Spring 2023: CSCI 4588/5588 Programming Assignment #1

Submitted By:
Name: Padam Jung Thapa
ID: 2623560

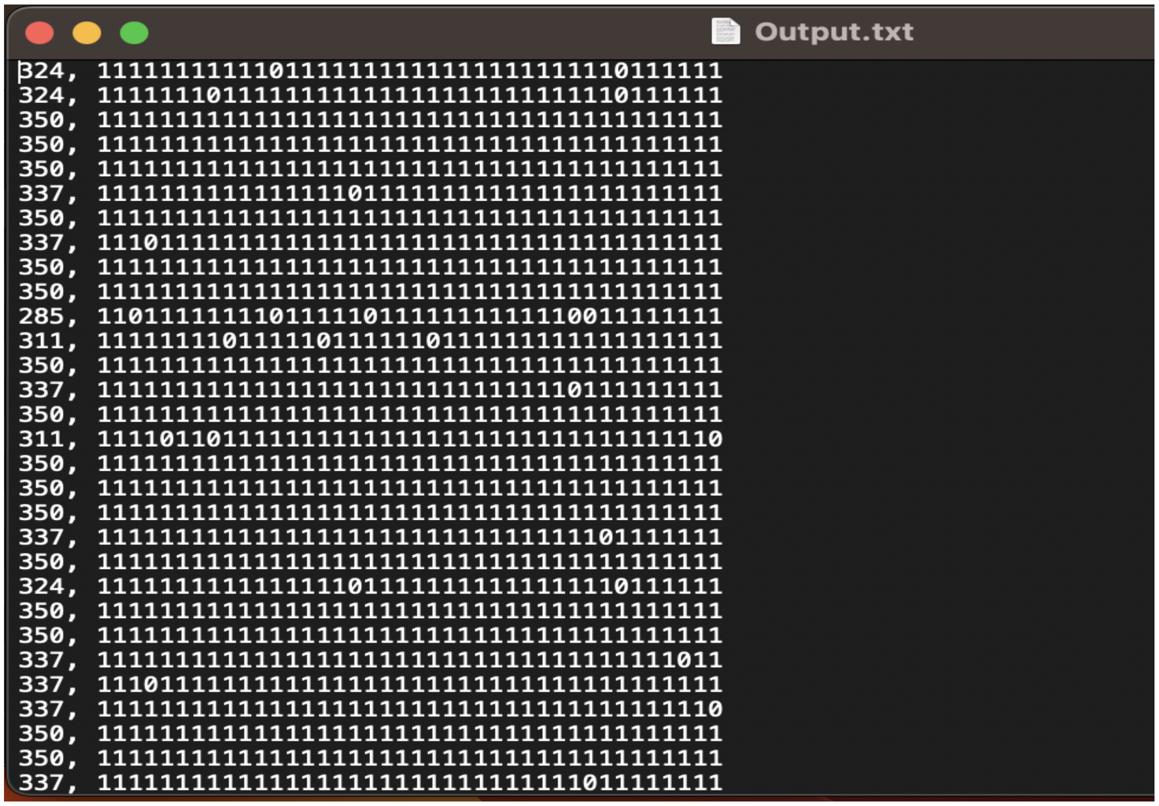
1. Write a Hill-Climbing algorithm to find the maximum value of a function f, where f = |13 • one (v) -170|. Here, v is the input binary variable of 40 bits. The one counts the number of ‘1’s in v. Set MAX =100, thus reset the algorithm 100 times for the global maximum and print the found maximum-value for each reset separated by a comma in the Output.txt file.

Answer:

```
# Import required modules
import random
import time
import tracemalloc

# Define function to generate a random neighbor of a binary string
def random_neighbor(v):
  # Convert binary string to a list of characters
  v_list = list(v)

  # Choose a random index in the binary string
  i = random.randint(0, len(v) - 1)

  # Flip the bit at the chosen index
  v_list[i] = '0' if v[i] == '1' else '1'

  # Convert the list of characters back to a binary string
  return ''.join(v_list)

# Define Hill Climbing algorithm to find the maximum value of the function f = |13 * one(v) - 170|
def hill_climbing(v, max_iterations):
  # Set current binary string and its value as the input binary string and its value
  current_v = v
  current_value = abs(13 * v.count('1') - 170)

  # Set best binary string and its value to be the current binary string and its value
  best_value = current_value
  best_v = current_v
  # Loop for a maximum of `max_iterations` times
  for t in range(max_iterations):
    # Set flag to indicate whether local optimum has been found
    local_optimum = True

    # Generate 30 random neighbors and check if any of them have a better value than the
current binary string
    for i in range(30):
      neighbor = random_neighbor(current_v)
      neighbor_value = abs(13 * neighbor.count('1') - 170)
      if neighbor_value > current_value:
        current_v = neighbor
        current_value = neighbor_value
        local_optimum = False
        if neighbor_value > best_value:
          best_value = neighbor_value
          best_v = neighbor

    # If no better neighbor is found, generate a new random binary string as the current binary
string
    if local_optimum:
      current_v = ''.join([str(random.randint(0, 1)) for i in range(40)])

  # Return the best binary string and its value found after executing the Hill Climbing algorithm
  return best_value, best_v

# Start measuring the time complexity
start_time = time.time()

# Start tracking memory usage
tracemalloc.start()

# Execute main code only when the module is run as a standalone program
if __name__ == '__main__':
  # Set the number of times to run the Hill Climbing algorithm
  MAX = 100
  # Generate a random binary string of length 40
  v = ''.join([str(random.randint(0, 1)) for i in range(40)])
  # List to store the results of each run of the Hill Climbing algorithm
  results = []
  # Loop to run the Hill Climbing algorithm `MAX` times
  for i in range(MAX):
    result = hill_climbing(v, 100) # run the Hill Climbing algorithm and store the result
    results.append(result)         # append the result to the results list
```

```
     v = ''.join([str(random.randint(0, 1)) for i in range(40)]) # generate a new random binary string of length 40
  with open('Output.txt', 'w') as f:  # Write the results to a text file 'Output.txt'
    for value, v in results:
      f.write(f'{value}, {v}\n') # write the value and binary string in the format "value, binary_string"

# Calculate and print the time complexity
end_time = time.time() #end time
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop() #stop tracking memory usage

print("Time complexity: ", end_time - start_time, "seconds") #print the time complexity
print("Space complexity: ", peak, "bytes", "or", peak/1024, "KB") #print the space complexity
```

This code implements a Hill-climbing algorithm to find the maximum value of a function f, where f = |13 * one(v) - 170|, becoming the global maxima of |13*40-170| = 350. The input variable ‘v’ is a binary variable of 40 bits, and the one function counts the number of 1's in v. The algorithm resets 100 times and finds the maximum value for each reset, printing the found maximum values to an "Output.txt" file, separated by commas.



```
324, 1111111111101111111111111111111110111111
324, 1111111011111111111111111111111110111111
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
337, 1111111111111111011111111111111111111111
350, 1111111111111111111111111111111111111111
337, 1110111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
285, 1101111111101111101111111111110011111111
311, 1111111101111101111110111111111111111111
350, 1111111111111111111111111111111111111111
337, 1111111111111111111111111111110111111111
350, 1111111111111111111111111111111111111111
311, 1111011011111111111111111111111111111110
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
337, 1111111111111111111111111111111101111111
350, 1111111111111111111111111111111111111111
324, 1111111111111111011111111111111110111111
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
337, 1111111111111111111111111111111111111011
337, 1110111111111111111111111111111111111111
337, 1111111111111111111111111111111111111110
350, 1111111111111111111111111111111111111111
350, 1111111111111111111111111111111111111111
337, 1111111111111111111111111111111011111111
```

Fig: Snapshot of the Output.txt file of Hill-Climbing Algorithm

2. Write a Simulated-Annealing algorithm to find the maximum value of a function f, where f = |14 • one (v) -190|. Here, v is the input binary variable of 50 bits. The one counts the number of '1's in v. Set MAX =200, thus reset the algorithm 200 times for the global maximum and print the found maximum-value for each reset separated by a comma in the Output.txt file.

Answer:

Simulated Annealing algorithm is a metaheuristic optimization method inspired by the annealing process in metallurgy. It is used to find the global optimum of a given cost function. The algorithm starts with a random solution and iteratively improves it by making small random changes. The acceptance of these changes is guided by a temperature parameter that is gradually decreased over time. The algorithm terminates when the temperature reaches a predefined minimum value or a satisfactory solution is found.

```
# Import required modules
import random
import math
import time
import tracemalloc

# Set the number of iterations (resets)
MAX = 200

# Define the function f
def f(v):
   return abs(14 * sum(v) - 190)

# Simulated Annealing algorithm to find the global maximum of f
def simulated_annealing():
   # Initialize a random binary string of length 50
   v = [random.randint(0, 1) for _ in range(50)]

   # Initialize the temperature
   T = 100
   T_min = 0.00001
   alpha = 0.9

   # Initialize the best solution found so far
   current_best = f(v)
   best_v = v.copy()

   # Loop until the temperature reaches its minimum value
```

```
    while T > T_min:
        # Create a random neighbor solution
        i = random.randint(0, 49)
        v_new = v.copy()
        v_new[i] = 1 - v_new[i]
        delta_E = f(v_new) - f(v)

        # Decide if we should accept the new solution
        if delta_E > 0:
            v = v_new
            current_best = f(v)
            best_v = v.copy()
        else:
            # Accept the new solution with probability exp(delta_E / T)
            acceptance_prob = math.exp(delta_E / T)
            if acceptance_prob > random.uniform(0, 1):
                v = v_new

        # Decrease the temperature
        T *= alpha
    return current_best, best_v

# Start tracking the execution time and memory usage
start_time = time.time()
tracemalloc.start()

results = []
for i in range(MAX):
    result = simulated_annealing() # run the Simulated Annealing algorithm and store the result
    results.append(result)         # append the result to the results list

with open("output.txt", "w") as f:  # Write the results to a text file 'output.txt'
    for max_value, max_v in results:
        f.write(f"{max_value}, {max_v}\n")

# Stop tracking the execution time and memory usage
end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

# Print the execution time and memory usage
print("Execution time:", end_time - start_time, "seconds")
print("Peak memory usage:", peak, "bytes")
```

The simulated annealing algorithm here starts with an initial temperature T and a cooling rate alpha, and runs for a maximum of 200 iterations. For each iteration, a new random binary string is generated and its value of f is calculated. If the current value of f is greater than the maximum value found so far, the maximum value is updated. The temperature T is then decreased by multiplying it by alpha, and if T drops below the minimum temperature T_min, it is reset back to 100. The maximum value of f for each iteration is stored in the results list. Finally, the results list is written to a file named "Output.txt", and the total time taken to run the simulation is printed to the console.

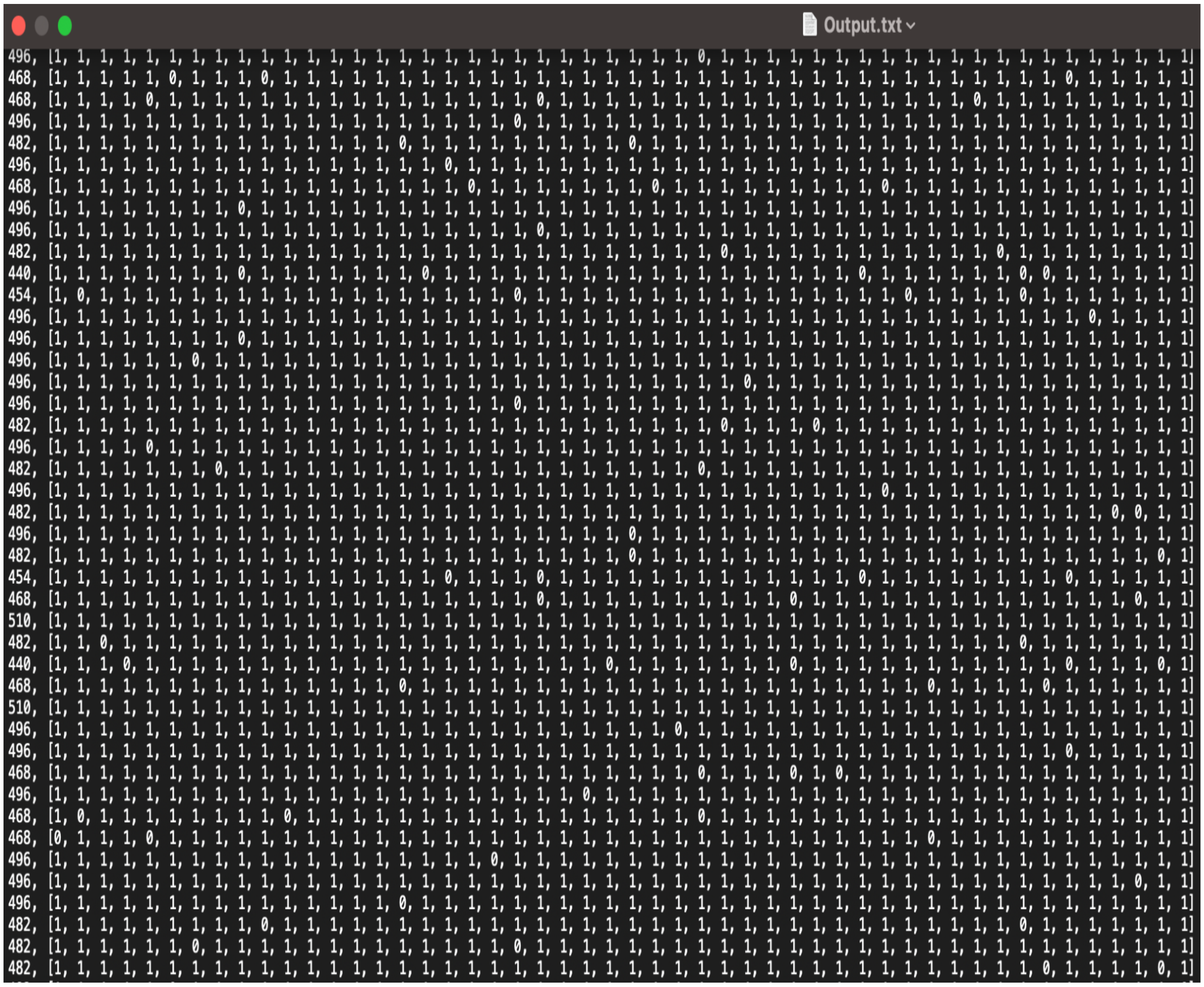
The global maximum of the function f = |14 * one (v) -190|, where v is a 50-bit binary string, is 510. The maximum is achieved when all the bits in the binary string are equal to 1.

In SA, alpha is often used to refer to the cooling rate, which is the rate at which the temperature T decreases over time. The cooling rate determines how quickly the algorithm transitions from exploring the solution space to exploiting the current solution. A high cooling rate allows for more rapid exploration of the solution space, but may also cause the algorithm to converge prematurely to a suboptimal solution. A low cooling rate allows for a more thorough exploration of the solution space, but may also cause the algorithm to take a long time to converge to an optimal solution. The optimal cooling rate depends on the specific problem and the desired trade-off between exploration and exploitation.

Explanation:

This code implements the Simulated Annealing algorithm in Python to find the global maximum of a specific function f.

1. 'MAX' sets the number of iterations or resets of the algorithm.
2. 'f' is a function that returns the absolute value of the difference between the sum of its argument (a list of binary values) and 190, multiplied by 14.
3. The 'simulated_annealing' function contains the main logic of the algorithm.
    a. It starts by initializing a random binary string of length 50, the temperature (T) and the best solution found so far.
    b. The while loop continues until the temp. reaches its minimum value, T_min.
    c. Inside the loop, a random neighbor solution is generated, and its energy difference (delta_E) with the current solution is calculated.
    d. If delta_E is positive, the new solution is accepted as the current solution. If delta_E is negative, the new solution is accepted with probability exp(delta_E/T).
    e. The temperature is decreased by a factor of alpha after each iteration.
    f. The function returns the current best soln. & its value after the loop terminates.
4. The code then executes the Simulated Annealing algorithm MAX times and stores the results in a list results.
5. Finally, the results are written to a text file output.txt.
6. The execution time and memory usage of the code are measured and printed.



Output.txt

```
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
468, [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
468, [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
468, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1]
440, [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1]
454, [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]
454, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
468, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
510, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
440, [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1]
468, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
510, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
468, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
468, [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
468, [0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
496, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
482, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1]
```

Fig: Snapshot of the Output.txt file of the Simultated Annealing (SA) Algorithm