ", zx.tcount(g))

zx.simplify.full_reduce(g)

zx.draw(g, scale=20, labels=True)
print("T-count = ", zx.tcount(g))
gT = zx.tcount(g)

gDecomp = zx.simulate.find_stabilizer_decomp(g)
print("\nExact:   ",len(gDecomp))
print("Estimate:",7**(gT/6))

g = h.copy()
```



```
T-count =  47
```



```
T-count =  22

Exact:    303
Estimate: 1255.1418585378788
CPU times: total: 1.66 s
Wall time: 2.24 s
```

303 terms! Almost 10 times as many terms, with an alpha = 0.37, as compared to our alpha = 0.225.

Evidently, our procedural method is very effective! For the paper, we repeated this last section ("Benchmarking 2") for many random circuits of various depths and found that our method was consistently magnitudes better than the BSS approach - achieving typically alpha between 0.1 and 0.2 (as compared to ~0.4 with the BSS method) :D

```
In [ ]:
```