Solving Maxcut on the Ising Model

James Saslow

10/3/2024

Solving The Maximum Cut Problem on an Ising Machine

The maximum cut problem is formulated as follows:

$$C = \sum_{<i,j>} \frac{1 - \sigma_i \sigma_j}{2}$$

where σ_i and σ_j are binary variables (they are only allowed to take on values of $+1$ or -1)

Where the objective is to determine the statevector $\ket\psi$ that *maximizes* C. (Feel free to read more here https://en.wikipedia.org/wiki/Maximum_cut. Moral of the story, max-cut is an important problem in computer science, with a bunch of applications i.e. partitioning, computer vision, social network analysis, etc). However, we are more interested in how to solve it instead of what it does.

To solve it using an ising machine, we need to somehow reformulate C into a Hamiltonian. When we say we're maximizing C, we're really maximizing $-\sum_{<i,j>} \sigma_i \sigma_j$, which, within a minus sign, is equivalent to minimizing $\sum_{<i,j>} \sigma_i \sigma_j$. Thus our Hamiltonian we want is

$$H = \sum_{<i,j>} \sigma_i \sigma_j$$

whose ground state wavefunction $\ket\psi$ is the solution to max-cut with eigen-energy magnitude proportional to the maximum number of cuts.

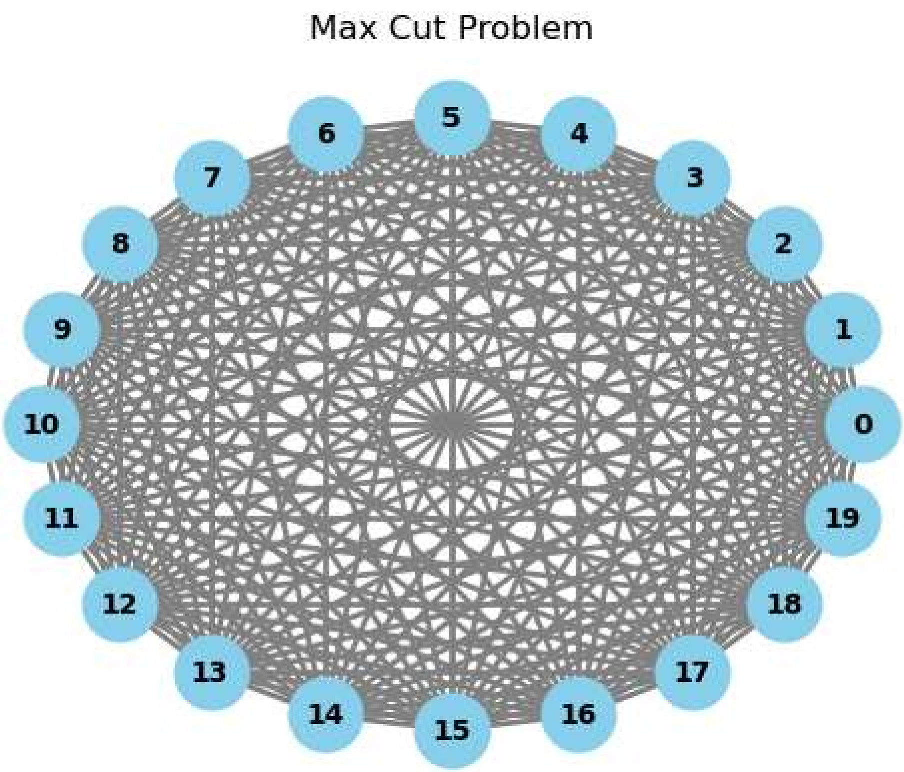
In [36]:
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
```

In [37]:
```
mu = 1
stdev = 0

np.random.normal(mu,stdev)
```

Out[37]: `1.0`

In [38]:
```
# Number of nodes
n = 20

# Create a complete graph with n nodes
```

```
G = nx.complete_graph(n)


# All edges have a weight = 1
for (i, j) in G.edges():
    G[i][j]['weight'] = np.random.normal(mu, stdev) #np.random.random() #1

plt.title('Max Cut Problem')
pos = nx.circular_layout(G)

nx.draw(G, pos, with_labels=True, node_size=700, node_color="skyblue",
        font_size=10, font_color="black", font_weight="bold", width=2, edge_color="
plt.show()
```



In [39]:
```
def get_energy(lattice):

    energy = 0
    edge_array = [] # Creating an array of all edge node connections i.e. [(0,1), (
    for edge in G.edges:
        edge_array.append(edge)

    target_nodes = np.arange(0, n) # Generating a list of all nodes
    x,y = np.transpose(edge_array) # Separating the edge_array in 2 arrays i.e. x =

    num_edges = len(edge_array)

    for target_node in target_nodes:

        index_x = np.where(x == target_node)[0] # Calculating where the target node
        index_y = np.where(y == target_node)[0] # Calculating where the target node
        index   = np.hstack((index_x,index_y)) # Recording where the target node is
```

```
        # Searching all connections with the target index
        connections = []
        for i in index:
            connections.append(edge_array[i])

        for c in connections:
            J = G.edges[c]["weight"] # Retrieving the interaction parameter J for e

            i,j = c   # Finding the i and j indices associated with the lattice con
            sigma_i = lattice[i] # Spin value on site i
            sigma_j = lattice[j] # Spin value on site j

            energy += J*sigma_i*sigma_j / 2 # Dividing by 2 to avoid double countin

    cut = (num_edges - energy)/2 # Formula to relate the number of cuts with the ei

    return energy, cut
```

In [40]:

```
def metropolis_update(lattice, T):
    lattice_copy = np.copy(lattice) # Making a copy of the lattice
    beta = 1/T # Inverse Temperature

    edge_array = [] # Creating an array of all edge node connections i.e. [(0,1), (
    for edge in G.edges:
        edge_array.append(edge)

    target_node = np.random.randint(0, n-1) # Selecting a node at random

    lattice_spin_flip = np.copy(lattice) # Making a copy of the lattice
    lattice_spin_flip[target_node] *= -1 # Exact copy of the lattice but the target

    x,y = np.transpose(edge_array) # Separating the edge_array in 2 arrays i.e. x =

    index_x = np.where(x == target_node)[0] # Calculating where the target node is
    index_y = np.where(y == target_node)[0] # Calculating where the target node is

    index = np.hstack((index_x,index_y)) # Recording where the target node is invol

    # Searching all connections with the target index
    connections = []
    for i in index:
        connections.append(edge_array[i])

    E_i = 0 # Initial Energy Before the spin-flip
    for c in connections:
        J = G.edges[c]["weight"] # Retrieving the interaction parameter J for each

        i,j = c   # Finding the i and j indices associated with the lattice connect
        sigma_i = lattice[i] # Spin value on site i
        sigma_j = lattice[j] # Spin value on site j

        E_i += J*sigma_i*sigma_j

    E_f = 0 # Final Energy after the spin-flip
    for c in connections:
```

```
        J = G.edges[c]["weight"]
        i,j = c
        sigma_i = lattice_spin_flip[i]
        sigma_j = lattice_spin_flip[j]

        E_f += J*sigma_i*sigma_j

    dE = E_f-E_i # Calculating the difference in energy

    # Case I
    if dE < 0: # Accepting a spin flip if it lowers the energy
        ans = lattice_spin_flip

    # Case II
    if dE >= 0:
        if np.random.random() < np.exp(-beta*dE): # Only accepting the spin flip wi
            ans = lattice_spin_flip
        else:
            # If spin is not excepted with Boltzman probability, the lattice stays
            dE = 0 # Not using the spin flipped lattice => No energy change
            ans = lattice_copy # Flipped configuration is not accepted, defaulting

    # Returning the lattice statevector "ans" and the change in energy "dE"
    return ans, dE
```

In [41]:
```
lattice = 2*np.random.randint(0,2,n)-1
E0, cut0 = get_energy(lattice)
E = [E0]
M = []    # Magnetization


print("Randomly Generated Initial Lattice: ", lattice)
print("Starting Energy: ", E0)
print("Starting Cut: ", cut0)
```

```
Randomly Generated Initial Lattice:  [ 1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1  1 -
1  1 -1 -1  1]
Starting Energy:  22.0
Starting Cut:  84.0
```

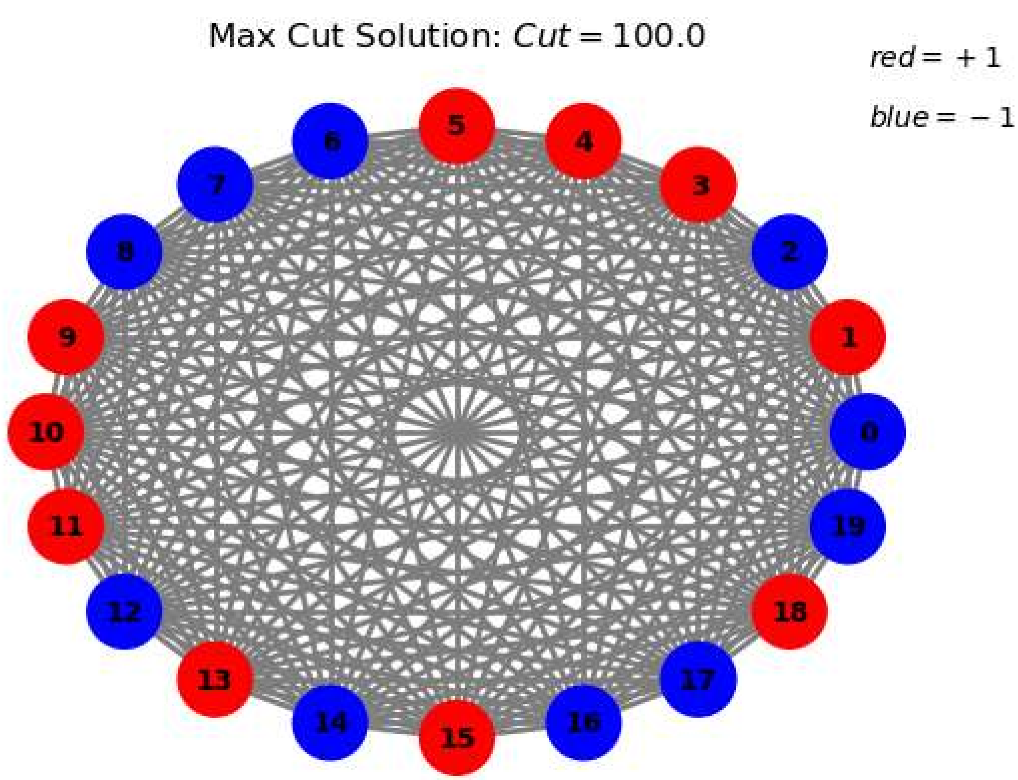
In [42]:
```
# Metropolis Algorithm for Determining the solution to the Ising model (Solution to

itn = 100 # Number of metropolis iterations
for i in range(itn):
    lattice, dE = metropolis_update(lattice, T = 0.1) # T = Temperature
    E.append(E[-1] + dE)
    M.append(sum(lattice))
```

In [43]:
```
# Final Energy of the system
min_energy, max_cut = get_energy(lattice)

print("Ising Solution: ", lattice)
print("Min Energy: ", min_energy)
print("Max Cut: ", max_cut)
```

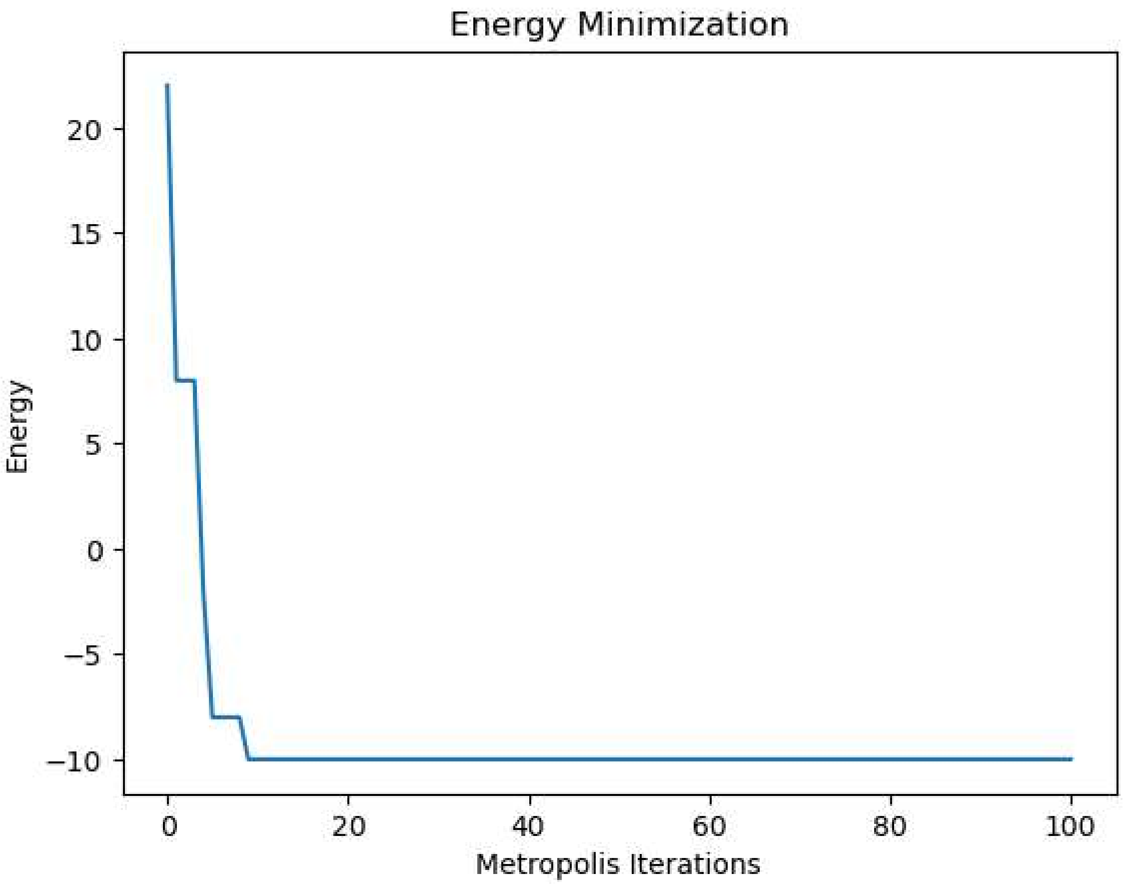
```
Ising Solution:  [ 1 -1  1 -1 -1 -1  1  1  1 -1 -1 -1  1 -1  1 -1  1  1 -1  1]
Min Energy:  -10.0
Max Cut:  100.0
```

In [44]:
```
# Visualizing the max-cut solution

colors = []
for i in lattice:
    if i == +1:
        colors.append("blue") # +1 class of Nodes
    else:
        colors.append("red") # -1 Class of Nodes


print("Red = +1 ")
print("Blue = -1")

plt.title('Max Cut Solution: $Cut = ' + str(max_cut) + "$")
pos = nx.circular_layout(G)

plt.text(1,1.2,'$red = +1$')
plt.text(1,1.0,'$blue = -1$')

nx.draw(G, pos, with_labels=True, node_size=700, node_color=colors,
        font_size=10, font_color="black", font_weight="bold", width=2, edge_color="

plt.show()
```

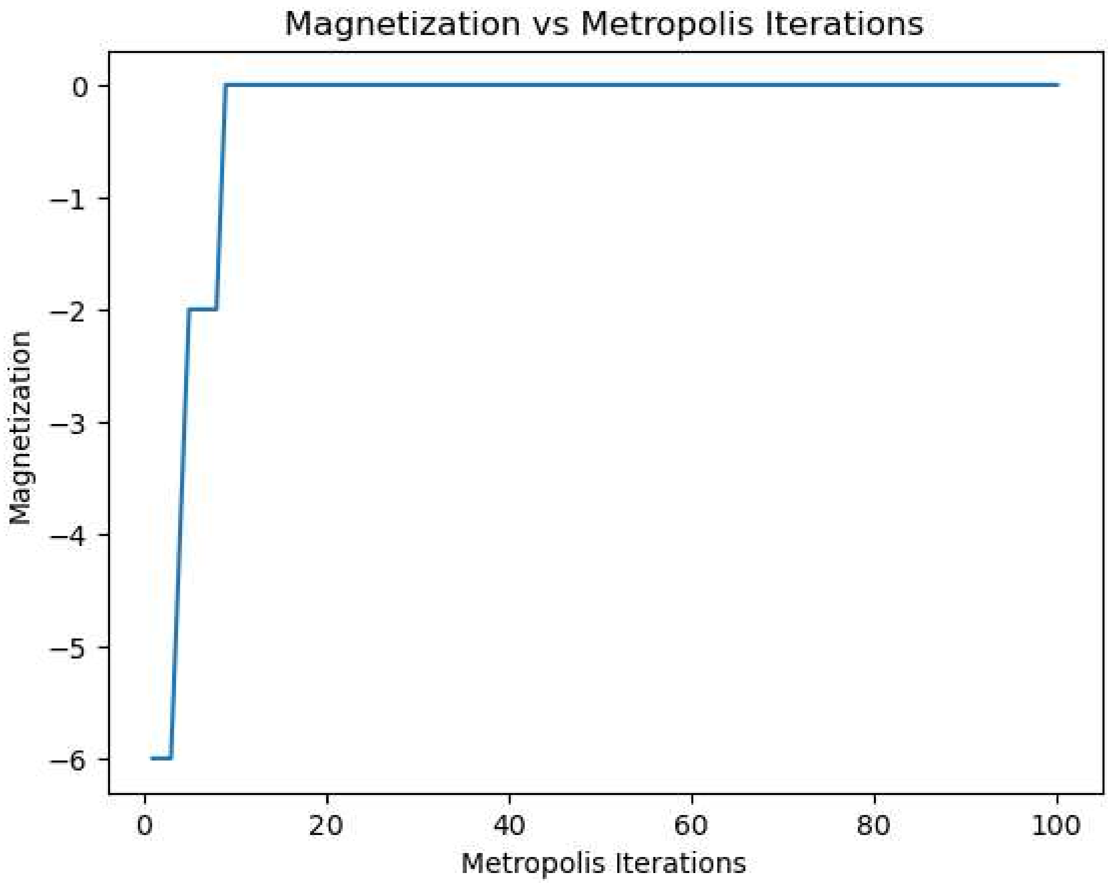
```
Red = +1
Blue = -1
```



In [45]:
```
# Plotting the energy
plt.plot(np.arange(0,itn+1),E)
plt.title("Energy Minimization")
plt.xlabel("Metropolis Iterations")
plt.ylabel("Energy")
plt.show()
```

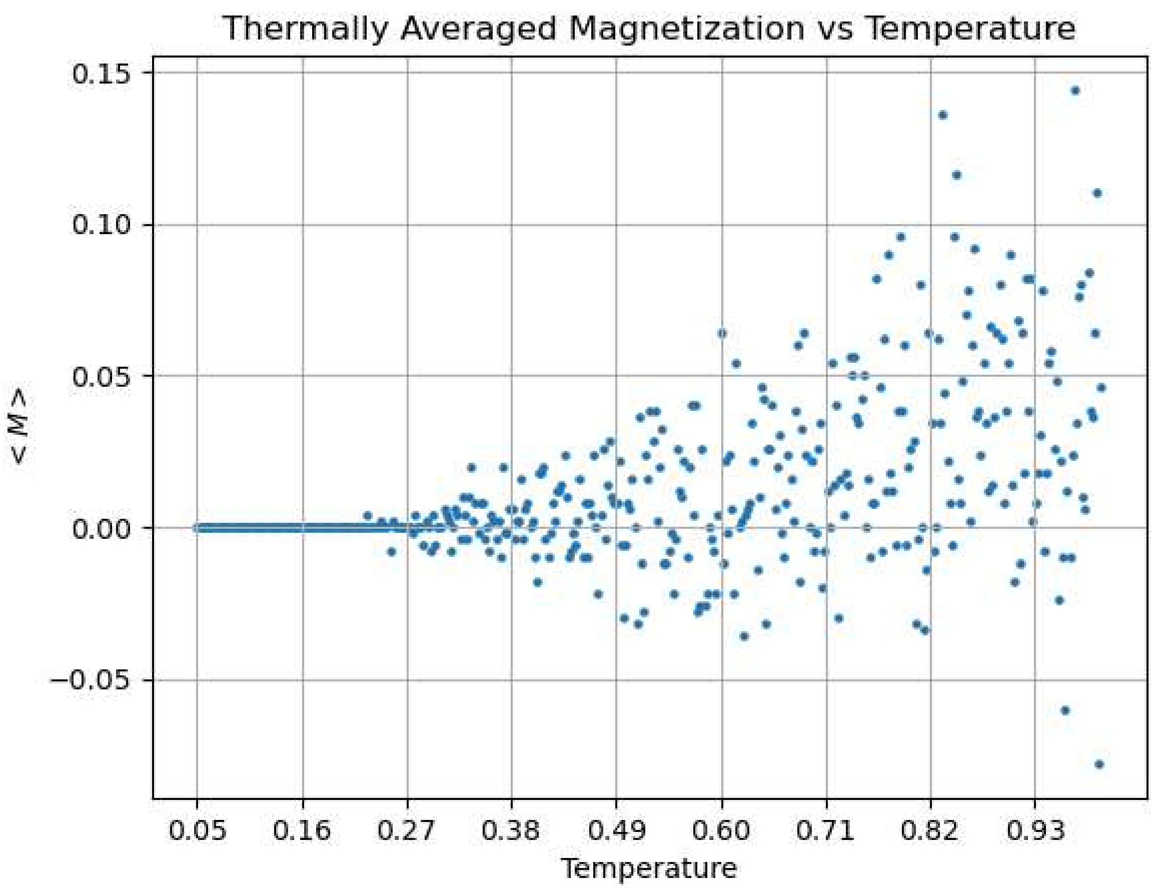


In [46]:
```
plt.plot(np.arange(1,itn+1),M)
plt.title("Magnetization vs Metropolis Iterations")
plt.xlabel("Metropolis Iterations")
plt.ylabel("Magnetization")
plt.show()
```



In [47]:
```
def magnetic_sweep(lattice,T, itn):
    '''
    Sweeping <M> values for different T
    '''
    lattice_copy = np.copy(lattice)

    M_avg = []
    for Temp in T:
        lattice = lattice_copy

        E0, cut0 = get_energy(lattice) # Getting the initial energy & cut of the la
        E = [E0]  # Energy
        M = []    # Magnetization

        for i in range(itn):
            lattice, dE = metropolis_update(lattice, Temp) # T = Temperature
            E.append(E[-1] + dE)
            M.append(sum(lattice))

        M_avg.append(np.mean(M)) # Averaging M over all iterations

    return M_avg


# lattice = 2*np.random.randint(0,2,n)-1 # Randomly generating an initial lattice

Temp = np.linspace(0.05, 1, 400) # Temperature sweep
M_avg = magnetic_sweep(lattice,Temp, 1000) # Calculating <M> as a function of T
```

In [48]:
```
plt.scatter(Temp,M_avg, s = 5)
plt.title("Thermally Averaged Magnetization vs Temperature")
plt.xlabel('Temperature')
plt.ylabel('$<M>$')

plt.xticks(np.arange(min(Temp),max(Temp), 0.11))

plt.grid()
plt.show()
```



In [49]:
```
(np.pi/2 -1)*4
```

Out[49]: 2.2831853071795862

In [50]:
```
0.34 / 0.12
```

Out[50]: 2.8333333333333335

In [ ]: