Project 2 - Simulated Annealing of the Traveling Salesman Problem

Austin Phillips
ECE 4850
UVU Electrical Engineering
Orem, Utah
austin.phillips@uvu.edu

Angel Rodriguez
ECE 4850
UVU Electrical Engineering
Orem, Utah
10801309@uvu.edu

***Abstract*—This lab focuses on the traveling salesman problem in which there are a list of cities a salesman wants to travel to for business and end the journey by returning home, but without exhausting unnecessary energy and traveling distance. Simulated Annealing is to be used to determine a path that completes the task and uses a reasonable amount of energy with the best case scenario being the least amount of energy/distance possible.**

***Index Terms*—Traveling Salesman, Simulated Annealing, Monte Carlo Method**

I. The Traveling Salesman

Simulated annealing is a Monte Carlo method that can be used to seek the optimum of a multi-modal function. This problem contains a discrete domain since all the coordinates are known and given. The traveling salesman problem is an optimization problem in which the minimum length path through a circuit is found. Here are N cities with positions (x_i, y_i). The problem is to find a path which visits each city exactly once and returns to the original city.

Since there are N cities, there are N! different orders in which they may be visited. A configuration is a permutation of the numbers 1, 2,..., N as the order in which the cities are visited. With 20 points given for this problem, there are $2.43290201 * 10^{18}$ different possible paths which makes it impractical to choose a random path of travel.

Let Equation 1 denote a permutation of the numbers 1, 2,..., N. For example, when N = 5, we might have Equation 2 denoting that the order of cities is first 3, then 5, etc. The objective function defined as a function of σ in Equation 3.

II. Simulated Annealing

The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Both are attributes of the material that depend on their thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy or Gibbs energy. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail; even though it usually achieves an approximate solution to the global minimum, it could be enough for many practical problems.

This notion of slow cooling implemented in the simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions allows for a more extensive search for the global optimal solution instead of converging to a local minima. In general, simulated annealing algorithms work as follows:

The temperature progressively decreases from an initial positive value to zero (or small decimal value approaching 0). At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease toward zero.

$$\sigma = (\sigma_1, \sigma_2, ..., \sigma_N) \tag{1}$$

$$\sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (3, 5, 2, 1, 4) \tag{2}$$

$$E(\sigma) = \sum_{i=1}^{N} \sqrt{(x_{\sigma i} - x_{\sigma i+1})^2 + (y_{\sigma i} - y_{\sigma i+1})^2} \tag{3}$$

$$\Delta E = E_i - E_{i-1} \tag{4}$$

III. Implementation in Python

The Simulated Annealing Traveling Salesman Algorithm is implemented as shown in Algorithm 1. A total of 3 functions are created: $get\_distance(x1, y1, x2, y2)$ (used for calculating the distance between two ordered pairs), $swap(iter)$ (given an iterator array, a random index in swapped with its previous neighboring index), and $total\_energy(x, y, iter)$ (calculates the total energy given an array of x-coordinates, y-coordinates, and an iterator to describe the path order). The algorithm is successfully implemented in Python in which the constant adjustments to the optimal path are displayed in a figure as shown in Figure 1. Every plot also displays energy and

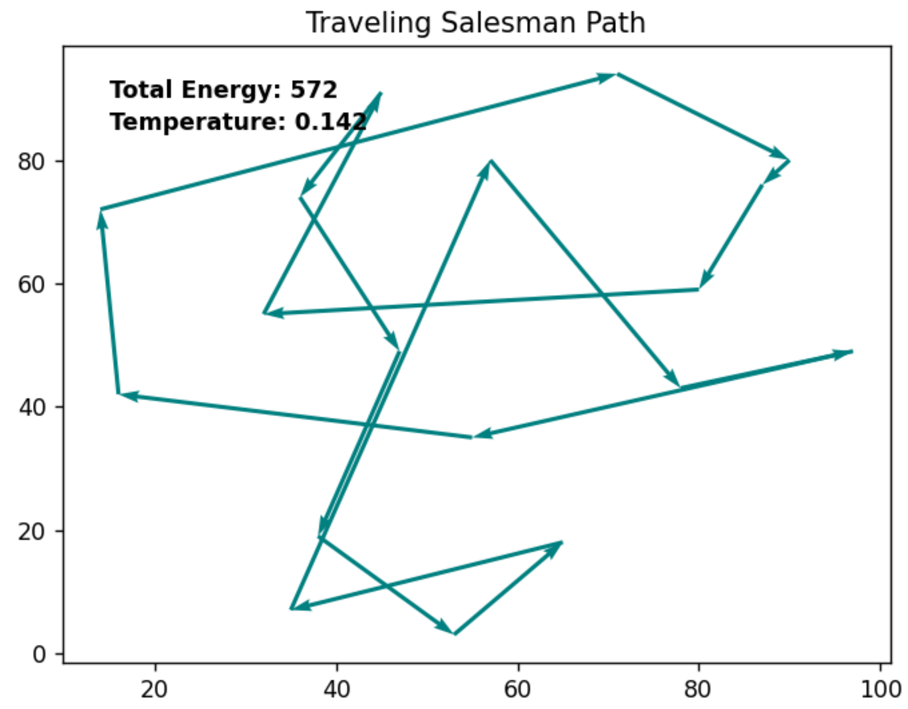

Fig. 1. An Example of a Traveling Salesman Path During Program Run Time

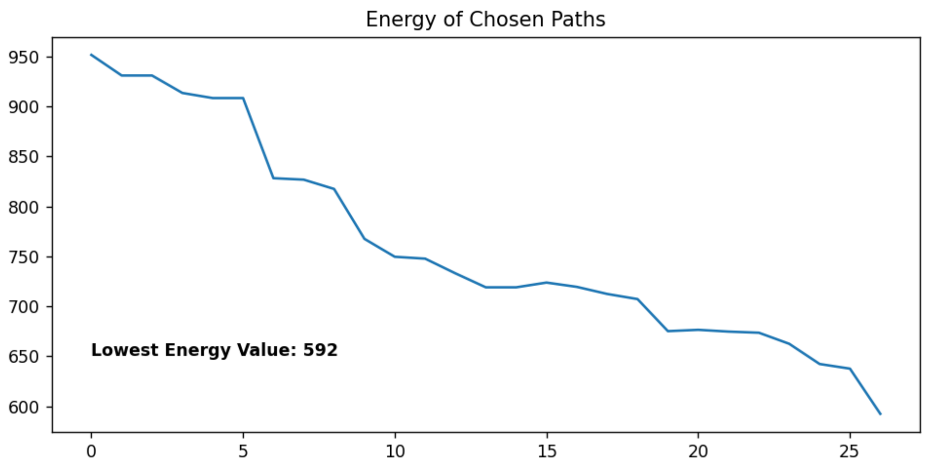

Fig. 2. Energy of Chosen Paths Over Time

temperature both of which progressively decrease over time. At the end of the program, the elements of $energy\_list$ are plotted showing the overall decrease of energy as new paths are chosen. The result signals a convergence in energy usage as observed in Figure 2. To make clear of which points are of which coordinates, code for plotting coordinate labels were added as shown in Figure 3. The next step involves plotting the path of the most optimal path which involves keeping track of the iterator associated with the least calculated energy. This part of the project had a very difficult bug to fix. It was realized that Python does not make copies of arrays with the assignment operator $=$, but rather creates a pointer to the original array. This is fine until changes are made to the original array which pointed to by the copy and so changes are made to the copy as well. Since the iterator was constantly being updated, so was the optimal iterator leading to the optimal iterator holding an order of elements that did not result in the smallest energy value within $energy\_list$. The solution was that instead of coding $optimal\_iterator = iterator$, $optimal\_iterator = copy.deepcopy(iterator)$ had to be used instead which created a true copy and not a pointer. Finally, code is added to depict three subplots including the traveling

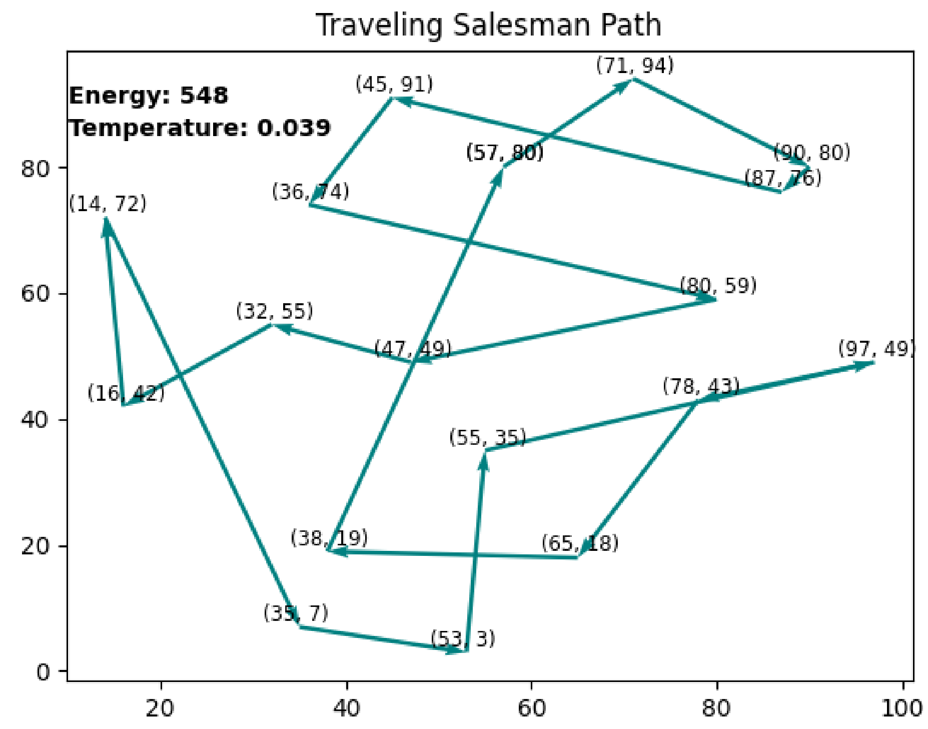

Fig. 3. Coordinates Added to Traveling Salesman Path During Program Run Time

salesman plot before simulated annealing, traveling salesman plot after simulated annealing, and the plot of chosen path energies. The results of this very informative figure can be observed in Figure 4.

IV. Conclusion

Our model and findings in this project succeed in finding much less energy intensive paths, but fall short in yielding the best results every run. Since random numbers and probability is utilized in Algorithm 1, it is expected that the same results are not achieved. The algorithm's purpose of not always accepting the better energy path is to avoid converging on a local minima where the best case scenario goal is to converge on the global minimum. The results of accepting energies over time can be observed in Figure 2 where the energy does increase at some points but overall decreases over time. During the program run time, each selected path is displayed in a figure with arrows to indicate the direction of travel as shown in Figure Figure 1. As far as results go, it seems adjusting the starting value of T and its decrement value (i.e. $T = T * 0.99$) can make the amount of total iterations of the while loop increase greatly, but more efficient paths are encountered much less frequently the longer the program runs. Most program results in an energy value within the 600s. The best results are observed in Figure 5 where the most efficient path found used 608 energy units. This shows the algorithm produces very well results but running into that scenario again is unlikely for several runs of the program. It is concluded that Simulated Annealing is an excellent implementation of solving the Traveling Salesman problem.

Algorithm 1 Simulated Annealing Traveling Salesman (SATS)

```
Let $T$ be the the starting temperature
Let $path_{min}$ be the path found with the lowest energy
Let $total\_energy(x, y, iterator)$ be an outer function which calculates the energy of arrays of x and y coordinates using Equation 3.
Let $iterator$ be $\sigma$ demonstrated in Equation 1 to represent the order of coordinates visited where $N$ is the amount of destinations.
Let $energy\_list$ be a list of recorded energies from varying paths.
Let $\Delta E = E_i - E_{i-1}$
Let $swap(iterator)$ be an outer function which takes the iterator array and generators a random index to swap its corresponding element with the previous neighboring element without moving the first index nor the last index. The newly updated iterator is returned
$iterator = np.arange(1, xpos.size - 1, 1)$
$energy\_list = [energy\_calc(xpos, ypos, iterator)]$
while $T > 0.001$ do
    $iterator_{temp}$ = $swap(iterator)$
    $E_i = energy\_calc(xpos, ypos, iterator_{temp})$
    $E_{i-1} = energy\_list[end]$
    $\Delta E = E_i - E_{i-1}$
    if $\Delta E < 0$ then
        $energy\_list = energy\_list.append(E_i)$
        $iterator = iterator_{temp}$
    else
        $u = random.uniform(0, 1)$
        if $e^{-\Delta E/T} > u$ then
            $energy\_list = energy\_list.append(E_i)$
            $iterator = iterator_{temp}$
        else
            $T = T * 0.99$
            Continue to next iteration of while loop
        end if
    end if
    $T = T * 0.99$
end while
$path_{min} = [xpos[iterator], ypos[iterator]]$
```

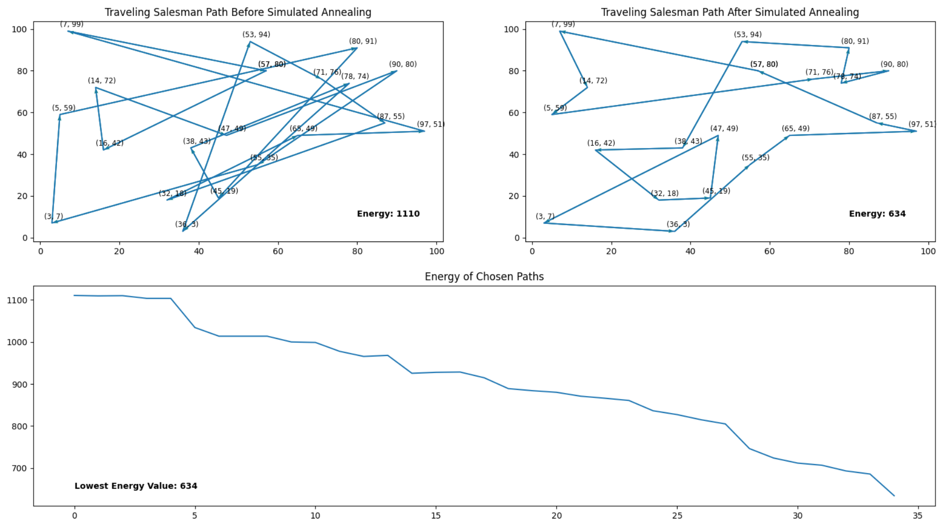

Fig. 4. Subplots Depicting Path Before Simulated Annealing, Path After Simulated Annealing, and Energy of Chosen Paths

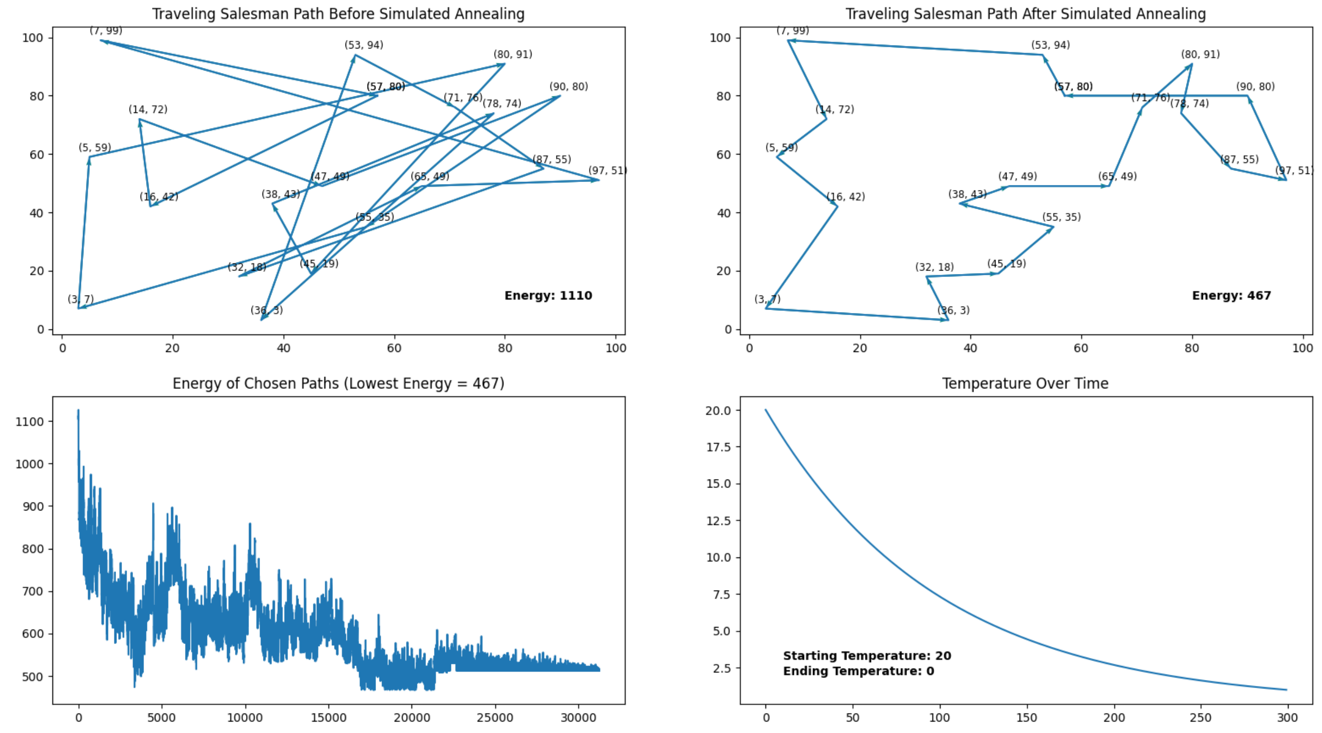

Fig. 5. The Most Efficient Path Found

V. Code

A. project2.py

```
import numpy as np
import matplotlib.pyplot as plt
import random
import time
import copy

xpos = np.array([57, 16, 14, 47, 90, 55, 3, 5, 80, 45, 38, 78, 36, 53, 71, 87, 32, 65, 97, 7])
ypos = np.array([80, 42, 72, 49, 80, 35, 7, 59, 91, 19, 43, 74, 3, 94, 76, 55, 18, 49, 51, 99]

# Define iterator, second parameter is the step size
# The first and the last coordinates in the paths will never be chosen
iterator = np.arange(1,xpos.size,1)
original_iterator = copy.deepcopy(iterator)

def get_distance(x1,y1,x2,y2):
    """x1/y1 correspond to first coordinate
    and x2/y2 correspond to second coordinate"""
    return np.sqrt((x2-x1)**2 + (y2-y1)**2)

def total_energy(x,y,iter):
    """x is the array path of x coordinates
    y is the array path of y coordinates"""
    total_energy = get_distance(x[0],y[0],x[iter[0]],y[iter[0]])
    for i in range(1, iter.size):
        energy = get_distance(x[iter[i]],y[iter[i]],x[iter[i-1]],y[iter[i-1]])
        total_energy += energy
    # Get energy from last point to starting point
    energy = get_distance(x[iter.size],y[iter.size],x[0],y[0])
    total_energy += energy
    return total_energy

def swap(iter):
    rand_int = random.randrange(0, iterator.size - 2)
    new_iter = iter
    new_iter[rand_int], new_iter[rand_int-1] = new_iter[rand_int-1], new_iter[rand_int]
    return new_iter


energy_list = np.array([])
energy_list = np.append(energy_list, total_energy(xpos,ypos,iterator))

# enable interactive mode
plt.ion()
# creating subplot and figure
fig = plt.figure()
ax = fig.add_subplot(111)
xpath = np.append(np.append(xpos[0],xpos[iterator]),xpos[0])
ypath = np.append(np.append(ypos[0],ypos[iterator]),ypos[0])
line1, = ax.plot(xpath, ypath)
plt.title("Traveling Salesman Path")
plt.text(15, 90, f’Total Energy: {int(energy_list[0])}’, fontsize=’medium’, weight="bold")
ax.quiver(xpath[:-1], ypath[:-1], xpath[1:]-xpath[:-1],
                   ypath[1:]-ypath[:-1],scale_units=’xy’, angles=’xy’, scale=1, color=’teal’,
ax.clear()

# We will use this to graph the minimum path later
best_iterator = copy.deepcopy(iterator)
loop_counter = 1
T = 2
T_stop = 0.000001
T_decimation = 0.99999
while T > T_stop:
    loop_counter += 1
    iterator = swap(iterator)
    energy = total_energy(xpos,ypos, iterator)

    # delta E = E_i - E_i-1 where E_i is current energy and E_i-1  is previous energy
    # if delta E < 0, accept the current path
    if energy < energy_list[-1]:
        energy_list = np.append(energy_list, energy)
        best_iterator = copy.deepcopy(iterator)

    else: # else, accept current path if e^(-delta_E/T) > u
        u = random.uniform(0, 1)
        delta_E = energy - energy_list[-1]
        if np.exp(-delta_E/T) > u: # accept path
            energy_list = np.append(energy_list, energy)
        else: # This is when we do not want to update the iterator/path
            # Always update T
            T *= T_decimation
            continue
    # Always update T
    T *= T_decimation
    print(f"Accepted energy: {energy_list[-1]}")

    # Plot
    # updating the values of x and y
    xpath = np.append(np.append(xpos[0],xpos[iterator]),xpos[0])
    ypath = np.append(np.append(ypos[0],ypos[iterator]),ypos[0])
    line1.set_xdata(xpath)
    line1.set_ydata(ypath)
    ax.quiver(xpath[:-1], ypath[:-1], xpath[1:]-xpath[:-1],
                   ypath[1:]-ypath[:-1],scale_units='xy', angles='xy', scale=1, color='teal', 
    # re-drawing the figure
    plt.title("Traveling Salesman Path")
    for i, j in zip(xpath, ypath): # Writes the (x,y) coordinates above the coordinate locatio
            plt.text(i-4, j+1, '({}, {})'.format(i, j), fontsize='small')
    plt.text(60, 15, f'Energy: {int(energy)}', fontsize='medium', weight="bold")
    format_T = "{:.3f}".format(T)
    plt.text(60, 10, f'Temperature: {format_T}', fontsize='medium', weight="bold")
    fig.canvas.draw()
    # to flush the GUI events
    fig.canvas.flush_events()
    ax.clear()
    time.sleep(0.1)

# print(f"original_iterator size: {original_iterator.size}\n{original_iterator}")
# print(f"best_iterator size: {best_iterator.size}\n{best_iterator}")
lowest_energy = total_energy(xpos,ypos,best_iterator)
```

```
# print(f"lowest_energy: {lowest_energy}")
# print(f"min value in energy_list: {np.amin(energy_list)}")

# Turn off the fast updating plot
plt.ioff()
ax.remove()

# Plot the energy decrease over time
plt.subplot(2, 1, 2)
plt.plot(energy_list)
plt.title("Energy of Chosen Paths")
plt.text(0, 650, f'Lowest Energy Value: {int(np.amin(energy_list))}', fontsize='medium', weight

# Plot the unoptimized path
plt.subplot(2, 2, 1)
xpath = np.append(xpos,xpos[0])
ypath = np.append(ypos,ypos[0])
plt.plot(xpath,ypath)
plt.quiver(xpath[:-1], ypath[:-1], xpath[1:]-xpath[:-1],
                   ypath[1:]-ypath[:-1],scale_units='xy', angles='xy', scale=1, color='teal',
plt.title("Traveling Salesman Path Before Simulated Annealing")
for i, j in zip(xpath, ypath): # Writes the (x,y) coordinates above the coordinate location
            plt.text(i-2, j+2, '({}, {})'.format(i, j), fontsize='small')
plt.text(80, 10, f'Energy: {int(energy_list[0])}', fontsize='medium', weight="bold")

# Plot the ending path
plt.subplot(2, 2, 2)
xpath = np.append(np.append(xpos[0],xpos[best_iterator]),xpos[0])
ypath = np.append(np.append(ypos[0],ypos[best_iterator]),ypos[0])
plt.plot(xpath,ypath)
plt.quiver(xpath[:-1], ypath[:-1], xpath[1:]-xpath[:-1],
                   ypath[1:]-ypath[:-1],scale_units='xy', angles='xy', scale=1, color='teal',
plt.title("Traveling Salesman Path After Simulated Annealing")
for i, j in zip(xpath, ypath): # Writes the (x,y) coordinates above the coordinate location
            plt.text(i-2, j+2, '({}, {})'.format(i, j), fontsize='small')
plt.text(80, 10, f'Energy: {int(lowest_energy)}', fontsize='medium', weight="bold")
# plt.subplots_adjust(left=0.125, bottom=0.044, right=0.589, top=0.943, wspace=0.2, hspace=0.2
manager = plt.get_current_fig_manager()
manager.full_screen_toggle() # Make full screen for better view, Alt-F4 to exit full screen
plt.show()
```