graph_energy

March 21, 2024

1 Graph Based Spin Lattice

1.0.1 Find lowest ground state of simple Ising Hamiltonian

For a graph, $G = (E, V)$, defined by a set of edges, E, and vertices, V, we want to represent an Ising model, where the edge weights, w_{ij} are given by the spin interactions, i.e., $w_{ij} = J_{ij}$.

Given a configuration of spins (e.g., $\uparrow\downarrow\downarrow\uparrow\downarrow$) we can define the energy using what is referred to as an Ising Hamiltonian:

$$\hat{H} = \sum_{(i,j)\in E} J_{ij} s_i s_j$$

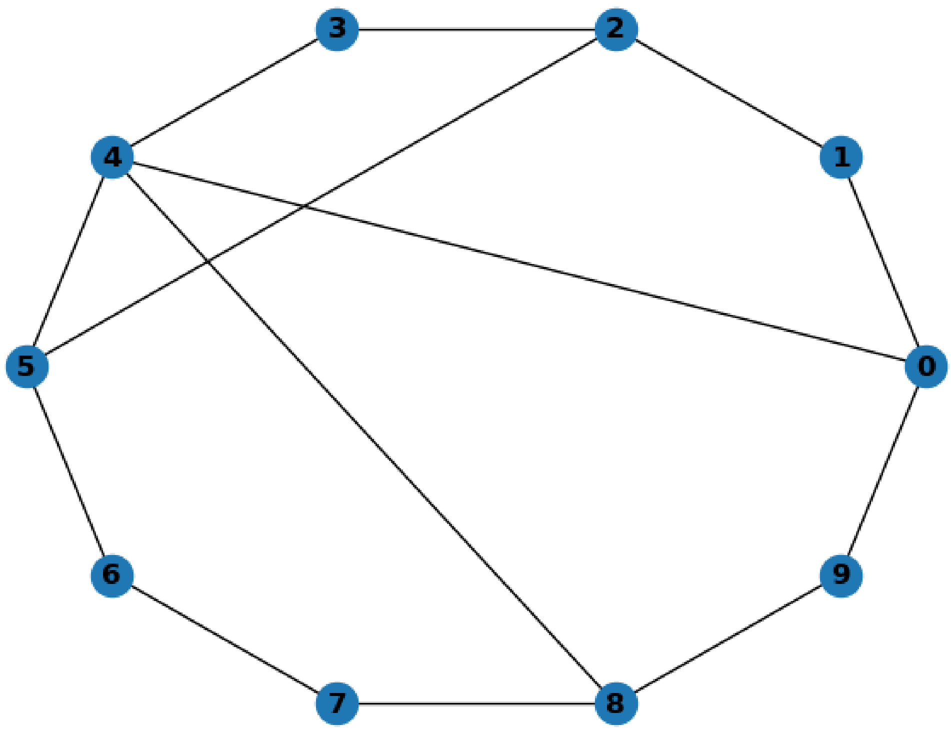
where, $s_i = 1$ if the i^{th} spin is `up` and $s_i = -1$ if it is `down`, and the sumation runs over all edges in the graph.

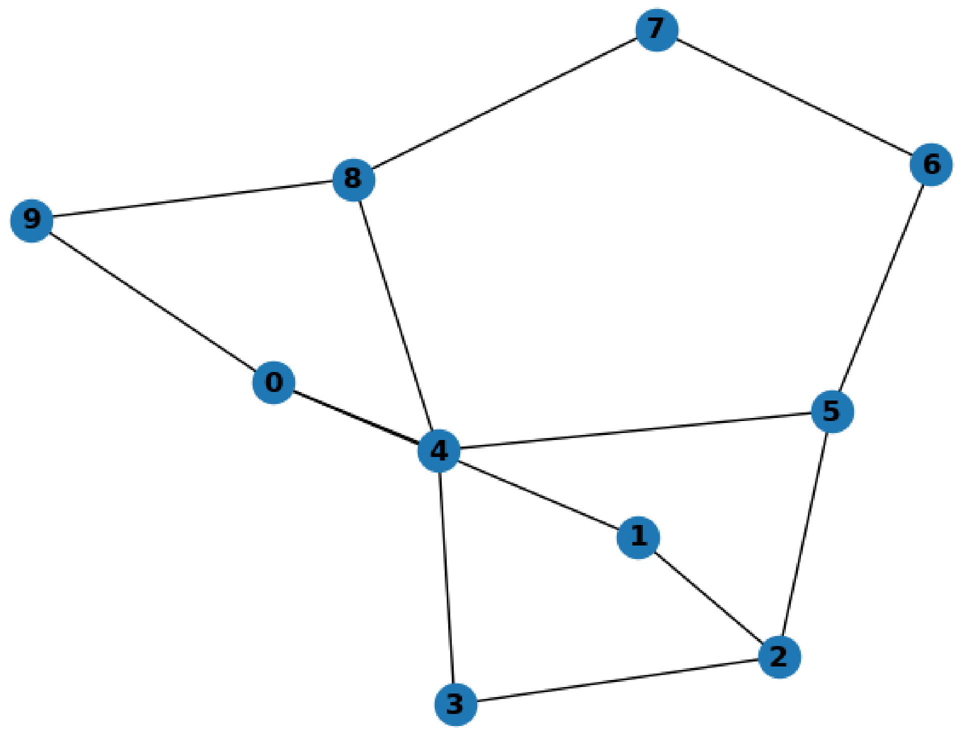
[ ]:
```
# Load relevant libraries. If you have errors you probably need to install them␣
↪into your conda env

import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
# import scipy
random.seed(2)
```

1.0.2 Create a graph that defines the Ising interactions

[ ]:
```
G = nx.Graph()
G.add_nodes_from([i for i in range(10)])
G.add_edges_from([(i,(i+1)% G.number_of_nodes() ) for i in range(10)])
G.add_edge(2,5)
G.add_edge(4,8)
G.add_edge(4,0)
for e in G.edges:
    G.edges[e]['weight'] = 1.0

# Now Draw the graph. First we will draw it with the nodes arranged on the␣
↪circle, then we will draw the same graph
# with the position of the nodes optimized for easier visualization
```

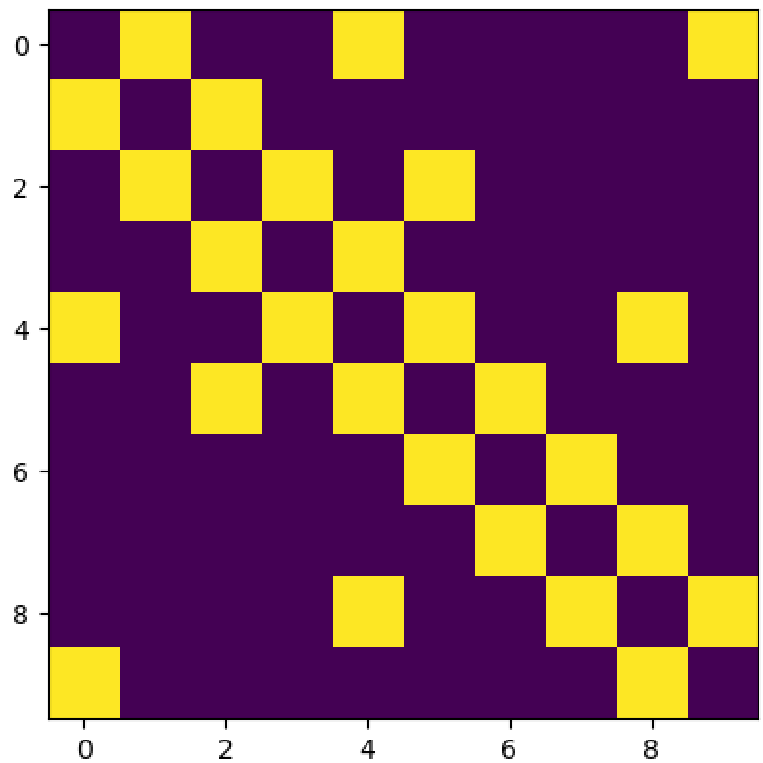
```python
plt.figure(1)
nx.draw(G, with_labels=True, font_weight='bold', pos=nx.circular_layout(G))
plt.figure(2)
nx.draw(G, with_labels=True, font_weight='bold')
plt.show()
```





[ ]:
```
A = nx.adjacency_matrix(G).todense()
display(A)
plt.imshow(A);
```

```
array([[0., 1., 0., 0., 1., 0., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 1., 0., 0., 0., 0., 0.],
       [1., 0., 0., 1., 0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 1., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 1., 0., 0., 1., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 0., 1., 0.]])
```



1.0.3 Add your BitString class below

[ ]:
```python
import numpy as np
import math


class BitString:
    """
    Simple class to implement a config of bits
    """
    def __init__(self, N):
        self.N = N
        self.config = np.zeros(N, dtype=int)

    def __repr__(self):
        selfString = ''
        for bit in self.config:
            selfString += str(bit)
        return selfString

    def __eq__(self, other):
        return (self.config == other.config).all()
```

```python
    def __len__(self):
        return self.N

    def on(self):
        num_on = 0
        for bit in self.config:
            if bit == 1:
                num_on += 1
        return num_on

    def off(self):
        num_off = 0
        for bit in self.config:
            if bit == 0:
                num_off += 1
        return num_off

    def flip_site(self,i):
        self.config[i] ^= 1

    def int(self):
        return int(str(self), 2)

    def set_config(self, s:list[int]):
        self.config = s

    def set_int_config(self, dec:int):
        self.config = np.zeros(self.N, dtype=int)

        i = 1
        while dec != 0:
            self.config[-i] = dec % 2
            dec = dec // 2
            i += 1
```

[ ]:
```python
def energy(bs: BitString, G: nx.Graph):
    """Compute energy of configuration, `bs`

        .. math::
            E = \\left<\\hat{H}\\right>

    Parameters
    ----------
    bs   : Bitstring
        input configuration
    G    : Graph
```

```
        input graph defining the Hamiltonian
    Returns
    -------
    energy  : float
        Energy of the input configuration
    """
    # energy = sum of J_ij * S_i * S_j = J_ij * translation_thingy(S_i, S_j)
    # J represents whether or not the qubits are adjacent

    # translating 0 and 1 to -1 and 1
    def z1_to_n1(bit1, bit2):
        if bit1 == bit2:
            return 1
        else:
            return -1

    # array_J = nx.adjacency_matrix(G).todense()
    nrg = 0
    # for i in range(bs.N):
    #     for j in range(i, bs.N):
    #         nrg += array_J[i,j] * z1_to_n1(bs.config[i], bs.config[j])
    for (i,j) in G.edges:
        nrg += G.edges[(i,j)]['weight'] * z1_to_n1(bs.config[i], bs.config[j])

    return nrg
```

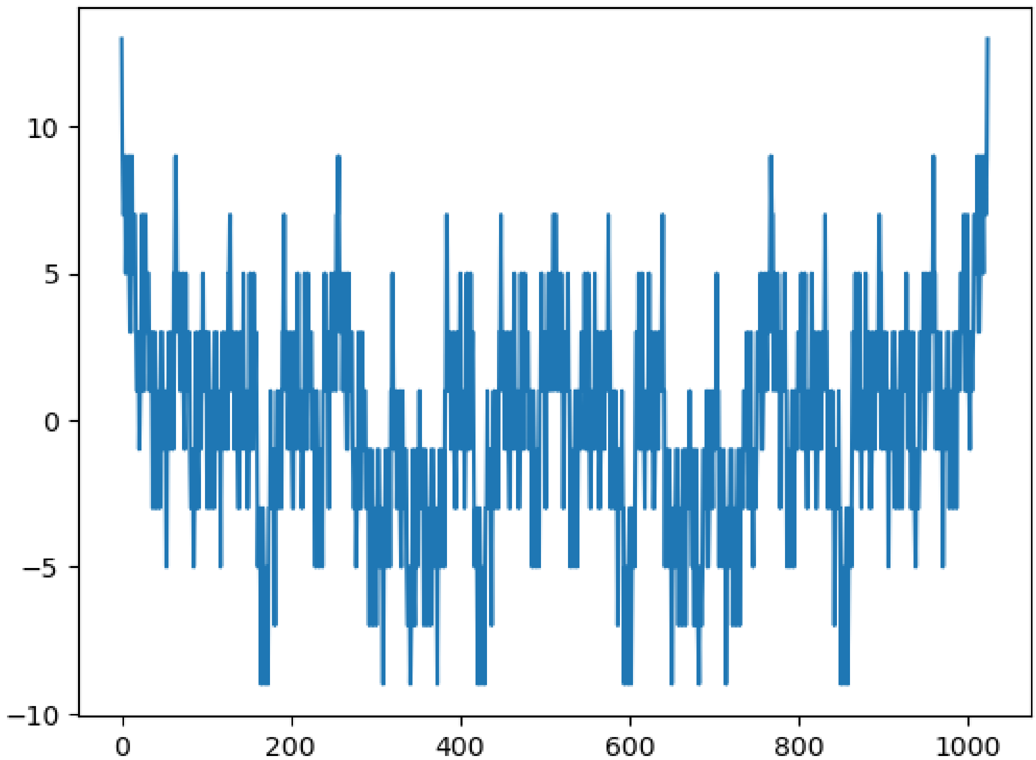
1.0.4 Naive minimization

Loop over all configurations and compute energies to find the lowest

[ ]:
```
x = [] # Store list of indices
y = [] # Store list of energies
xmin = None # configuration of minimum energy configuration
emin = 0 # minimum of energy
my_bs = BitString(10)

for i in range(2 ** 10):
    my_bs.set_int_config(i)
    y.append(energy(my_bs, G))
    x.append(i)
emin = min(y)
xmin = y.index(emin)
print(emin)

# Print out all the energies
plt.plot(x,y);
```

```
# Print out the lowest energy configuration
my_bs.set_int_config(xmin)
print(" Lowest energy %12.8f: %s" %(emin, my_bs), ' (', my_bs.int(), ') ',␣
↪sep='')

assert(abs(energy(my_bs, G) - -9) < 1e-12)
```

```
-9.0
 Lowest energy  -9.00000000: 0010100101 (165)
```



1.0.5 Visualize ground state

Print out the graph again, this time coloring each node according to it's state (0 or 1)

```
[ ]: print("Configuration: %s" %my_bs)
nx.draw(G, node_color=my_bs.config)
```

```
Configuration: 0010100101
```