

# ECE457A: Assignment 2 Solution

Jacob Bodera

2024-10-11

# Question 1 Solution

## 1. Problem Formulation

**State Representation:** For this simulated annealing problem, the representation of a state is simply the current coordinate pair of  $x_1$  and  $x_2$  values. And thus, an arbitrary state can be represented as  $S = (x_1, x_2)$

**Initial State:** The initial state of this problem was chosen to be a random coordinate,  $(x_1, x_2) \forall \in [-100, 100]$ .

**Goal State:** The goal state of this problem is to find the coordinate which minimizes the *Easom* function (within some degree of error). This goal state is  $S = (\pi, \pi)$ .

**Actions::** The actions available for the simulated annealing algorithm are as follows:

- change current state values of  $x_1$  and  $x_2$  using the neighbourhood function
- accept the new state if the new cost is less than the current cost
- accept or reject the new state if the new cost is less than or equal to the current cost based on the probability function  $P = e^{\frac{-\Delta c}{t}}$

**Cost:** Due to the structure of this problem, I decided that the best cost function was to use a variation of the *Easom* function itself. Since the function is essentially constant for values surrounding the minimum at  $(\pi, \pi)$ , it is important that the cost function is extremely sensitive to small changes so that it is much more likely to detect the steep function decrease. Therefore, the cost function chosen is the following:

$$c(x_1, x_2) = -\cos(x_1)\cos(x_2)((x_1 - \pi)^2 + (x_2 - \pi)^2) \cdot 10^6$$

**Neighbourhood Function:** The neighbourhood function in determines how the current state may change from one iteration to another in simulated annealing. For the neighbourhood function for this problem, it was chosen to be an addition or subtraction of a random uniform distribution within the bounds  $(-10, 10)$  from the current state. It was also important to saturate the new state values to the bounds given in the problem  $[-100, 100]$ .

$$x_{1_{i+1}} = \min(\max(x_{1_i} + \text{rand.unif}(-10, 10), -100), 100)$$

$$x_{2_{i+1}} = \min(\max(x_{2_i} + \text{rand.unif}(-10, 10), -100), 100)$$

## 2. Solution Description

**Pseudo Code from Simulated Annealing:**

```
define simulated_annealing(temp_0, alpha, bounds, max_iterations):
    curr_x = random (x1, x2) values in bounds
    curr_cost = cost_function(curr_x)

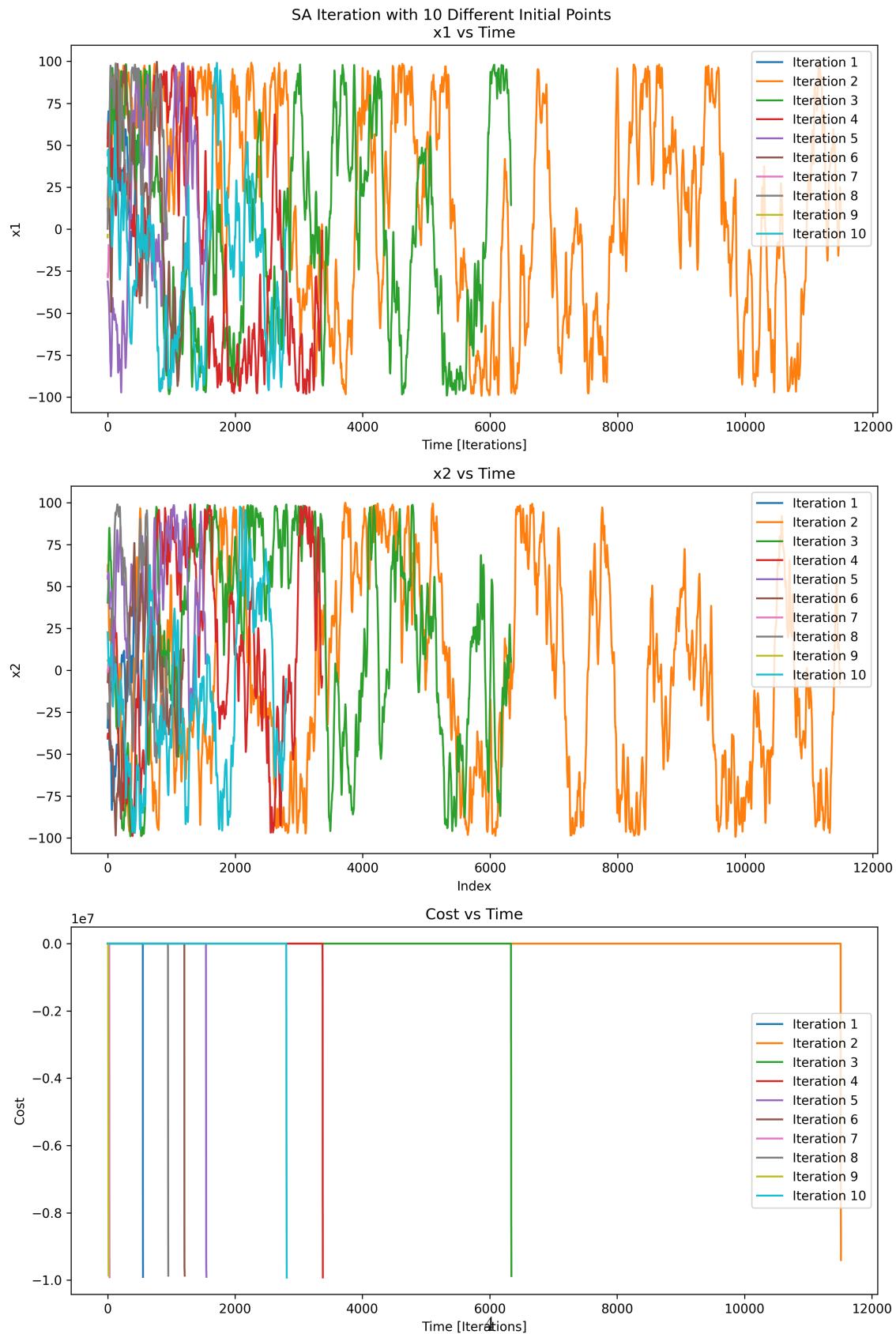
    loop in max_iterations:
        new_x = neighbour_function(curr_x)
        new_cost = cost_function(new_x)
        delta_cost = new_cost - curr_cost

        if delta_cost < 0:
            curr_x = new_x
            curr_cost = new_cost
        else:
            if random < probability_function(delta_cost, temp)
```

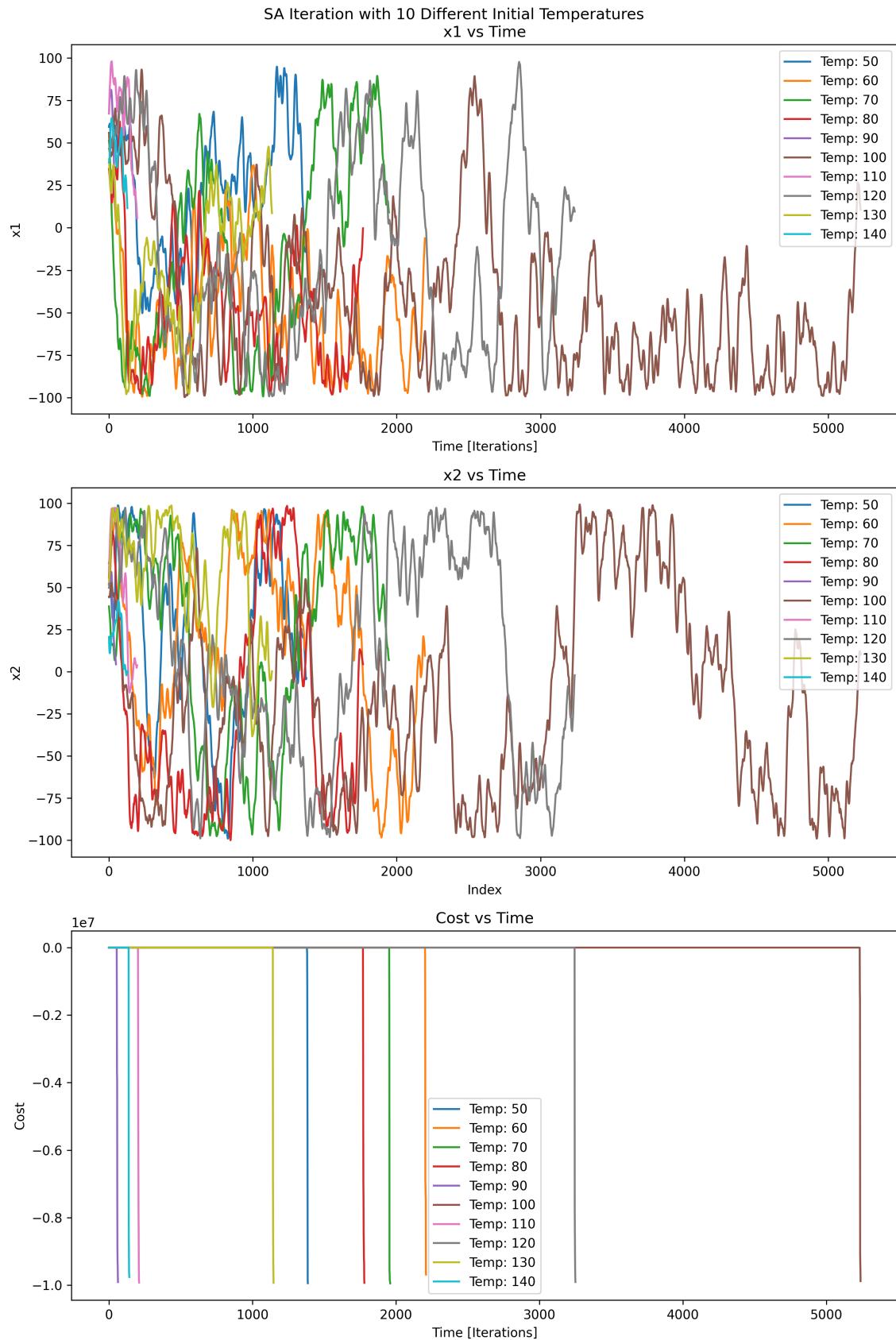
```
curr_x = new_x
curr_cost = new_cost
temp = cool_function(temp, alpha)
return curr_x, curr_cost
```

### 3. Solution Profile for Different Variates

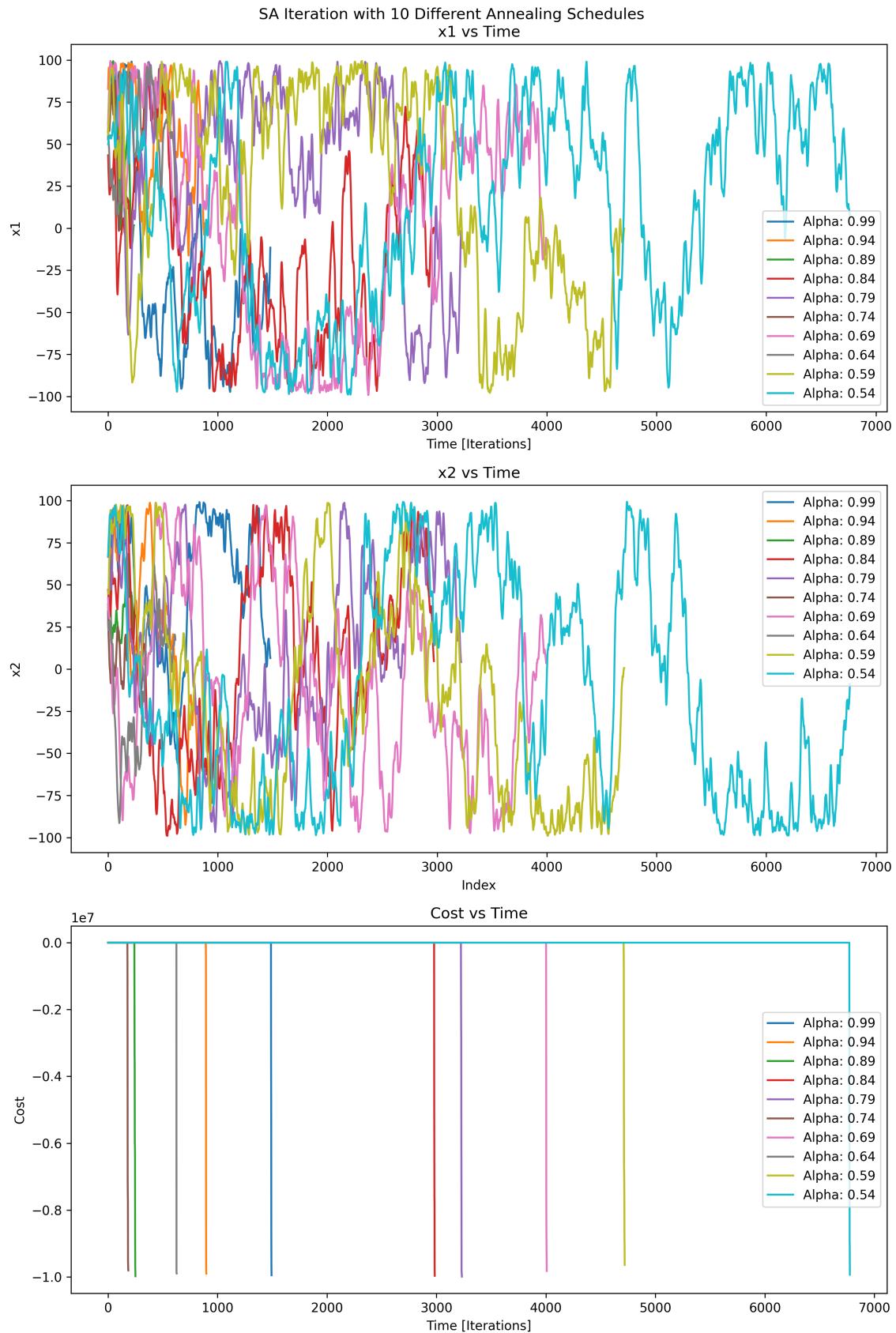
#### Experiment 1: 10 Different Initial Points



## Experiment 2: 10 Different Initial Temperatures



### Experiment 3: 10 Different Annealing Schedules



## 4. Observations

From experiment one, it can be seen that the algorithm is generally very good at quickly finding a solution to this optimization problem, regardless of the initial guess. This can be seen in the plot as 8/10 of the executions took less than 3500 iterations. The two outliers in the plot show that there is some range of initial guess that requires more iterations of the algorithm to reach the goal.

From experiment two, it appears that temperature values around 100 perform the best. These tests were performed with a constant alpha value, and thus it appears that temperature around 100 worked the best for this cooling.

From experiment three, it can be seen that the lower values of alpha do not perform as well in finding a solution and take more iterations. With an alpha value too small, this decreases the temperature to rapidly until it approaches zero and causes the probability function to be undefined.

## 5. Performance of SA

Through the experimentation shown above, the best solution that was output from my program was a minimum value of approximately  $x_1 = 3.142557$  and  $x_2 = 3.138406$ , which a cost of  $c(x_1, x_2) = -9999833.714$ . For this iteration, the initial points were chosen randomly, with the remaining settings of the `sim_annealing()` function as follows:

- $t_0 = 100$
- $\alpha = 0.99$

The output of the program with these settings during this iteration can be seen in the figure below.

```
~/Desktop/School/ECE457A/assignments/assignment2/code git h main !7 ?2
python3 problem1_a2.py
Minimum Cost: -9999833.714428004
Point: (3.1425572336414196, 3.138405903440075)
```

These settings results in good solution due to a good cooling function and initial temperature. The geometric cooling function used for this algorithm results in the values being more likely to be accepted near the beginning of the execution and not accepted later in the execution. This allows the simulated annealing algorithm to be able so efficiently search the state space until it detects the minimum.

The time complexity of this program and can be described as the following:

- `easom()` performs one simple calculation and thus the space and time complexity are both  $O(1)$
- `cost()` performs one calculation with  $x$  and therefore, has time and space complexity of  $O(1)$
- `cool()` simply updates the temperature value and so time and space complexity are both  $O(1)$
- `sim_annealing()` this algorithm contains one `for` loop which is dependent on the number of iterations and with each iteration, the cost and coordinates are stored in lists. Therefore, the time complexity is  $O(i)$  and the space complexity is  $O(2i) \Rightarrow O(i)$ , where  $i$  is the number of iterations.

## 6. Code

```
from math import cos, exp, pi
from random import uniform, random
import matplotlib.pyplot as plt
import numpy as np

def easom(x1, x2):
    return -cos(x1)*cos(x2)*exp(-(x1 - pi)**2 - (x2 - pi)**2)

def cost(x1, x2, t):
    return easom(x1, x2) * 1E7

def cool(temp, alpha):
    return temp ** alpha

def sim_annealing(t_0, alpha, bounds, max_iterations, x = None):
    # Initial random guess within bounds
    if x is None:
        x1 = (random() - 0.5) * bounds[0][1]
        x2 = (random() - 0.5) * bounds[1][1]
    else:
        x1, x2 = x

    current_x = list()
    current_x.append((x1, x2))
    current_cost = list()
    current_cost.append(cost(x1, x2, t = "easom"))

    temp = t_0
    for i in range(max_iterations):
        # new x's in neighbourhood and ensure in bounds
        neighbourhood_factor = 10
        new_x1 = min(max(current_x[-1][0] + uniform(-1, 1) * neighbourhood_factor, bounds[0][0]), bounds[0][1])
        new_x2 = min(max(current_x[-1][1] + uniform(-1, 1) * neighbourhood_factor, bounds[1][0]), bounds[1][1])

        new_x = (new_x1, new_x2)
        new_cost = cost(new_x1, new_x2, t = "easom")

        delta_cost = new_cost - current_cost[-1]

        if temp < 1E-5:
            return current_x, current_cost

        # accept new x
        if delta_cost < 0:
            current_x.append(new_x)
            current_cost.append(new_cost)
        else:
            if random() < exp(-delta_cost/temp):
                current_x.append(new_x)
                current_cost.append(new_cost)
```

```

        temp = cool(temp, alpha)

    return current_x, current_cost

x, c = sim_annealing(t_0 = 100, alpha = 0.99, bounds = [(-100, 100), (-100, 100)], max_iterations = 2000)

print(f"Minimum Cost: {c[-1]}\nPoint: {x[-1]}")

# Moving average function for smoothing
def moving_average(data, window_size=10):
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

'''

## Experiments
exp1_x = list()
exp1_cost = list()
for i in range(10):
    x, c = sim_annealing(t_0 = 100, alpha = 0.99 - (0.05*i), bounds = [(-100, 100), (-100, 100)], max_i
    exp1_x.append(x)
    exp1_cost.append(c)

window_size = 15
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 15))

# Plot x1 vs index
for i, x_values in enumerate(exp1_x):
    x1_values = [x[0] for x in x_values]
    smoothed_x1 = moving_average(x1_values, window_size)
    ax1.plot(range(len(smoothed_x1)), smoothed_x1, label=f'Alpha: {round(0.99 - (0.05*i), 2)}')
ax1.set_title('x1 vs Time')
ax1.set_xlabel('Time [Iterations]')
ax1.set_ylabel('x1')
ax1.legend()

# Plot x2 vs index
for i, x_values in enumerate(exp1_x):
    x2_values = [x[1] for x in x_values]
    smoothed_x2 = moving_average(x2_values, window_size)
    ax2.plot(range(len(smoothed_x2)), smoothed_x2, label=f'Alpha: {round(0.99 - (0.05*i), 2)}')
ax2.set_title('x2 vs Time')
ax2.set_xlabel('Index')
ax2.set_ylabel('x2')
ax2.legend()

# Plot cost vs index
for i, cost_values in enumerate(exp1_cost):
    ax3.plot(range(len(cost_values)), cost_values, label=f'Alpha: {round(0.99 - (0.05*i), 2)}')
ax3.set_title('Cost vs Time')
ax3.set_xlabel('Time [Iterations]')
ax3.set_ylabel('Cost')
ax3.legend()

# Display the plots
fig.suptitle("SA Iteration with 10 Different Annealing Schedules")

```

```
plt.tight_layout()  
plt.savefig('experiment3_plot.png', dpi=300)  
plt.show()  
'''
```

## Question 2 Solution

### 1. Problem Description and Strategy

#### Problem Formulation

**State Representation:** For the Conga game, the state representation is the board describing how many stones are in each tile on the board and to which player the stones belong. This is represented in the program by two 2-dimensional 4x4 matrices where the player matrix contains either "B" for black stones, "W" for white pieces, and None for empty squares. For the stones matrix, each index contains the number of square on the corresponding tile, regardless of the colour.

**Initial State:** The initial state is the board configuration at the beginning of the game. This means that the player board has a "B" in the position (1, 4) and a "W" in the (4, 1) position. For the stones board, there are 10 stones in both the (1, 4) and (4, 1) positions.

$$\begin{bmatrix} B & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & W \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

**Goal State:** The goal state for this problem is any configuration in which a player does not have any legal moves. One such configuration is as follows:

$$\begin{bmatrix} B & W & - & - \\ W & W & - & - \\ - & - & - & - \\ - & - & - & - \end{bmatrix} \begin{bmatrix} 10 & 3 & 0 & 0 \\ 6 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Actions::** An action is defined as performing any legal move in Conga. This is defined in the assignment documentation, however, it involves removing all stones from a tile and moving in one of the 8 possible directions. Travel as far as possible in that direction and stop until you reach the edge of the board or a tile with enemy stones on it. Place 1 stone in each tile along that path, with all remaining stones being added to the last tile. Player turns alternate every action.

**Cost:** The cost defined for this problem is the number of legal moves available to white subtracted by the number of legal moves available for black. The means that positive cost values favour white and negative cost values favour black.

$$\text{Cost} = (\# \text{ legal white moves}) - (\# \text{ legal black moves})$$

### 2. Observations of Algorithm vs. Random Agent

### 3. Program Output

### 4. Discussion on Time and Memory Complexity

## Question 3 Solution

### 1. Optimization Formulation

### 2. Performance Evaluation

#### Experiment 1: Different Initial Solutions

Iteration 0:

Initial config: [3, 4, 6, 9, 5, 7, 14, 0, 2, 16, 13, 15, 10, 18, 19, 12, 17, 1, 11, 8]  
Best configuration found: [17, 1, 13, 2, 8, 18, 14, 11, 9, 15, 3, 19, 7, 10, 12, 16, 6, 4, 0, 5]  
Cost: 2574.0

Iteration 1:

Initial config: [11, 3, 16, 8, 7, 6, 2, 9, 1, 12, 13, 4, 18, 0, 5, 10, 19, 17, 14, 15]  
Best configuration found: [16, 4, 6, 0, 5, 18, 14, 19, 7, 12, 3, 1, 11, 10, 15, 17, 13, 9, 2, 8]  
Cost: 2570.0

Iteration 2:

Initial config: [19, 5, 14, 0, 11, 13, 9, 8, 7, 10, 12, 15, 16, 3, 6, 1, 18, 17, 2, 4]  
Best configuration found: [8, 2, 9, 13, 17, 15, 10, 11, 1, 3, 12, 7, 19, 14, 18, 5, 0, 6, 4, 16]  
Cost: 2570.0

Iteration 3:

Initial config: [9, 2, 14, 19, 0, 10, 17, 6, 5, 4, 15, 11, 18, 1, 8, 16, 13, 7, 12, 3]  
Best configuration found: [5, 0, 4, 6, 16, 12, 10, 7, 19, 3, 15, 9, 11, 14, 18, 8, 2, 13, 1, 17]  
Cost: 2574.0

Iteration 4:

Initial config: [0, 18, 5, 3, 6, 2, 19, 14, 13, 11, 8, 9, 15, 10, 17, 4, 7, 12, 16, 1]  
Best configuration found: [16, 4, 6, 0, 5, 18, 14, 19, 7, 12, 3, 1, 11, 10, 15, 17, 13, 9, 2, 8]  
Cost: 2570.0

Iteration 5:

Initial config: [8, 13, 14, 2, 9, 0, 18, 15, 17, 7, 5, 3, 10, 4, 6, 11, 19, 12, 1, 16]  
Best configuration found: [8, 2, 9, 13, 17, 15, 10, 11, 1, 3, 12, 7, 19, 14, 18, 5, 0, 6, 4, 16]  
Cost: 2570.0

Iteration 6:

Initial config: [14, 4, 7, 11, 17, 0, 5, 19, 1, 16, 12, 6, 9, 18, 8, 15, 3, 2, 13, 10]  
Best configuration found: [16, 6, 4, 0, 5, 3, 19, 7, 10, 12, 18, 14, 11, 9, 15, 17, 1, 13, 2, 8]  
Cost: 2574.0

Iteration 7:

Initial config: [0, 7, 1, 6, 14, 5, 10, 12, 19, 17, 11, 9, 8, 4, 2, 18, 13, 16, 15, 3]  
Best configuration found: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17]  
Cost: 2570.0

Iteration 8:

Initial config: [6, 9, 14, 1, 19, 2, 18, 17, 0, 5, 15, 7, 8, 16, 3, 4, 11, 10, 13, 12]  
Best configuration found: [17, 13, 9, 2, 8, 3, 1, 11, 10, 15, 18, 14, 19, 7, 12, 16, 4, 6, 0, 5]  
Cost: 2570.0

Iteration 9:

Initial config: [1, 3, 9, 15, 13, 11, 6, 19, 18, 14, 10, 12, 4, 5, 0, 8, 16, 17, 7, 2]

```

Best configuration found: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17]
Cost: 2570.0

Iteration 10:
Initial config: [12, 13, 8, 19, 0, 18, 5, 6, 4, 2, 11, 14, 16, 3, 7, 1, 17, 10, 15, 9]
Best configuration found: [17, 13, 9, 2, 8, 3, 1, 11, 10, 15, 18, 14, 19, 7, 12, 16, 4, 6, 0, 5]
Cost: 2570.0

Iteration 11:
Initial config: [18, 7, 9, 16, 3, 10, 8, 5, 2, 0, 19, 15, 6, 13, 4, 1, 17, 14, 11, 12]
Best configuration found: [2, 17, 9, 0, 5, 13, 1, 11, 6, 4, 8, 14, 7, 19, 12, 15, 18, 10, 3, 16]
Cost: 2588.0

Iteration 12:
Initial config: [10, 11, 6, 12, 2, 5, 4, 14, 7, 13, 19, 8, 16, 0, 9, 15, 3, 17, 18, 1]
Best configuration found: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17]
Cost: 2570.0

Iteration 13:
Initial config: [10, 19, 3, 0, 6, 1, 2, 14, 18, 9, 15, 13, 11, 17, 5, 7, 8, 16, 12, 4]
Best configuration found: [16, 4, 6, 0, 5, 18, 14, 19, 7, 12, 3, 1, 11, 10, 15, 17, 13, 9, 2, 8]
Cost: 2570.0

Iteration 14:
Initial config: [8, 10, 18, 7, 11, 15, 1, 0, 12, 19, 4, 6, 5, 17, 2, 14, 9, 13, 16, 3]
Best configuration found: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17]
Cost: 2570.0

Iteration 15:
Initial config: [11, 6, 15, 5, 13, 3, 0, 7, 10, 18, 16, 17, 1, 14, 4, 12, 19, 2, 9, 8]
Best configuration found: [5, 0, 9, 17, 2, 4, 6, 11, 1, 13, 12, 19, 7, 14, 8, 16, 3, 10, 18, 15]
Cost: 2588.0

Iteration 16:
Initial config: [8, 17, 19, 10, 12, 2, 4, 14, 3, 5, 7, 6, 13, 16, 18, 9, 15, 11, 1, 0]
Best configuration found: [5, 0, 6, 19, 16, 12, 4, 10, 7, 3, 15, 9, 11, 14, 18, 8, 2, 13, 1, 17]
Cost: 2580.0

Iteration 17:
Initial config: [11, 6, 10, 14, 13, 12, 19, 3, 15, 2, 4, 1, 9, 8, 7, 16, 5, 18, 0, 17]
Best configuration found: [8, 2, 9, 13, 17, 15, 10, 11, 1, 3, 12, 7, 19, 14, 18, 5, 0, 6, 4, 16]
Cost: 2570.0

Iteration 18:
Initial config: [10, 2, 15, 8, 13, 18, 5, 7, 17, 6, 1, 4, 9, 0, 14, 11, 19, 12, 3, 16]
Best configuration found: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17]
Cost: 2570.0

Iteration 19:
Initial config: [5, 8, 18, 16, 13, 2, 10, 14, 7, 12, 3, 1, 0, 4, 15, 6, 19, 11, 9, 17]
Best configuration found: [16, 6, 4, 0, 5, 3, 19, 7, 10, 12, 18, 14, 11, 9, 15, 17, 1, 13, 2, 8]
Cost: 2574.0

```

**Experiment 2: Different Tabu Sizes**

**Experiment 3: Dynamic Tabu Size**

**Experiment 4: Adding Aspiration Criteria**

**Experiment 5: Frequency-Based Tabu List**