MagneticLattices

March 21, 2024

1 Basic examples

This notebook outlines the construction of Hamiltonians which are not inherently useful on their own, but are of a similar form to what we expect for the most basic simulations where a quantum computer may be useful. This version is a working version and will likely be updated later to include more application relevant examples. The dynamic simulation of these Hamiltonians can be thought of as a necessary but not sufficient condition for demonstrating the viability of a quantum computer as a useful tool for quanutm dynamic simulation.

Note that running this entire notebook should take on the order of an hour. To see smaller cases, simply reduce the sizes of the lattices for the various Hamiltonians.

1.1 Basic Hamiltonian

In this section we consider the time independent transverse field Ising Hamiltonians represented by graphs with a lattice structure. There are more complicated models which would likely be interesting to simulate, but as an initial pass we start by considering two models * 32x32 Triangular Lattice (1024 spins, 2 dimensions) with antiferromagnetic unit couplings * 12x12x12 Cubic Lattice (1728 spins, 3 dimensions) with random unit couplings The transverse field Ising Hamiltonian for a lattice graph, $G = (V, E)$ is as follows:

$$H = \sum_{i \in V} \Gamma_i \sigma_i^x + \sum_{i \in V} h_i \sigma_i^z + \sum_{(i,j) \in E} J_{i,j} \sigma_i^z \sigma_j^z \tag{1}$$

Note that in the instances presented here, we do not consider local longitudinal field terms ($h = 0$), though this may not always be the case. We also do not consider in this example the state preparation circuit that may be required as that is heavily application dependent. For the purposes of these initial experiments, we can assume that the initial state is an eigenstate of either the X or Z basis since the preparation circuits for these have $O(1)$ depth.

To be clear, the dynamic simulations mean simulating the Schrodinger Equation

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = H |\psi(t)\rangle \tag{2}$$

with solution

$$|\psi(t)\rangle = e^{-\frac{i}{\hbar} H t} |\psi(0)\rangle \tag{3}$$

A subgraph of each lattice is plotted below to clarify the graph structure to clarify the graph structure.

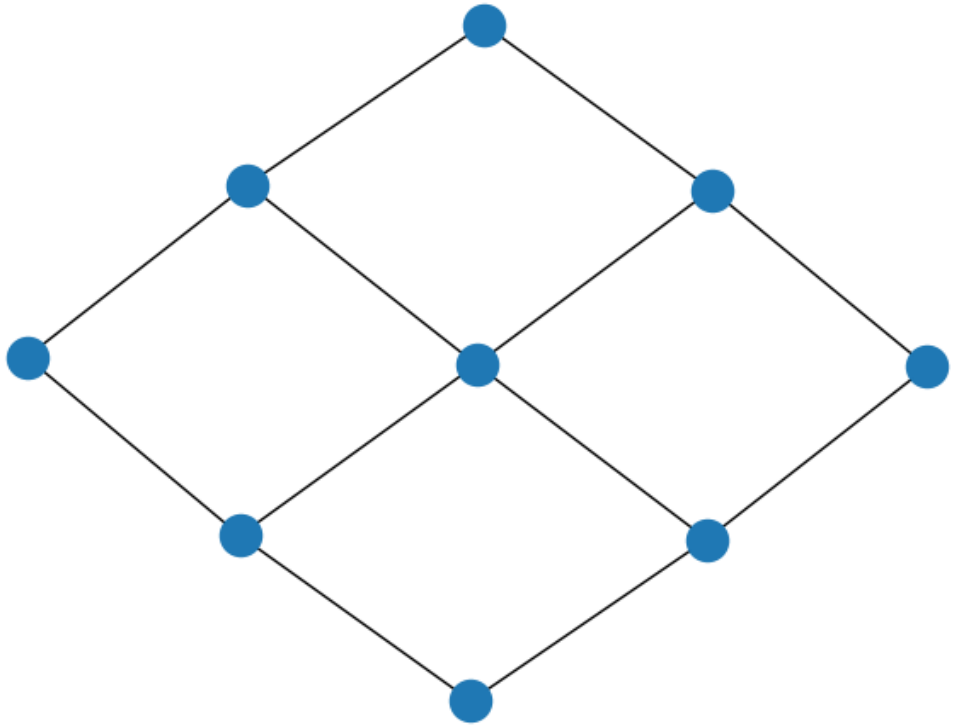
[1]:
```
import os
import random
import numpy as np
import networkx as nx
from networkx import grid_graph
from networkx.classes.graph import Graph
from networkx.generators.lattice import hexagonal_lattice_graph
from qca.utils.utils import plot_histogram
from qca.utils.algo_utils import estimate_qsp, estimate_trotter
from qca.utils.hamiltonian_utils import (nx_triangle_lattice, flatten_nx_graph,
                                         generate_square_hamiltonian,␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator,
                                         assign_directional_triangular_labels,␣
 ↪generate_triangle_hamiltonian,
                                         assign_hexagon_labels)
```

```
/Users/jonhas/anaconda3/lib/python3.11/site-packages/attr/_make.py:918:
RuntimeWarning: Running interpreter doesn't sufficiently support code object
introspection.  Some features like bare super() or accessing __class__ will not
work with slotted classes.
  set_closure_cell(cell, cls)
```

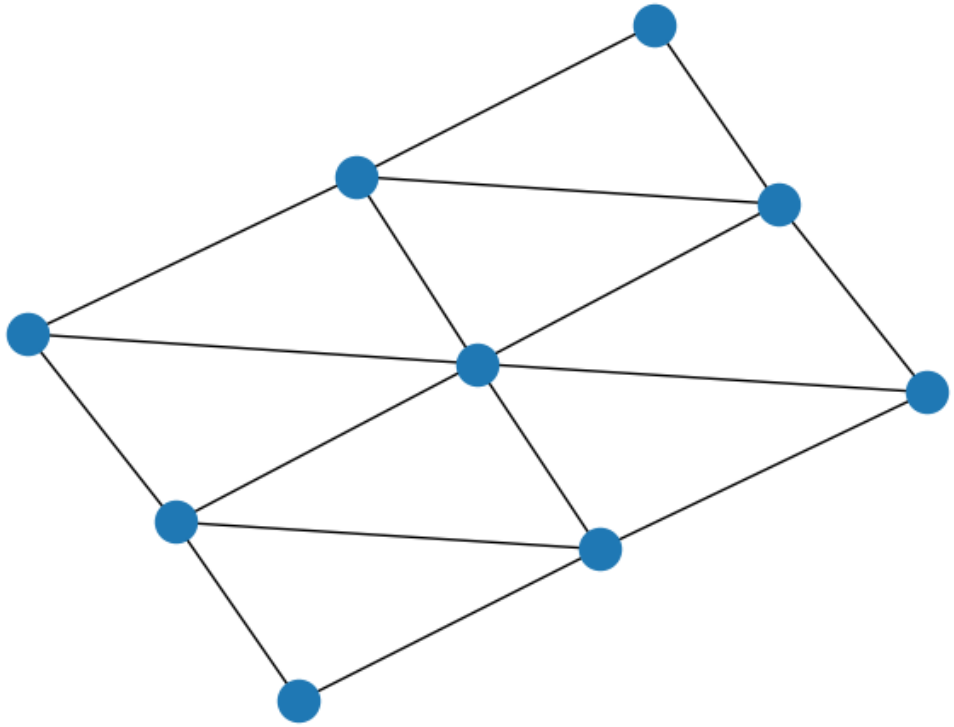
[2]:
```
from pyLIQTR.utils.Hamiltonian import Hamiltonian as pyH
```

[3]:
```
lattice_size = 3
graph_square = grid_graph(dim = (lattice_size, lattice_size))
graph_triangle = nx_triangle_lattice(lattice_size)
graph_cube = grid_graph(dim = (lattice_size, lattice_size, lattice_size))
```

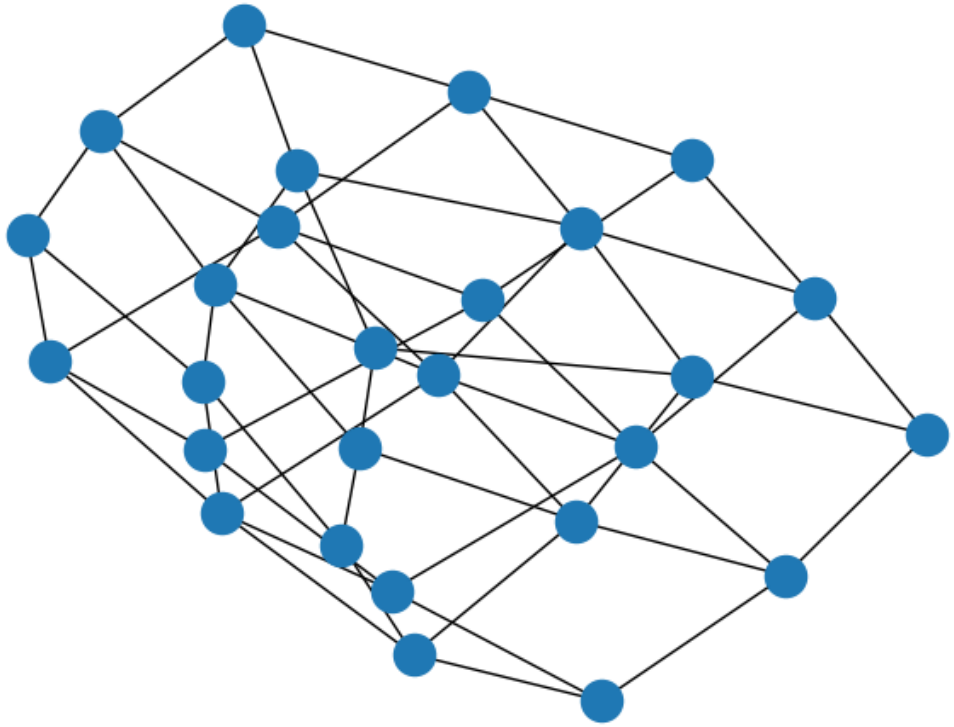
[4]:
```
nx.draw(graph_square)
```



[5]:
```
nx.draw(graph_triangle)
```



[6]:
```
nx.draw(graph_cube)
```



Now that we have an idea of what our Hamiltonians look like, we can begin generating them in a form accepted by pyLIQTR.

[7]:
```
## initializing seed for consistent results for the cubic lattice and␣
 ↪generating Hamiltonians
random.seed(0)
np.random.seed(0)
square_lattice_size=10
triangle_lattice_size=32
cubic_lattice_size=12
test_lattice_size=3

square_hamiltonian = generate_square_hamiltonian(square_lattice_size, dim=2)
cubic_hamiltonian = generate_square_hamiltonian(cubic_lattice_size, dim=3)
triangular_hamiltonian = generate_triangle_hamiltonian(triangle_lattice_size)
```

Now that we have generated our Hamiltonians, we can start generating the circuits for a preferred dynamic simulation method (e.g., QSP, Trotter 4th order, …). Shown below is QSP.

[8]:
```
timesteps=1000
required_precision = 1e-16
```

```
#feeding square Hamiltonian to PyLIQTR for circuit generation
H_square = pyH(square_hamiltonian[0] + square_hamiltonian[1])
H_triangle = pyH(triangular_hamiltonian[0] + triangular_hamiltonian[1])
H_cube = pyH(cubic_hamiltonian[0] + cubic_hamiltonian[1])
```

Now we need to extract resource estimates for each of the subcircuits of the high level circuit. The information which we are interested in is as follows: * Total gate count for full circuit * Total depth for full circuit * Decomposed QASM circuit for each subcircuit * Clifford + T circuits for each subcircuit * Total T gate count for each subcircuit * Total Clifford count for each subcircuit * T gate depth for each subcircuit

This information can be extracted by obtaining the information for each subcircuit and multiplying by the number of repetitions of the subcircuits. In this notebook, we show the process for both QSP and second order Suzuki-Trotter, though this process could likely be generalized to many quantum algorithms. It should be noted that the approach shown here provides a slight over-estimation since it assumes that each of the subcircuits operate in serial. This assumption is valid for both QSP and second order Suzuki-Trotter since the operations which consume the bulk of the circuit volume do operate in serial (Select and Reflect Operations and Trotter steps respectively).

[9]:
```
print('Estimating Square', flush=True)
qsp_circ_square = estimate_qsp(H_square,
                               timesteps,
                               required_precision,
                               'QSP/square_circuits/',
                               hamiltonian_name='square',
                               timestep_of_interest=1)
print('Estimating Triangle', flush=True)
qsp_circ_triangle = estimate_qsp(H_triangle,
                                 timesteps,
                                 required_precision,
                                 'QSP/triangle_circuits/',
                                 hamiltonian_name='triangle',
                                 timestep_of_interest=1)
print('Estimating Cube', flush=True)
qsp_circ_cube = estimate_qsp(H_cube,
                             timesteps,
                             required_precision,
                             'QSP/cube_circuits/',
                             hamiltonian_name='cube',
                             timestep_of_interest=1)
print('Finished estimating', flush=True)
```

```
Estimating Square
Time to generate high level QSP circuit: 0.08332987500034506 seconds
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
0.0001392499998473795 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 2.1916999685345218e-05 seconds
```

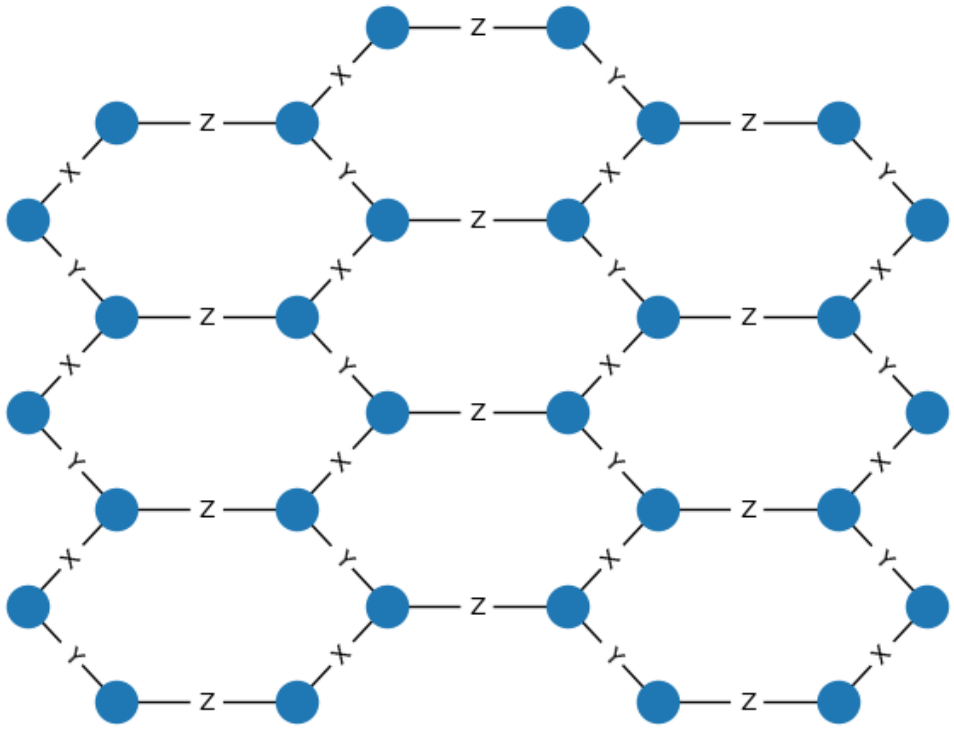
```
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
5.7790999562712386e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.007780042000376852 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.01522908300103154 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 1.00242574999902 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.00023595799939357676 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
Clifford+T: 0.005790667000837857 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 0.08631074999902921 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 0.7279529580009694 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.0010074170004372718
seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.009604582999600098 seconds
Estimating Triangle
Time to generate high level QSP circuit: 5.158821459001047 seconds
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
9.449999924981967e-05 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 1.8708999050431885e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
4.220800110488199e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.006958625001061591 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.12769904199922166 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 7.213463500000216 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.0011460000005172333 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
Clifford+T: 0.005767165999714052 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 2.1373179590009386 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 7.187856292001015 seconds
```

```
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.005298500000208151
seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.00996491700061597 seconds
Estimating Cube
Time to generate high level QSP circuit: 13.26960166599929 seconds
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
8.04999999672873e-05 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 1.833300120779313e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
4.312499913794454e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.006988999999521184 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.21882054199886625 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 10.699302750001152 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.0012747919990943046 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
Clifford+T: 0.0053769160003867 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 4.155046707999645 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 8.489463792000606 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.00903512500008219 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.009699250000267057 seconds
Finished estimating
```

2 More complicated Hamiltonian

There are a few ways to make the simulation task more challenging and more application relevant. The Kitaev honeycomb model is hypothesized to be a good model for the structure of many materials (see for example here). This model is appealing for quantum simulation purposes because while the behavior in the infinite time regime is understood, the behavior with minor perturbations to the model or the dynamics of the model are not so well understood. The Kitaev honeycomb model consists of directionally defined XX, YY, and ZZ couplings on a honeycomb lattice. These assignments are denoted in the plot below by 'X', 'Y', or 'Z' respectively.

[10]:
```
hexagon_graph = hexagonal_lattice_graph(3,3)
pos = nx.get_node_attributes(hexagon_graph, 'pos')
assign_hexagon_labels(hexagon_graph, 'X', 'Y', 'Z')
edge_labels = dict([((n1, n2), d['label']) for n1, n2, d in hexagon_graph.
 ↪edges(data=True)]);
nx.draw(hexagon_graph, pos)
nx.draw_networkx_edge_labels(hexagon_graph, pos,edge_labels = edge_labels);
```



From this graph we can generate the Hamiltonian

$$H_{\text{Kitaev}} = \sum_{(i,j)\in X \text{ edges}} J_{i,j}\sigma_i^x\sigma_j^x + \sum_{(i,j)\in Y \text{ edges}} J_{i,j}\sigma_i^y\sigma_j^y + \sum_{(i,j)\in Z \text{ edges}} J_{i,j}\sigma_i^z\sigma_j^z$$

[11]:
```
def nx_kitaev_terms(g:Graph, p:float) -> list:
    hamiltonian = []
    n = len(g.nodes)
    for (n1, n2, d) in g.edges(data=True):
        label = d['label']
        weight = 1 if random.random() < p else -1
        pauli_string = n * 'I'
        for i in range(len(g)):
```

```
            if i == n1 or i == n2:
                pauli_string = f'{pauli_string[:i]}{label}{pauli_string[i+1:]}'
            else:
                pass
        hamiltonian.append((pauli_string, weight))
    return hamiltonian

def generate_kitaev_hamiltonian(lattice_size:int, weight_prob:float=1):
    graph = hexagonal_lattice_graph(lattice_size,lattice_size)
    assign_hexagon_labels(graph, 'X', 'Y', 'Z')
    graph = flatten_nx_graph(graph)
    H = nx_kitaev_terms(graph, weight_prob)
    return H
```

[12]:
```
# lattice_size_kitaev = 3
lattice_size_kitaev = 32
kitaev_hamiltonian = generate_kitaev_hamiltonian(lattice_size_kitaev)

timesteps = 1000
required_precision = 1e-16
H_kitaev = pyH(kitaev_hamiltonian)
print('Estimating Kitaev', flush=True)
qsp_circ_kitaev = estimate_qsp(H_kitaev, timesteps,
                               required_precision,
                               'QSP/kitaev_circuits/',
                               hamiltonian_name='kitaev',
                               timestep_of_interest=1)
print('Finished Estimating', flush=True)
```

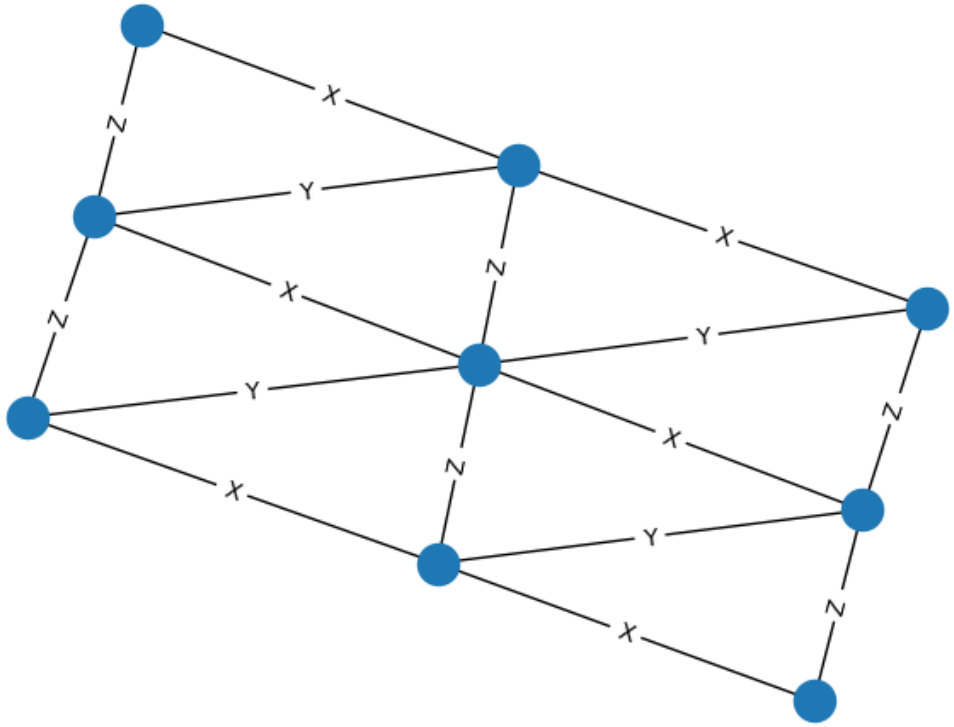
```
Estimating Kitaev
Time to generate high level QSP circuit: 10.782345749999877 seconds
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
0.00010645900147210341 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 2.5167000785586424e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
4.508299934968818e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.008271624999906635 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.1268073329993058 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 12.766324958000041 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.0013629999994009268 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
```

```
Clifford+T: 0.005630667001241818 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 3.2289712500005407 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 8.597125958998731 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.005721875000745058
seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.009585792000507354 seconds
Finished Estimating
```

[13]:
```
def generate_directional_triangular_hamiltonian(lattice_size:int, weight_prob:
 ↪float = 1):
    graph = nx_triangle_lattice(lattice_size)
    assign_directional_triangular_labels(graph,lattice_size)
    graph = flatten_nx_graph(graph)
    H = nx_kitaev_terms(graph, weight_prob)
    return H

g_triangle = nx_triangle_lattice(3)
assign_directional_triangular_labels(g_triangle,3)
```

[14]:
```
graph_triangle = nx_triangle_lattice(3)
assign_directional_triangular_labels(graph_triangle, 3)
pos = nx.spring_layout(graph_triangle)
edge_labels = dict([((n1, n2), d['label']) for n1, n2, d in graph_triangle.
 ↪edges(data=True)]);
nx.draw(graph_triangle, pos)
nx.draw_networkx_edge_labels(graph_triangle, pos,edge_labels = edge_labels);
```



[15]:
```
lattice_size_directional_triangle = 32
# lattice_size_directional_triangle = 3
directional_triangle_hamiltonian =␣
 ↪generate_directional_triangular_hamiltonian(lattice_size_directional_triangle)

timesteps = 1000
required_precision = 1e-16
timestep_of_interest = 1 # sim_time
H_directional_triangle = pyH(directional_triangle_hamiltonian)

print('Estimating Directional Triangle', flush=True)
qsp_circ_directional_triangle = estimate_qsp(H_directional_triangle, timesteps,
                                             required_precision, 'QSP/
 ↪directional_triangle_circuits/',
                                            ␣
 ↪hamiltonian_name='directional_triangle',
                                             timestep_of_interest=1)
print('Finished Estimating', flush=True)
```

```
Estimating Directional Triangle
Time to generate high level QSP circuit: 4.367456165999101 seconds
```

```
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
0.00012187500033178367 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 1.887500002339948e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
4.824999996344559e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.007282624999788823 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.12934874999882595 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 13.96208787499927 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.0011908750002476154 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
Clifford+T: 0.0058305830007157056 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 2.769196749999537 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 9.05375654099953 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.005342458000086481
seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.00963308300015342 seconds
Finished Estimating
```

3 Trotter imeplementations

Now we can do all of this again for second order Suzuki Trotter rather than QSP. Below we will see how to construct a single trotter step of second order Suzuki-Trotter for these hamiltonians along with a loose upper bound for the number of trotter steps required. According to the documentation from openfermion the trotter step estimate for the second order Suzuki-Trotter expansion.

[16]:
```
# Translating the hamiltonians from above into a form usable by openfermion
openfermion_hamiltonian_square =␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_square)
openfermion_hamiltonian_triangle =␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_triangle)
openfermion_hamiltonian_cube =␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_cube)
openfermion_hamiltonian_kitaev =␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_kitaev)
```

```
openfermion_hamiltonian_directional_triangle =␣
 ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_directional_triangle)
```

3.1 Estimates

Trotterizing the Hamiltonians and writing estimates to files

[17]:
```
# defining precision required for the trotterized circuit
energy_precision = 1e-6
timesteps=1000
```

[18]:
```
print('Estimating Square', flush=True)
cpt_trotter_square = estimate_trotter(openfermion_hamiltonian_square,
                                      timesteps, energy_precision,
                                      'Trotter/square_circuits/',
                                      hamiltonian_name='square')
print('Estimating Triangle', flush=True)
cpt_trotter_triangle = estimate_trotter(openfermion_hamiltonian_triangle,
                                        timesteps, energy_precision,
                                        'Trotter/triangle_circuits/',
                                        hamiltonian_name='triangle')
print('Estimating Cube', flush=True)
cpt_trotter_cube = estimate_trotter(openfermion_hamiltonian_cube,
                                    timesteps, energy_precision,
                                    'Trotter/cube_circuits/',
                                    hamiltonian_name='cube')
print('Estimating Kitaev', flush=True)
cpt_trotter_kitaev = estimate_trotter(openfermion_hamiltonian_kitaev,
                                      timesteps, energy_precision,
                                      'Trotter/kitaev_circuits/',
                                      hamiltonian_name='kitaev')
print('Estimating Directional Triangle', flush=True)
cpt_trotter_directional_triangle =␣
 ↪estimate_trotter(openfermion_hamiltonian_directional_triangle,
                                                    timesteps,␣
 ↪energy_precision, 'Trotter/directional_triangle_circuits/',
                                                   ␣
 ↪hamiltonian_name='directional_triangle')
print('Finished with estimates', flush=True)
```

```
Estimating Square
Time to estimate number of trotter steps required (5366564): 0.01868758399905346
seconds
Time to find term ordering: 0.0011280840008112136 seconds
Time to generate trotter circuit from openfermion: 1.0420008038636297e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 2.1824925000000803
seconds
```

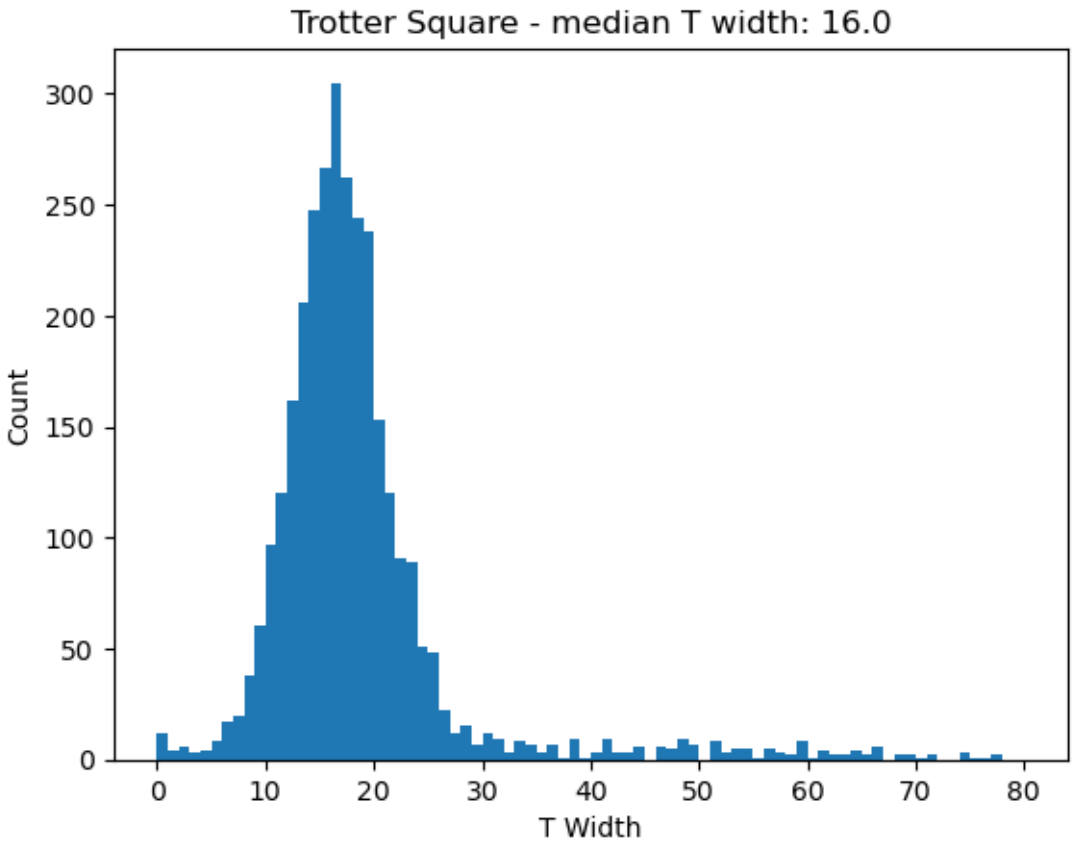
```
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
0.00010183300037169829 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 2.0666999262175523e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
4.758300019602757e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.0009515839992673136 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
0.00011466599971754476 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.00017304200082435273 seconds
Estimating Triangle
Time to estimate number of trotter steps required (21707142): 2.4670277909990546
seconds
Time to find term ordering: 0.024339332998351892 seconds
Time to generate trotter circuit from openfermion: 1.1250012903474271e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 44.691588333000254
seconds
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
9.495800077274907e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 1.741699998092372e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
4.099999932805076e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.024086249999527354 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
0.00013591599963547196 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.00016274999870802276 seconds
Estimating Cube
Time to estimate number of trotter steps required (27573901): 6.701031333999708
seconds
Time to find term ordering: 0.038902124999367516 seconds
Time to generate trotter circuit from openfermion: 1.0840012691915035e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 67.10455520800133
seconds
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
0.00010570800077402964 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 1.833300120779313e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
4.0666998756933026e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.007196000000476488 seconds
```

```
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
9.583300015947316e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.00013970800137030892 seconds
Estimating Kitaev
Time to estimate number of trotter steps required (222782406):
1.8685654580003757 seconds
Time to find term ordering: 0.020169500001429697 seconds
Time to generate trotter circuit from openfermion: 1.0420008038636297e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 37.945370167000874
seconds
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
9.833299918682314e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 1.8665999959921464e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
4.3000000005122274e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.00012791700100933667 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
9.579099969414528e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.0001520000005257316 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
4.279200038581621e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.005919958999584196 seconds
Estimating Directional Triangle
Time to estimate number of trotter steps required (433303589):
1.7011602090005908 seconds
Time to find term ordering: 0.02558583300015016 seconds
Time to generate trotter circuit from openfermion: 1.4170000213198364e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 38.07089270799952
seconds
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
0.0001032090003718622 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 1.7874999684863724e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
3.816599928541109e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.0001140000003942987 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
8.549999984097667e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.00012720800077659078 seconds
```

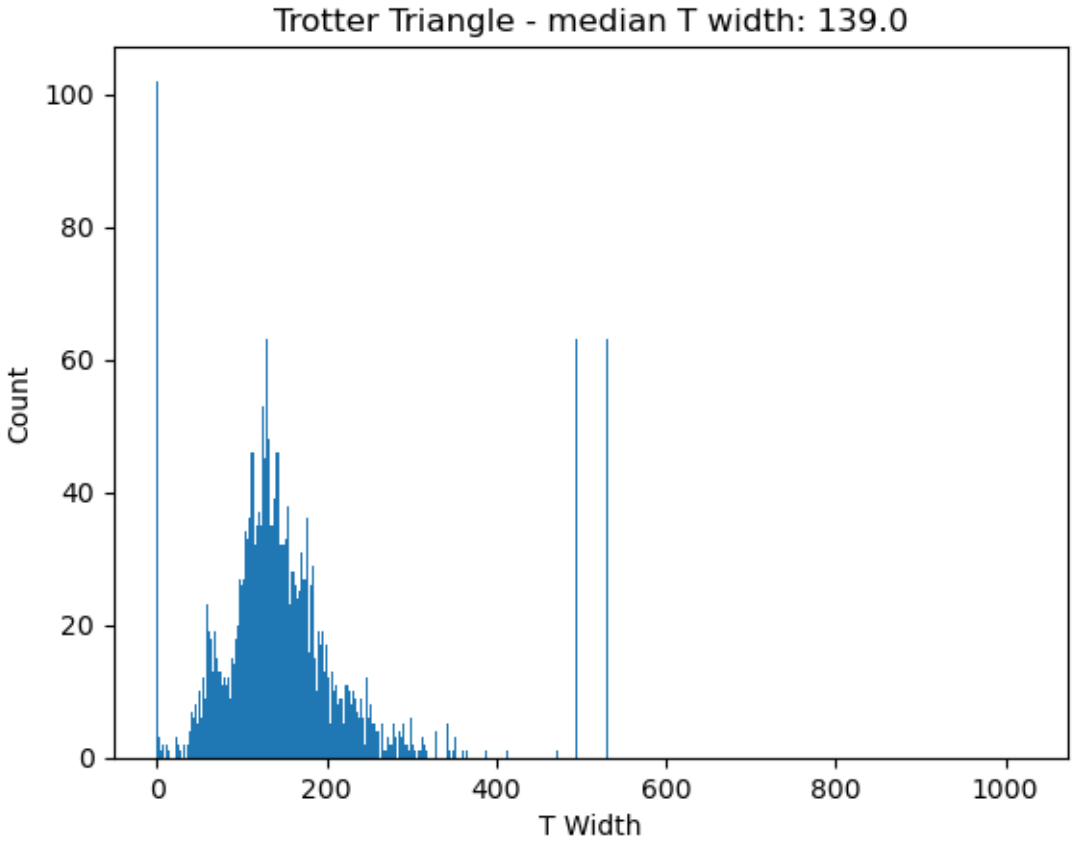
```
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
3.758300044864882e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.007114500000170665 seconds
Finished with estimates
```

[19]:
```
figdir = 'Trotter/Figures/'
widthdir = 'Trotter/Widths/'
if not os.path.exists(figdir):
    os.makedirs(figdir)
if not os.path.exists(widthdir):
    os.makedirs(widthdir)
```

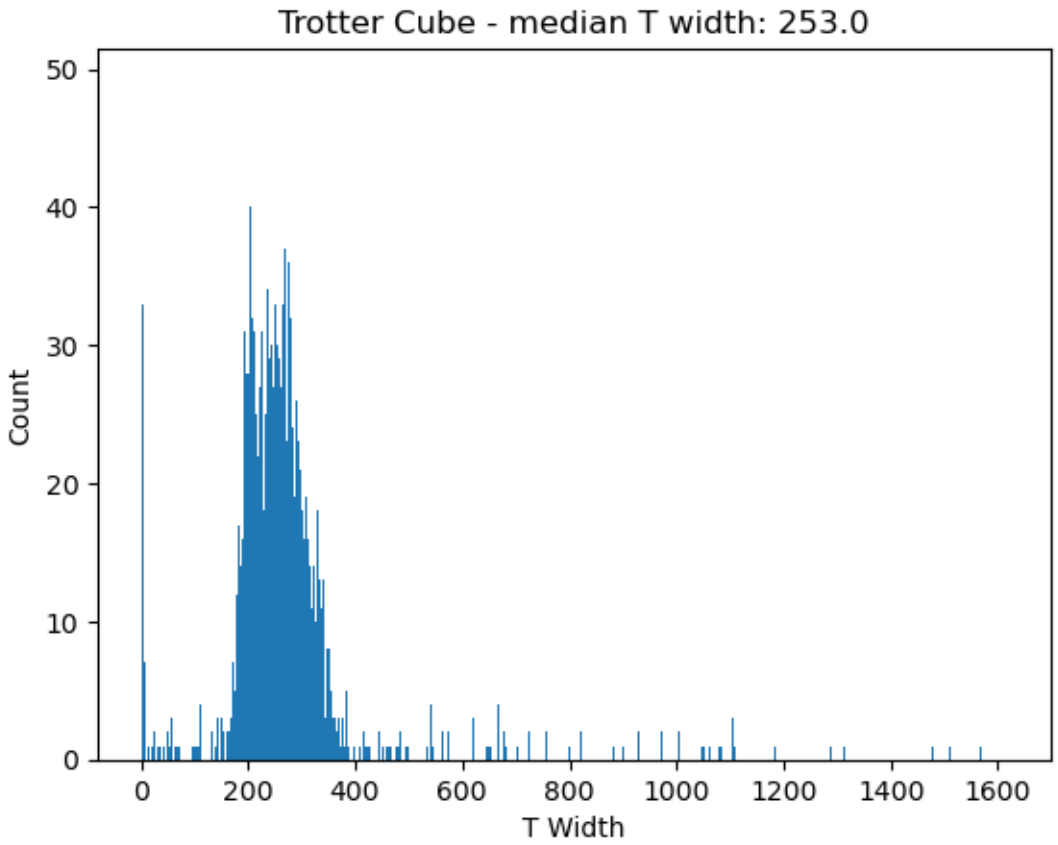
[20]:
```
plot_histogram(
    cpt_trotter_square,
    'Trotter Square',
    figdir,
    widthdir
)
```



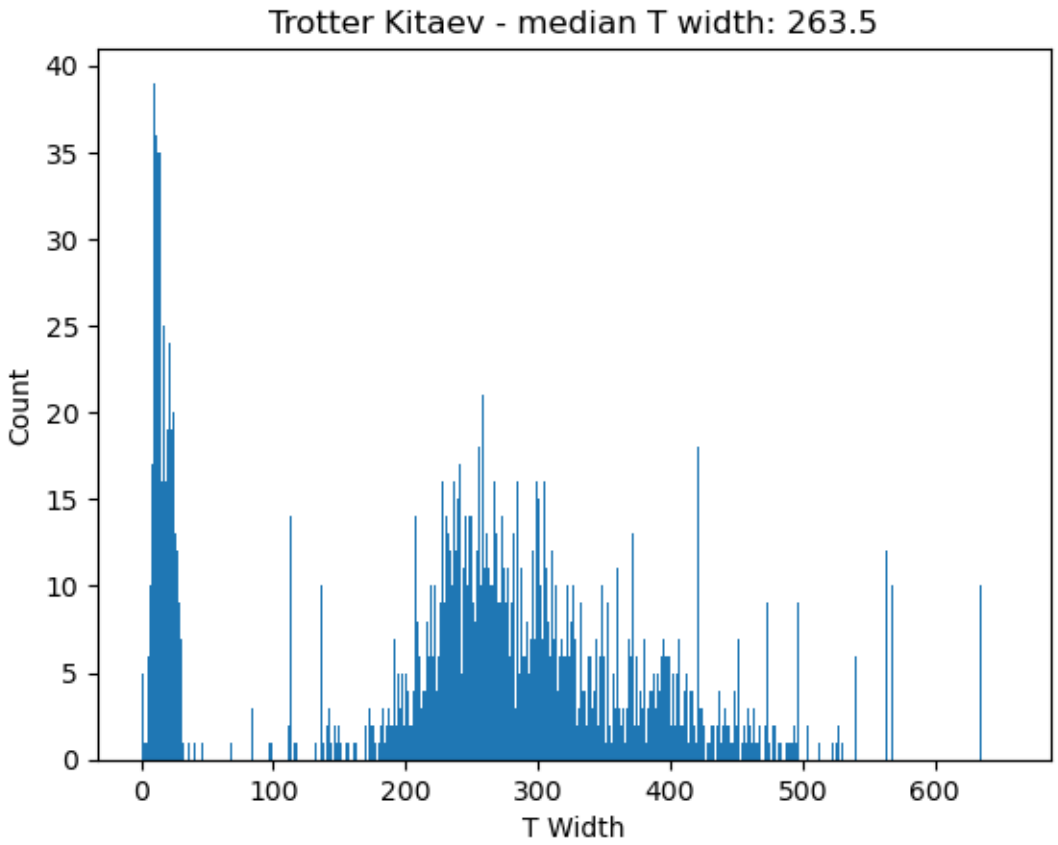
[21]:
```
plot_histogram(
    cpt_trotter_triangle,
    'Trotter Triangle',
    figdir,
    widthdir
)
```



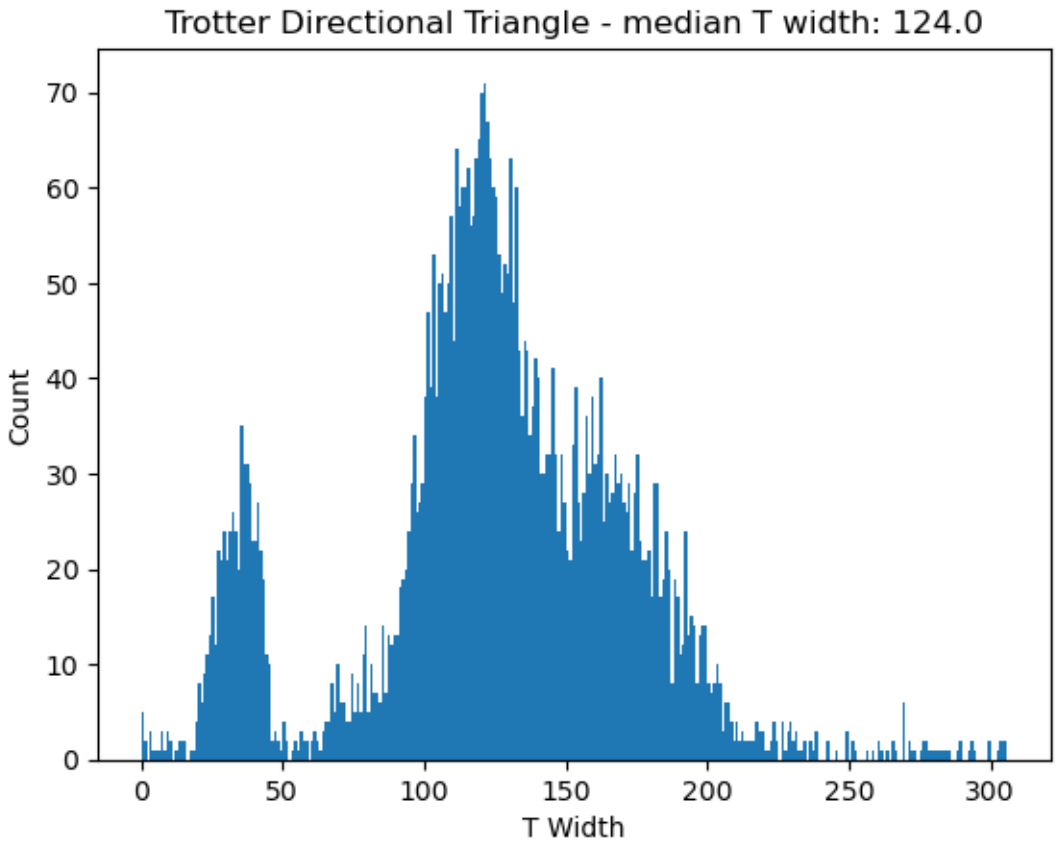
[22]:
```
plot_histogram(
    cpt_trotter_cube,
    'Trotter Cube',
    figdir,
    widthdir
)
```



[23]:
```
plot_histogram(
    cpt_trotter_kitaev,
    'Trotter Kitaev',
    figdir,
    widthdir
)
```



[24]:
```
plot_histogram(
    cpt_trotter_directional_triangle,
    'Trotter Directional Triangle',
    figdir,
    widthdir
)
```


Trotter Directional Triangle - median T width: 124.0
Count
T Width
0
10
20
30
40
50
60
70
0
50
100
150
200
250
300

[ ]: