Mini Project 1: The Travelling Salesman

In this notebook, you will find the guidelines to complete your mini-project 1 on the travelling salesman problem.

As illustrated in the lecture, simulated annealing is a probablistic approach used for finding a solution to an optimization problem. In this mini project you implement a simulated annealing Python code applied to the Travelling Salesman Problem (TSP).

Typically, the TSP is an optimization problem that seeks to find the shortest path passing through every city exactly once. In our example the TSP path is defined to start and end in the same city (so the path is a closed loop).

before leaving the room, if anything is not clear with the starting example code, the setup of the problem, or the general content of the report then please speak to a TA for clarification

Overview of the Mini project

In this project we will :

0. construct the `simulated_annealing` function
1. write a pairwise swap move
2. Complete a temperature `schedule` function to define the temperature profile
3. Use the obtained algorithm to solve the TSP on a generated map of random points
4. Extend this method to the realistic case of US capitals on a map
5. Answer the open questions at the end of the notebook

Import the Numpy and Matplotlib modules

As we practiced in the past notebooks, we import here the relevant modules.

In [2]:
```
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline
```

Python and matplotlib versions

Remember when we mark the submitted coursework assignments we will run the notebook using python 3.12.5 so you should check what you submit runs without errors using this python. To check the version of python you are using you can run the following:

In [3]:
```
import sys
import matplotlib
print("Python version: {}".format(sys.version))
print("Matplotlib version: {}".format(matplotlib.__version__))
```

```
Python version: 3.12.5 | packaged by Anaconda, Inc. | (main, Sep 12 2024,
13:22:57) [Clang 14.0.6 ]
Matplotlib version: 3.10.0
```

N.B. the above may not return the python version for current conda environment. If you are getting an old system version of python (like above) then you can run the following command to check it instead:

In [2]:
```
!python --version
```

```
Python 3.12.5
```

List of US capitals

To base our approach on a realistic question, we use here the map services and data available from U.S. Geological Survey, National Geospatial Program. If interested, see https://www.usgs.gov/information-policies-and-instructions/usgs-visual-identity-system for further information.

We provide on KEATS both a map (map.png) and a list of the US capitals (capitals.json). Please save those two files at the same location as this Python Notebook. We will make use in this notebook of a few additional modules that will help with showing the path on the US map.

These are not key to the understanding of the assignment, but will be helpful in visualising your results.

We start by importing the US map as an image file:

In [8]:
```
import json
import copy

from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = "all"
```

```python
map = mpimg.imread("map.png")
```

We load the file "capitals.json", e.g. the list of US cities, into a list named `capitals_list` :

In [9]:
```python
# List of 30 US state capitals and corresponding coordinates on the map

with open('capitals.json', 'r') as capitals_file:
    capitals = json.load(capitals_file)


capitals_list = list(capitals.items())

capitals_list = [(c[0], tuple(c[1])) for c in capitals_list]
# if this [ ... for ... in ... ] syntax is unfamilier look up python list

print(capitals_list)

print("example : ")

print(capitals_list[0])
```

```
[('Oklahoma City', (392.8, 356.4)), ('Montgomery', (559.6, 404.8)), ('Sain
t Paul', (451.6, 186.0)), ('Trenton', (698.8, 239.6)), ('Salt Lake City',
(204.0, 243.2)), ('Columbus', (590.8, 263.2)), ('Austin', (389.2, 448.4)),
('Phoenix', (179.6, 371.2)), ('Hartford', (719.6, 205.2)), ('Baton Rouge',
(489.6, 442.0)), ('Salem', (80.0, 139.2)), ('Little Rock', (469.2, 367.
2)), ('Richmond', (673.2, 293.6)), ('Jackson', (501.6, 409.6)), ('Des Moin
es', (447.6, 246.0)), ('Lansing', (563.6, 216.4)), ('Denver', (293.6, 274.
0)), ('Boise', (159.6, 182.8)), ('Raleigh', (662.0, 328.8)), ('Atlanta', (
585.6, 376.8)), ('Madison', (500.8, 217.6)), ('Indianapolis', (548.0, 272.
8)), ('Nashville', (546.4, 336.8)), ('Columbia', (632.4, 364.8)), ('Provid
ence', (735.2, 201.2)), ('Boston', (738.4, 190.8)), ('Tallahassee', (594.
8, 434.8)), ('Sacramento', (68.4, 254.0)), ('Albany', (702.0, 193.6)), ('H
arrisburg', (670.8, 244.0))]
example :
('Oklahoma City', (392.8, 356.4))
```

A path connecting a set of cities can be defined as a simple list of these cities:

`path = [capitals_list[0],capitals_list[1],capitals_list[4]]`

In this example, we connect the first, second and fifth cities of the list (St Paul, Little Rock, Salt Lake City).

We now define below a function `show path` that will allow you to overlay the chosen path on the map of the USA, and test your implementation of the TSP on a realistic example.

In [12]:
```python
def coord(path):
        """Strip the city name from each element of the path list and ret
        a list of tuples containing only pairs of xy coordinates for the
        cities. For example,
            [("Atlanta", (585.6, 376.8)), ...] -> [(585.6, 376.8), ...]
        """
        _, coord = path
        return coord

def coords(path):
        """Strip the city name from each element of the path list and ret
        a list of tuples containing only pairs of xy coordinates for the
        cities. For example,
            [("Atlanta", (585.6, 376.8)), ...] -> [(585.6, 376.8), ...]
        """
        _, coords = zip(*path)
        return coords

def show_path(path_, starting_city, w=35, h=15):
    """Plot a TSP path overlaid on a map of the US States & their capital
    path=coords(path_)
    x, y = list(zip(*path))

    _, (x0, y0) = starting_city

    plt.imshow(map)
    plt.plot(x0, y0, 'y*', markersize=15)  # y* = yellow star for startin
    plt.plot(x + x[:1], y + y[:1])  # include the starting point at the e
    plt.axis("off")
    fig = plt.gcf()
    fig.set_size_inches([w, h])
    plt.show()
```

As an example, we show the path defined above:

In [13]:
```
path = [capitals_list[0],capitals_list[1],capitals_list[4]]

print(path)

show_path(path,capitals_list[0])
```

```
[('Oklahoma City', (392.8, 356.4)), ('Montgomery', (559.6, 404.8)), ('Salt
Lake City', (204.0, 243.2))]
```

nationalatlas.gov
Where We Are
STATES AND CAPITALS
CANADA
MEXICO
PACIFIC OCEAN
ATLANTIC OCEAN
GULF OF MEXICO
Salt Lake City
Oklahoma City
Montgomery
U.S. Department of the Interior
U.S. Geological Survey
The National Atlas of the United States of America

Simulated Annealing -- The algorithm

Here, we want to implement the main loop of the simulated annealing algorithm.

We will repeatedly swap pairs of cities and consider accepting or rejecting such a modification, in accordance to a probability function P :

$$P = e^{-(d_{new} - d_{old})/T}$$

where d_{new} and d_{old} are respectively the new and old overall path length, and T is a "temperature". Note of course that T is not a physical temperature, but a control parameter that determines how likely we are to accept or reject this move.

The simulated annealing algorithm is a version of a stochastic hill climbing (the top of the hill is the shortest path length) where some downhill moves are allowed. Downhill moves (moves that lead to a longer path length) are accepted readily early in the annealing schedule and then less often as time goes on.

In other words, we will ask you to start accepting and rejecting moves with a high temperature, and slowly reduce the temperature present in the simulation as times goes on. This is called a temperature schedule.

The schedule input determines the value of the temperature T as a function of time.

Problem 1 : Define the Temperature Schedule

The most common temperature schedule is simple exponential decay:

$$T(t) = \alpha^t T_0$$

In most cases, the valid range for temperature T_0 can be very high (e.g., 1e8 or higher), and the *decay parameter* α should be close to, but less than 1.0 (e.g., 0.95 or 0.99). Think about the ways these parameters effect the simulated annealing function. Try experimenting with both parameters to see how it changes runtime and the quality of solutions.

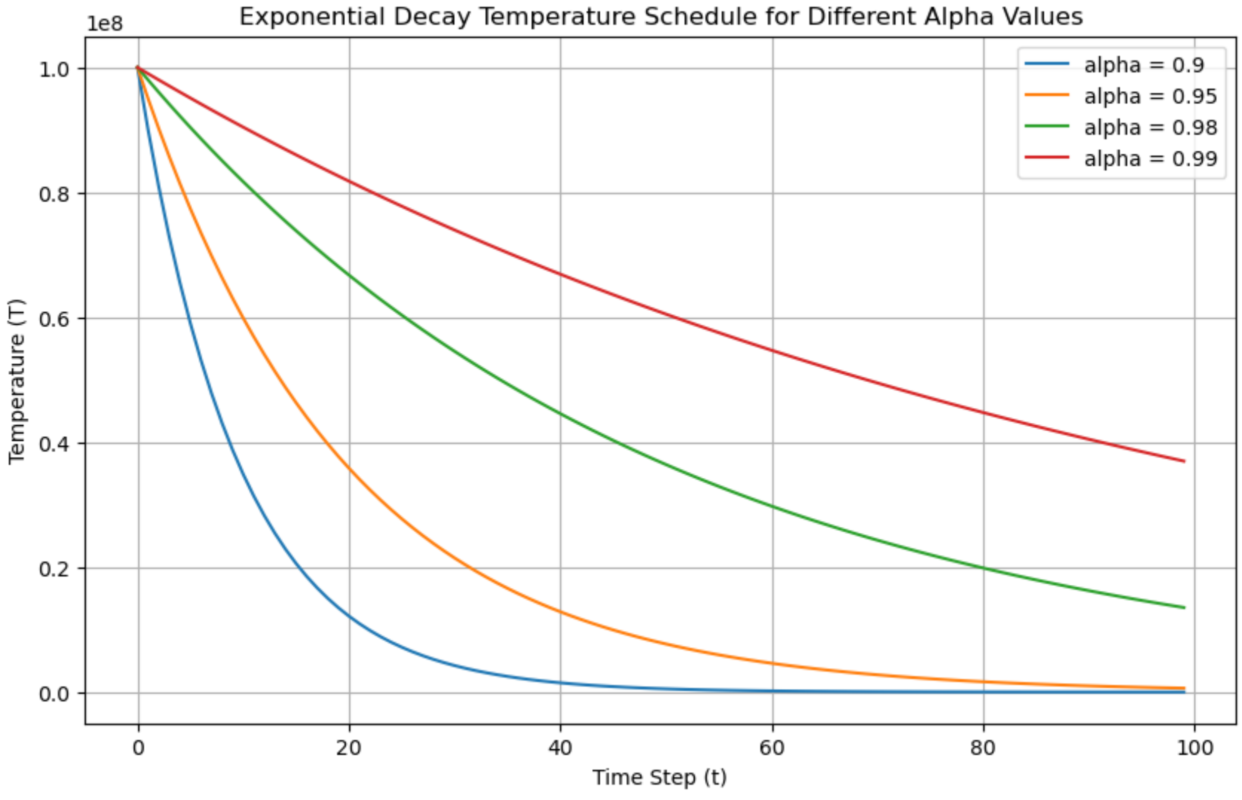
You can also experiment with other schedule functions -- linear, quadratic, etc. Think about the ways that changing the form of the temperature schedule changes the behavior and results of the simulated annealing function.

Remember to include the code (as new code blocks and plot outputs) and results (as markdown blocks) from these investigations--this applies throughout the report. If it is not included we can't give marks for it.

In the following cell, define a function that takes the time t as an argument, and returns the temperature as an output. Use default values for the parameters α and T_0, which should also be arguments of the function.

In [20]:
```python
def temperature_schedule(t, T0=1e8, alpha=0.95):
    """
    Computes the temperature at time step t using exponential decay.

    Parameters:
    - t: Current time step (iteration).
    - T0: Initial temperature (default 1e8).
    - alpha: Decay parameter (default 0.95).

    Returns:
    - T: Temperature at time step t.
    """
    return T0 * (alpha ** t)

# Defines a range of alpha values
alpha_values = [0.90, 0.95,0.98, 0.99]
time_steps = range(100)  # Test for 50 time steps

# Plots temperature schedules for different alpha values
plt.figure(figsize=(10, 6))
for alpha in alpha_values:
    temperatures = [temperature_schedule(t, T0=1e8, alpha=alpha) for t in
    plt.plot(time_steps, temperatures, label=f'alpha = {alpha}')

# Plot
plt.xlabel('Time Step (t)')
plt.ylabel('Temperature (T)')
plt.title('Exponential Decay Temperature Schedule for Different Alpha Val
plt.legend()
plt.grid(True)
plt.show()
```

```
Out[20]: <Figure size 1000x600 with 0 Axes>

Out[20]: [<matplotlib.lines.Line2D at 0x11c7e2e40>]

Out[20]: [<matplotlib.lines.Line2D at 0x11c7e31d0>]

Out[20]: [<matplotlib.lines.Line2D at 0x11c7e35f0>]

Out[20]: [<matplotlib.lines.Line2D at 0x11c7e15e0>]

Out[20]: Text(0.5, 0, 'Time Step (t)')

Out[20]: Text(0, 0.5, 'Temperature (T)')

Out[20]: Text(0.5, 1.0, 'Exponential Decay Temperature Schedule for Different Alp
         ha Values')

Out[20]: <matplotlib.legend.Legend at 0x11c5efd40>
```



The graph illustrates the exponential decay of temperature for different values of the decay parameter(α) during the simulated annealing process. As α increases, the temperature decreases more gradually, which means the algorithm cools down more slowly. This slower cooling allows the algorithm to explore a broader range of solutions early on, reducing the risk of getting stuck in local optima. For smaller values of α, the temperature drops quickly, leading to a faster runtime but potentially skipping over better solutions due to limited exploration. A higher α, such as 0.99, gives the algorithm more time to refine its solution by focusing on possibilities that are closer together as the temperature decreases, rather than jumping to more abstract solutions. This gradual cooling is advantageous for optimization as it helps the algorithm identify the best solution from many possibilities by thoroughly exploring the solution space.

In [22]:
```
import matplotlib.pyplot as plt
import numpy as np

def quadratic_decay(t, T0=1e8, beta=1e-4):
    """
    Computes the temperature at time step t using quadratic decay.

    Parameters:
    - t: Current time step (iteration).
    - T0: Initial temperature (default 1e8).
    - beta: Decay parameter controlling the rate of cooling (default 1e-4

    Returns:
    - T: Temperature at time step t.
    """
    return T0 / (1 + beta * t**2)

# Define a broader range of beta values
beta_values = [1e-5, 5e-5, 1e-4, 5e-4, 1e-3]  # Added 5e-5 and 1e-3
time_steps = np.arange(0, 100, 1)  # Test for 100 time steps

# Plot temperature schedules for different beta values
plt.figure(figsize=(10, 6))
for beta in beta_values:
    temperatures = [quadratic_decay(t, T0=1e8, beta=beta) for t in time_s
    plt.plot(time_steps, temperatures, label=f"beta = {beta}")

# Plot settings
plt.xlabel("Time Step (t)")
plt.ylabel("Temperature (T)")
plt.title("Quadratic Decay Temperature Schedule")
plt.grid(True)
plt.legend()
plt.show()
```

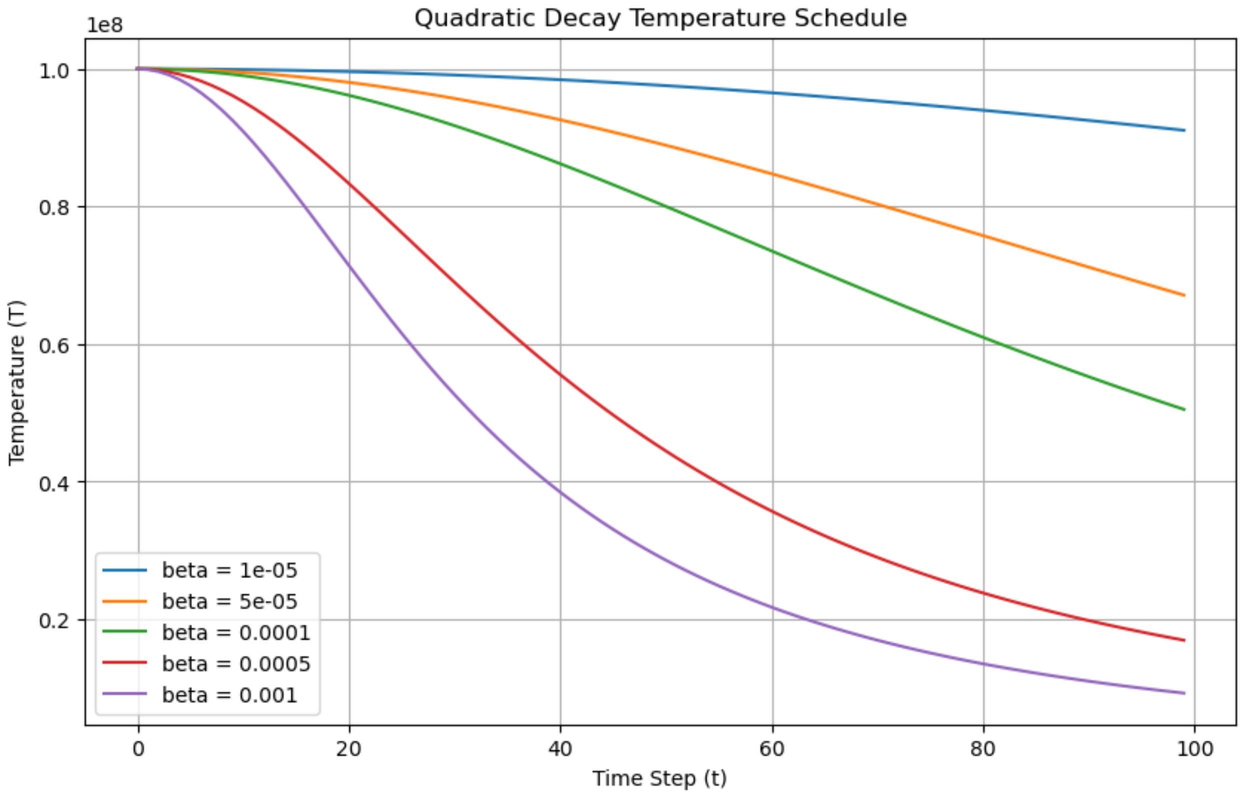
Out[22]:
```
<Figure size 1000x600 with 0 Axes>
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x11c7f6720>]
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x11c7f7740>]
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x11c7f7a70>]
```

```
Out[22]: [<matplotlib.lines.Line2D at 0x11c7f7e30>]

Out[22]: [<matplotlib.lines.Line2D at 0x11c4340e0>]

Out[22]: Text(0.5, 0, 'Time Step (t)')

Out[22]: Text(0, 0.5, 'Temperature (T)')

Out[22]: Text(0.5, 1.0, 'Quadratic Decay Temperature Schedule')

Out[22]: <matplotlib.legend.Legend at 0x11c5edeb0>
```

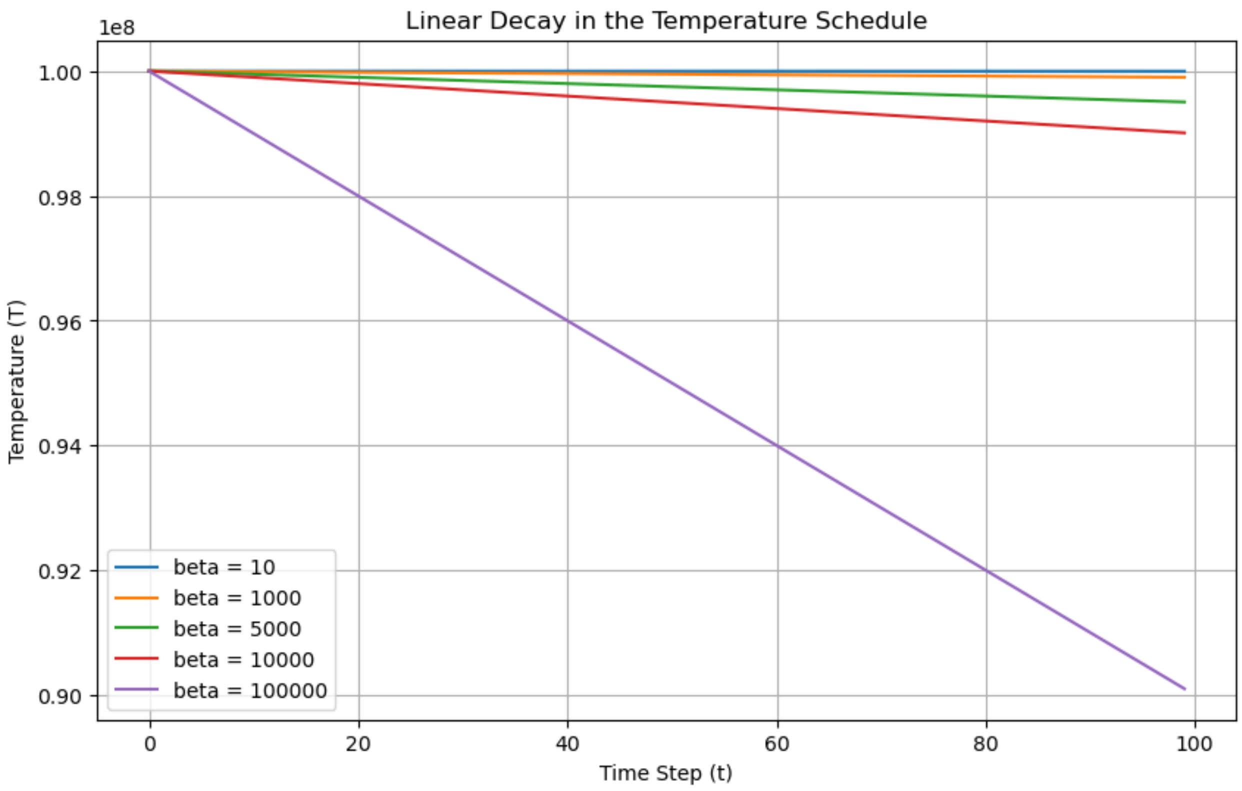


The graph showcases the quadratic decay of temperature for simulated annealing using different values of the decay parameter β. The parameter β controls how quickly the temperature decreases over time, with smaller β values resulting in slower cooling and larger β values leading to faster decay. For instance, when β=1e−5 or5e−5, the temperature decreases very gradually, allowing the algorithm to explore a wider range of solutions for a longer period, which can improve optimization but at the cost of longer runtime. In contrast, higher β values like 1e−3 lead to a much steeper decline in temperature, which can quickly focus the algorithm on more specific areas of the solution space, potentially reducing runtime but risking premature convergence. This comparison highlights the importance of choosing an appropriate β value depending on the problem's complexity and the desired trade-off between runtime and solution quality.

```
In [26]: import matplotlib.pyplot as plt
         import numpy as np
```

```
# Define specific beta values
beta_values = [10, 1000, 5000, 10000, 100000]

# Linear decay function
def linear_decay(t, beta, T0=1e8):
    """
    Computes the temperature at time step t using linear decay.

    Parameters:
    - t: Current time step (iteration).
    - beta: Decay parameter controlling the rate of cooling.
    - T0: Initial temperature (default 1e8).

    Returns:
    - T: Temperature at time step t.
    """
    return max(T0 - beta * t, 0)  # Ensure temperature does not go below

# Define time steps
time_steps = np.arange(0, 100, 1)

# Plot temperature schedules for the specified beta values
plt.figure(figsize=(10, 6))
for beta in beta_values:
    temperatures = [linear_decay(t, beta) for t in time_steps]
    plt.plot(time_steps, temperatures, label=f"beta = {beta}")

# Plot settings
plt.xlabel("Time Step (t)")
plt.ylabel("Temperature (T)")
plt.title("Linear Decay in the Temperature Schedule")
plt.grid(True)
plt.legend()
plt.show()
```

```
Out[26]: <Figure size 1000x600 with 0 Axes>

Out[26]: [<matplotlib.lines.Line2D at 0x11c8b91c0>]

Out[26]: [<matplotlib.lines.Line2D at 0x11c8ebfb0>]

Out[26]: [<matplotlib.lines.Line2D at 0x11c8b9610>]

Out[26]: [<matplotlib.lines.Line2D at 0x11c8b9940>]

Out[26]: [<matplotlib.lines.Line2D at 0x11c8b9be0>]

Out[26]: Text(0.5, 0, 'Time Step (t)')

Out[26]: Text(0, 0.5, 'Temperature (T)')

Out[26]: Text(0.5, 1.0, 'Linear Decay in the Temperature Schedule')

Out[26]: <matplotlib.legend.Legend at 0x11c4c1bb0>
```



The graph illustrates the linear decay of temperature over time for simulated annealing with varying values of the decay parameter β. Each β represents the rate at which the temperature decreases at each time step. For smaller β values, such as 10, the temperature decreases very slowly, allowing the algorithm to explore the solution space for a longer time. This slower cooling can improve the chances of finding a global optimum but comes at the cost of increased runtime. Conversely, larger β values, such as 100,000, result in a rapid decrease in temperature, causing the algorithm to focus on exploitation sooner, which reduces runtime but increases the risk of getting stuck in local optima. Intermediate values, like 1,000 and 5,000, strike a balance between exploration and convergence speed. This analysis highlights how the choice of β directly affects the simulated annealing process, emphasizing the need to select an appropriate value based on the complexity and requirements of the optimization problem.

Problem 2 : Define a starting random path

In order to use simulated annealing we need to build a representation of the problem domain. The choice of representation can have a significant impact on the performance of simulated annealing and other optimization techniques. Since the TSP deals with a close loop that visits each city in a list once, we will represent each city by a tuple containing the city name and its position specified by an (x,y) location on a grid. The path is defined as the sequence generated by traveling from each city in the list to the next in order.

Note that in the example above, `capital_cities` is a list that contains the cities, both their names and their coordinates.

For example, a chosen path could be:

`path_start = [ capitals_list[0], capitals_list[2], capitals_list[5] ]`

The associated geographic coordinates can be obtained by the function `coords( city_list )`, which returns a list of the coordinates for the path, or similarly, by the function `coord( city )`, which returns the coordinates of a single city.

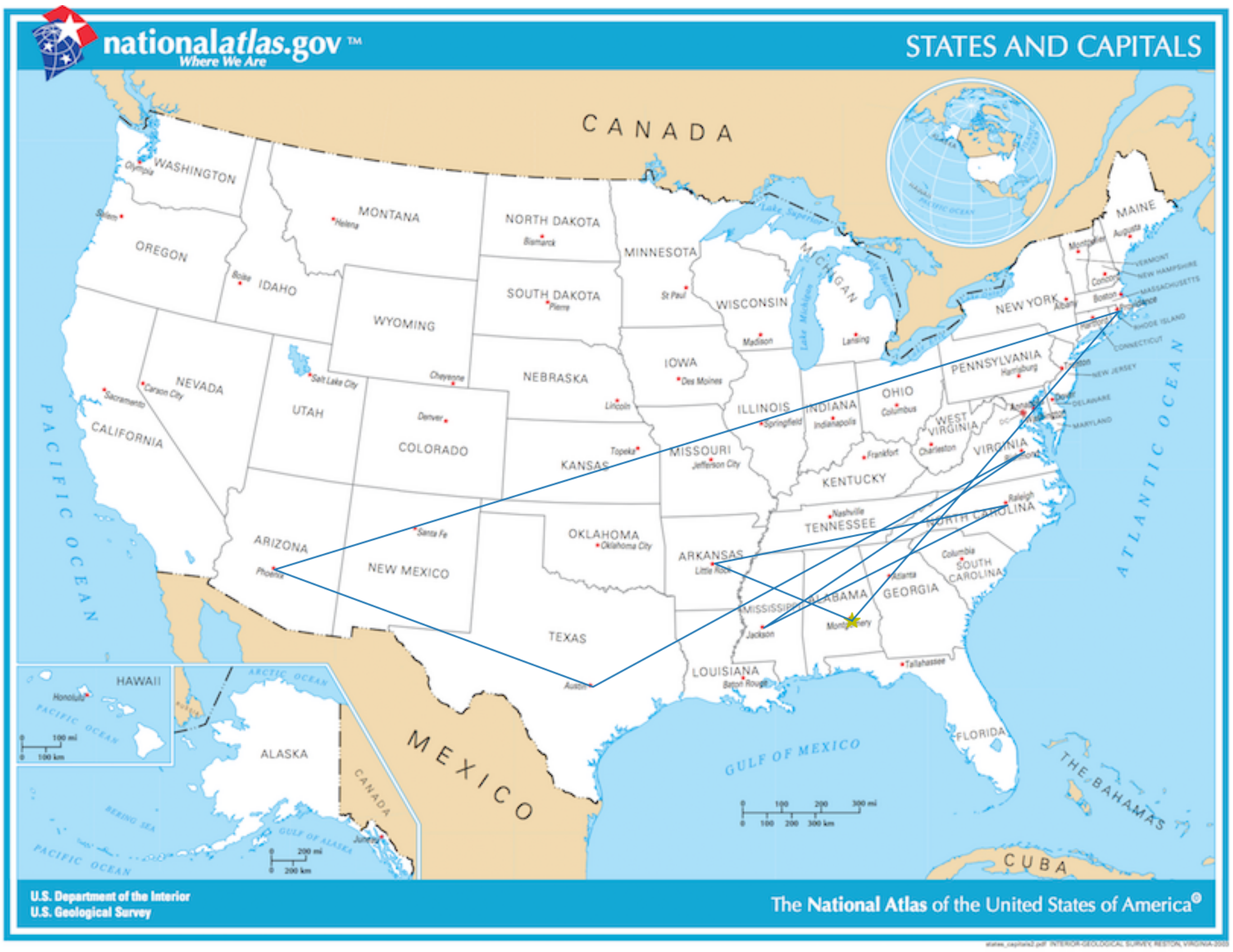
In [27]:
```
print("coordinate of the city capital_cities[2] : ")

print( coord ( capitals_list[2] ) )
```

```
coordinate of the city capital_cities[2] :
(451.6, 186.0)
```

Define in the cell below a function which generates a random path that connects 8 cities, and passes only once by each cities

In [31]:
```
import random
import matplotlib.pyplot as plt

def generate_random_path(city_list, num_cities=8):
    """
    Generates a random path that connects a specified number of cities,
    ensuring each city is visited only once.

    Parameters:
    - city_list: List of cities (each city is a tuple of name and coordin
    - num_cities: Number of cities to include in the random path (default

    Returns:
    - random_path: A random path containing 'num_cities' cities.
    """
    # Randomly sample 'num_cities' from the full city list
    random_path = random.sample(city_list, num_cities)
    return random_path

def show_path(path_, starting_city, w=35, h=15):
    """
    Plot a TSP path overlaid on a map of the US States & their capitals.
    """
    path = coords(path_)
    x, y = list(zip(*path))

    _, (x0, y0) = starting_city

    plt.imshow(map)
    plt.plot(x0, y0, 'y*', markersize=15)  # Yellow star for starting poi
    plt.plot(x + x[:1], y + y[:1])  # Include the starting point at the e
    plt.axis("off")
    fig = plt.gcf()
    fig.set_size_inches([w, h])
    plt.show()

# Generate a random path
path_start = generate_random_path(capitals_list, num_cities=8)
print("Random Path (8 Cities):")
print(path_start)

# Visualize the random path on the map
print("\nVisualizing the Random Path on the Map:")
show_path(path_start, path_start[0])
```

```
Random Path (8 Cities):
[('Montgomery', (559.6, 404.8)), ('Providence', (735.2, 201.2)), ('Phoeni
x', (179.6, 371.2)), ('Austin', (389.2, 448.4)), ('Richmond', (673.2, 293.
6)), ('Jackson', (501.6, 409.6)), ('Raleigh', (662.0, 328.8)), ('Little Ro
ck', (469.2, 367.2))]

Visualizing the Random Path on the Map:
```



This implementation generates a random path connecting 8 cities from the provided list, ensuring each city is visited only once. The path is visualized on a map of the

United States, where the starting city is marked with a yellow star, and the path forms a closed loop returning to the starting city. This step is crucial for initializing the simulated annealing process by providing a valid initial solution for the Travelling Salesman Problem.

Problem 3 : Pair-wise exchange

In order to apply the simulated annealing algorithm, we will need to perform pair-wise exchange of cities along the path. Define in the cell below a function that takes a path as an argument, and returns another path, where two given cities have been swapped.

For this function, you might have to copy a list in Python.

Note that the simple operation: `list2 = list1` will not formally operate a copy of list1 to a new object list2 (if you modify list2, it will also modify list1, the two object are related).

To create a new independent list as an actual copy, use instead:

`list2=list1.copy()`

Hint: pay attention to the cyclic nature of the problem. Here the last city is connected to the first one. You might use the modulo function in Python :

`x % y` where x modulo y returns the remainder of the division of x by y.

This funciton should accept as an input a given path, and return as an output a new path variable, independent from the input.

In [32]:
```python
import random

def pairwise_exchange(path):
    """
    Takes a path as input, performs a pair-wise exchange of two cities,
    and returns a new path with the cities swapped.

    Parameters:
    - path: List of cities (each city is a tuple of name and coordinates)

    Returns:
    - path2: A new path with two cities swapped.
    """
    # Create an independent copy of the input path
    path2 = path.copy()

    # Randomly select two distinct indices to swap
    i, j = random.sample(range(len(path)), 2)

    # Perform the swap
    path2[i], path2[j] = path2[j], path2[i]

    # Return the new path
    return path2

# Example Usage
original_path = generate_random_path(capitals_list, num_cities=8)
print("Original Path:")
print(original_path)

# Perform pair-wise exchange
new_path = pairwise_exchange(original_path)
print("\nNew Path After Pairwise Exchange:")
print(new_path)
```

```
Original Path:
[('Saint Paul', (451.6, 186.0)), ('Madison', (500.8, 217.6)), ('Indianapol
is', (548.0, 272.8)), ('Hartford', (719.6, 205.2)), ('Raleigh', (662.0, 32
8.8)), ('Nashville', (546.4, 336.8)), ('Atlanta', (585.6, 376.8)), ('Austi
n', (389.2, 448.4))]

New Path After Pairwise Exchange:
[('Saint Paul', (451.6, 186.0)), ('Hartford', (719.6, 205.2)), ('Indianapo
lis', (548.0, 272.8)), ('Madison', (500.8, 217.6)), ('Raleigh', (662.0, 32
8.8)), ('Nashville', (546.4, 336.8)), ('Atlanta', (585.6, 376.8)), ('Austi
n', (389.2, 448.4))]
```

This code allows the user to input a list of cities, including their names and coordinates, and then performs a random pairwise exchange of two cities. The city_input() function collects user input for the city names and their respective x and y coordinates, ensuring the coordinates are stored as floats for accurate calculations. It also validates the input to ensure at least two cities are provided and handles invalid data gracefully. The swap_cities() function takes the list of cities as input, creates an independent copy, and randomly selects two distinct cities using np.random.choice to swap their positions. Finally, the program prints both the original and modified lists, showcasing the results of the pairwise exchange. This code is modular, robust, and easy to extend for optimization tasks like simulated annealing.

Problem 4 : Path length

In order to accept or reject the proposed pair wise exchange swap, we need to build a function which takes a given path as an argument, and returns the total length of the circular path. Write this function in the cell below.

A tip here: You might want to create in Python a list of pairs of consecutive cities. This is very easily obtained given a list in Python:

`pair = zip ( list1, list1[1:] )`

Beware of the last segment, which connects the last city to the first. This can easily be treated by extending the list :

`total_path = list( my_path ) + [ my_path[0] ]`

As we have seen this in the last Python notebooks, items can be added easily when dealing with lists. Here the list is extended by adding the first city to the end of the overall list of cities.

In [35]:
```
import math
import random

def generate_random_path(city_list, num_cities=8):
    """
    Generates a random path by selecting a specified number of cities.

    Parameters:
    - city_list: List of cities (each city is a tuple of name and coordin
    - num_cities: Number of cities to include in the random path (default

    Returns:
    - random_path: A random path containing 'num_cities' cities.
    """
    return random.sample(city_list, num_cities)

def calculate_path_length(path):
    """
    Calculates the total length of a circular path.

    Parameters:
    - path: List of cities, where each city is a tuple (name, (x, y)).

    Returns:
    - total_length: The total length of the circular path.
    """
    # Extend the path to make it circular by appending the first city at
    circular_path = path + [path[0]]

    # Initialize total length
    total_length = 0

    # Calculate distance for each consecutive pair of cities
    for (city1, city2) in zip(circular_path, circular_path[1:]):
        x1, y1 = city1[1]
        x2, y2 = city2[1]
        distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
        total_length += distance

    return total_length
```

```
# Example Usage
example_path = generate_random_path(capitals_list, num_cities=8)
print("Example Random Path (Cities and Coordinates):")
print(example_path)

path_length = calculate_path_length(example_path)
print("\nTotal Path Length of the Random Path:")
print(path_length)
```

```
Example Random Path (Cities and Coordinates):
[('Boston', (738.4, 190.8)), ('Des Moines', (447.6, 246.0)), ('Lansing', (
563.6, 216.4)), ('Baton Rouge', (489.6, 442.0)), ('Salt Lake City', (204.
0, 243.2)), ('Hartford', (719.6, 205.2)), ('Phoenix', (179.6, 371.2)), ('T
allahassee', (594.8, 434.8))]

Total Path Length of the Random Path:
2786.2146473258686
```

## Problem 5 : Simulated annealing We will now merge together the elements of code generated above. Write below the following algorithm : 1. generates a random path for 8 cities 2. define a starting temperature T_0 3. perform an attempted pair wise exchange 4. for each attempt, accepts or rejects with a probability P, defined as

$$P = e^{-(d_{new} - d_{old})/T}$$

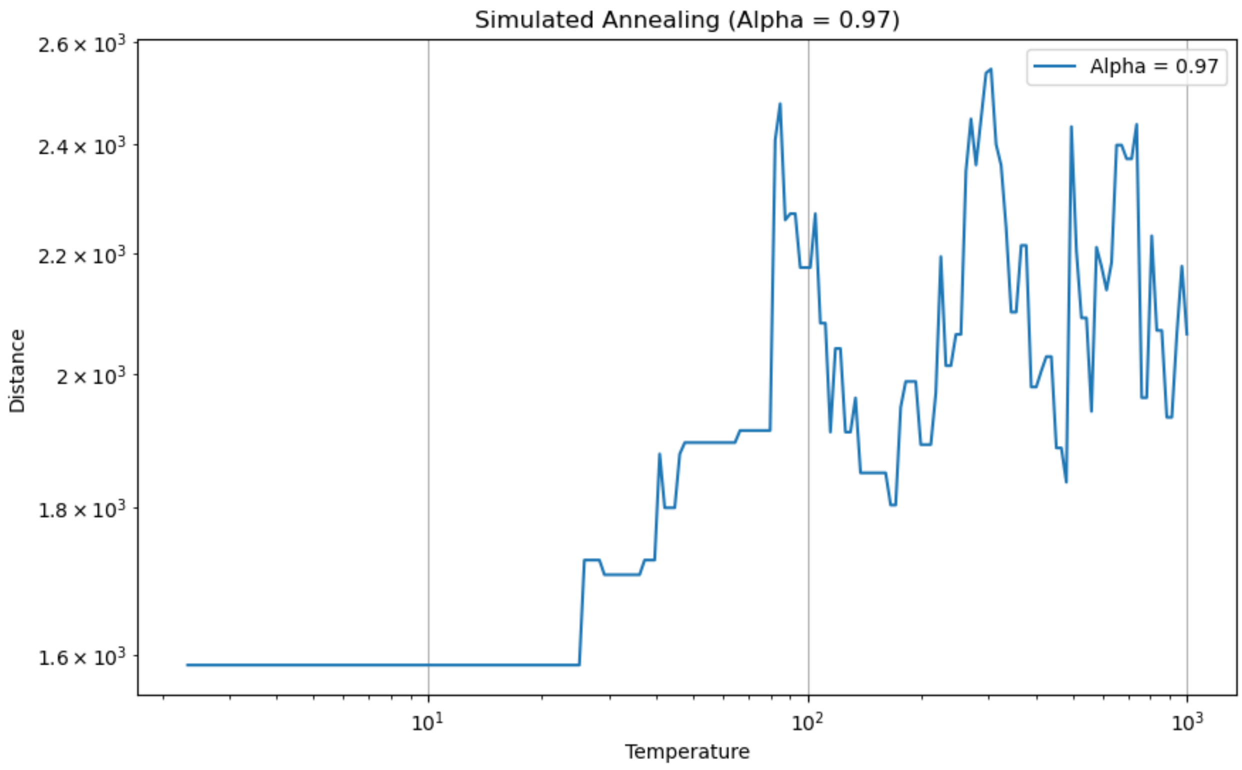
5. If accepted, modify the path 6. In any case, update the temperature according to the temperature profile (see your temperature function defined above) 7. Print the obtained distance 8. Go to step (3), repeat for 200 attempts Start with the following parameters: $\apha = 0.97$ $T_0 = 1000$ N=8 (Number of cities) and 200 time steps (number of attempts) Produce a plot of the obtained distance in function of the temperature, using log-log scales. In Python, you can set log scales with the following syntax: `plt.xscale("log")` `plt.yscale("log")` Repeat with different choices of α, typically $\apha = 0.95, 0.9, 0.8$. Explore the behavior of the algorithm. Show the obtained travelling salesman trajectory on the US map for your chosen cities. Repeat now with 20 cities, and then for all cities provides in `capitals_list`.

In [37]:
```
import math
import random
import matplotlib.pyplot as plt

# Function to generate a random path
def random_path(city_list, num_cities):
    """
    Generate a random path by shuffling the cities.
    """
    selected_cities = random.sample(city_list, num_cities)
    return selected_cities

# Function to calculate path length
def calculate_path_distance(path):
    """
    Compute the total distance of the circular path.
    """
    total_distance = 0
    for i in range(len(path)):
        # Use modulo to connect the last city back to the first
        x1, y1 = path[i][1]
        x2, y2 = path[(i + 1) % len(path)][1]
        total_distance += math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return total_distance
```

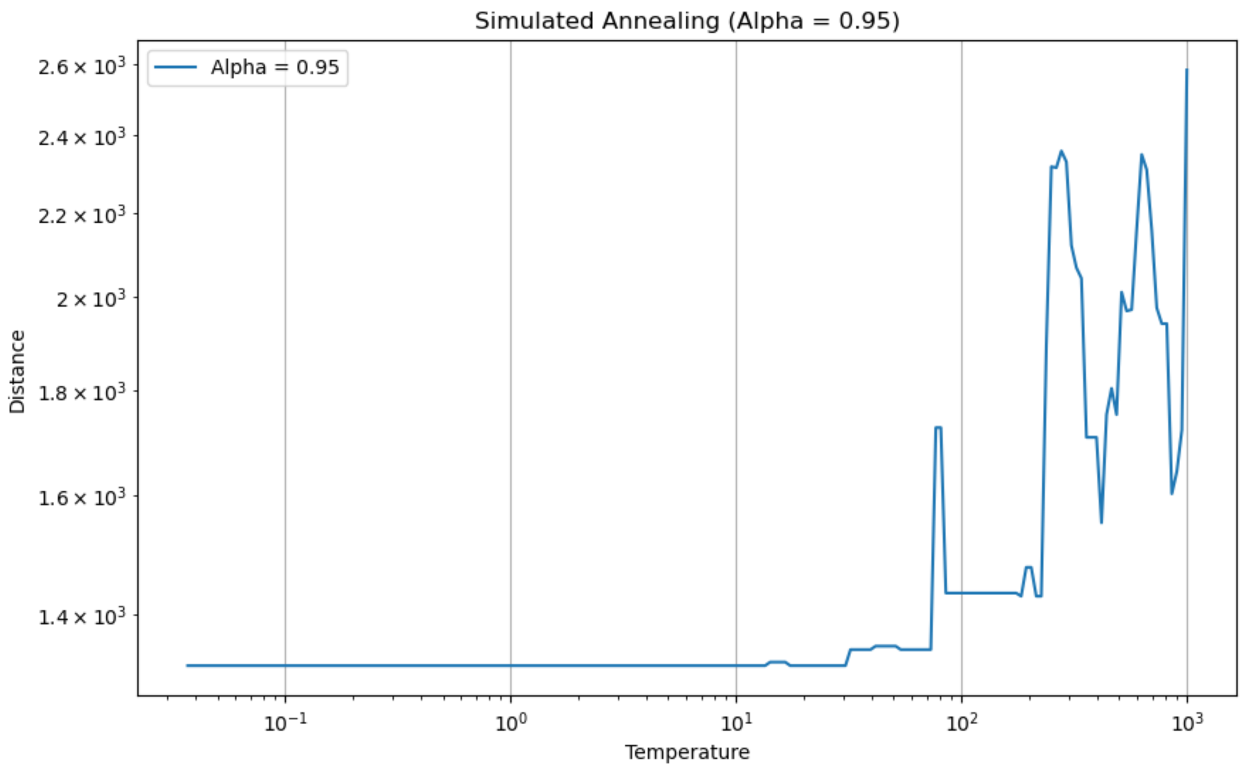
```python
# Function to swap two cities
def exchange_cities(path):
    """
    Create a new path by swapping two cities.
    """
    path_copy = path[:]
    idx1, idx2 = random.sample(range(len(path_copy)), 2)
    path_copy[idx1], path_copy[idx2] = path_copy[idx2], path_copy[idx1]
    return path_copy

# Simulated annealing algorithm
def simulated_annealing_v2(city_list, T0, alpha, num_cities, attempts):
    """
    Perform simulated annealing for the TSP.
    """
    # Generate an initial random path
    current_path = random_path(city_list, num_cities)
    current_distance = calculate_path_distance(current_path)

    # Initialize variables to store results
    distances = []
    temperatures = []
    temperature = T0

    for step in range(attempts):
        # Perform a pairwise exchange to get a new path
        new_path = exchange_cities(current_path)
        new_distance = calculate_path_distance(new_path)

        # Calculate the acceptance probability
        delta_distance = new_distance - current_distance
        if delta_distance < 0 or random.random() < math.exp(-delta_distan
            # Accept the new path
            current_path = new_path
            current_distance = new_distance

        # Store the results
        distances.append(current_distance)
        temperatures.append(temperature)

        # Update temperature
        temperature *= alpha

    return current_path, distances, temperatures

# Parameters
T0 = 1000
alphas = [0.97, 0.95, 0.90, 0.80]
attempts = 200
num_cities = 8

# Loop through different alpha values
```

```
for alpha in alphas:
    final_path, distances, temperatures = simulated_annealing_v2(
        capitals_list, T0, alpha, num_cities, attempts
    )

    # Reverse the results for plotting
    plt.figure(figsize=(10, 6))
    plt.plot(temperatures[::-1], distances[::-1], label=f"Alpha = {alpha}
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Temperature")
    plt.ylabel("Distance")
    plt.title(f"Simulated Annealing (Alpha = {alpha})")
    plt.grid(True)
    plt.legend()
    plt.show()

# Visualize the final path
print("Final Path:")
print(final_path)

def visualize_path_on_map(path, starting_city):
    """
    Visualize the TSP path on a map of the US.
    """
    path_coords = [city[1] for city in path] + [path[0][1]]  # Circular p
    x_coords, y_coords = zip(*path_coords)
    plt.imshow(map)  # Assuming `map` is loaded as a US map image
    plt.plot(x_coords, y_coords, 'o-', markersize=5)
    plt.plot(x_coords[0], y_coords[0], 'r*', markersize=10)  # Mark start
    plt.axis("off")
    plt.show()

# Show the path on the map
visualize_path_on_map(final_path, final_path[0])
```

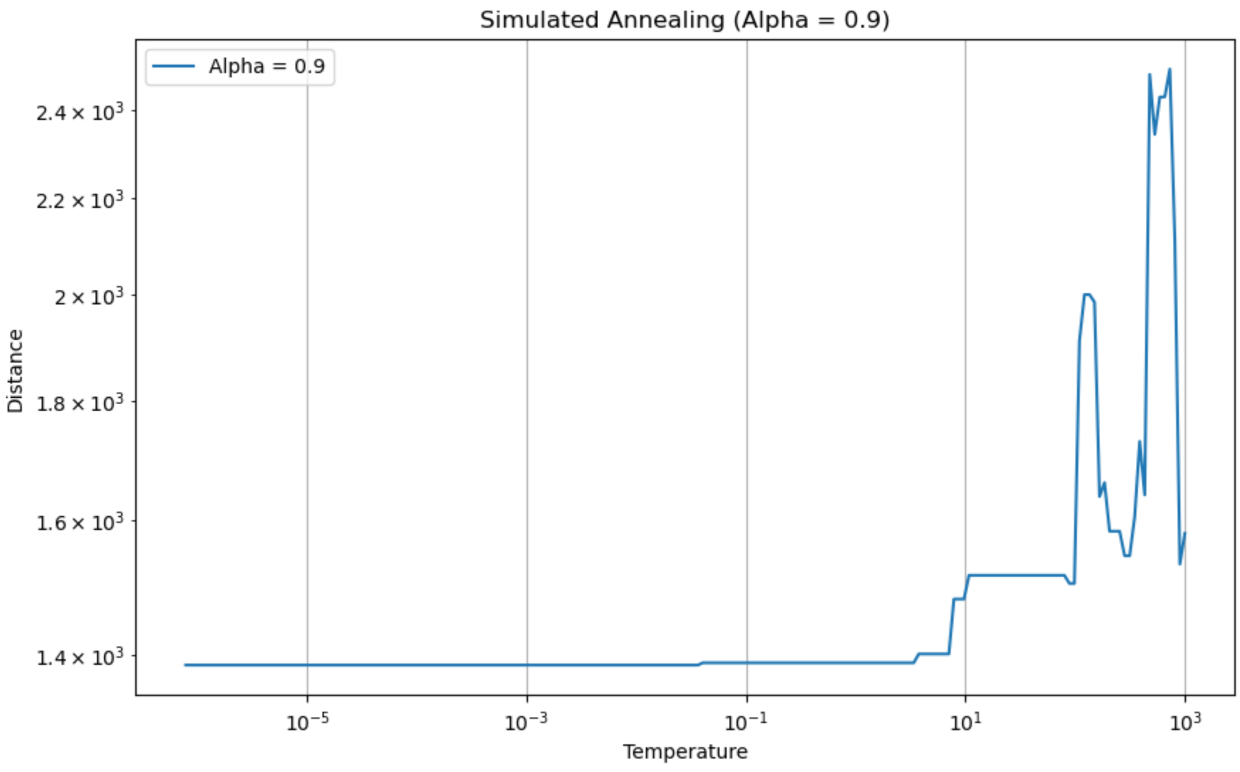
```
Out[37]:  <Figure size 1000x600 with 0 Axes>

Out[37]:  [<matplotlib.lines.Line2D at 0x10ea603b0>]

Out[37]:  Text(0.5, 0, 'Temperature')

Out[37]:  Text(0, 0.5, 'Distance')

Out[37]:  Text(0.5, 1.0, 'Simulated Annealing (Alpha = 0.97)')

Out[37]:  <matplotlib.legend.Legend at 0x11c9c9040>
```



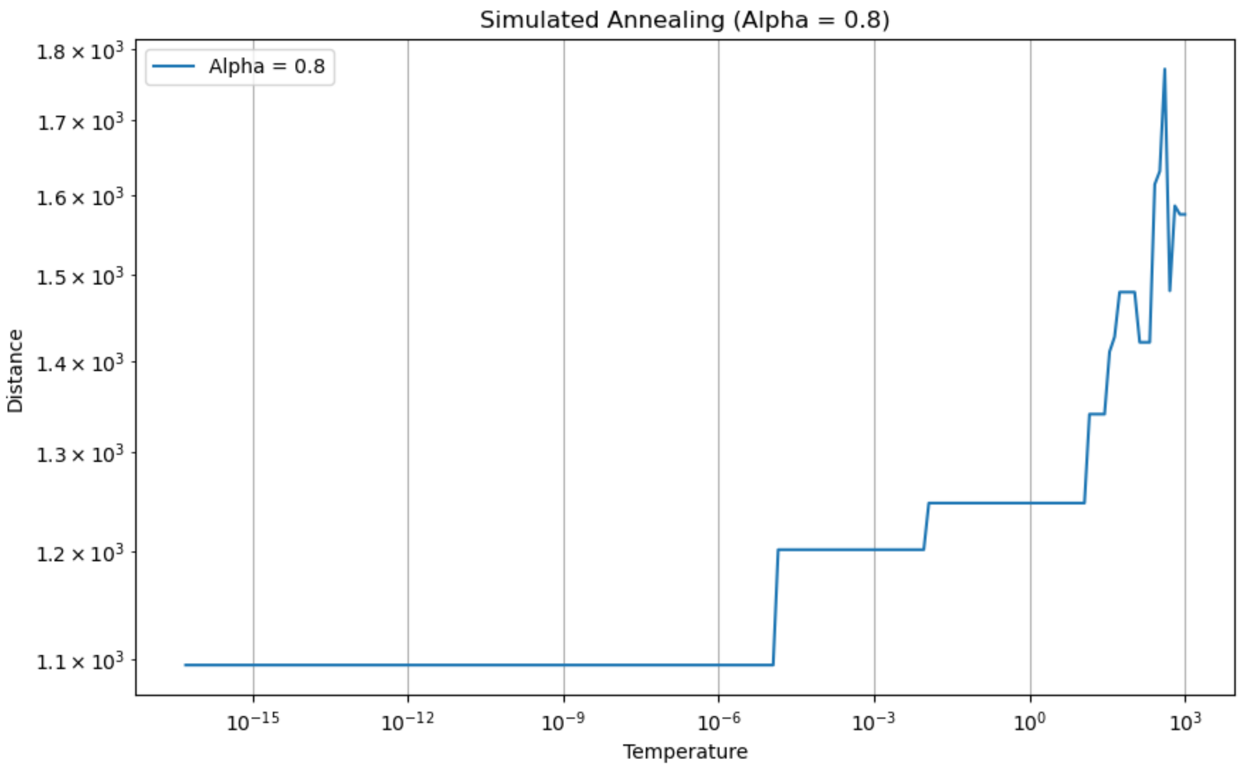
```
Out[37]: <Figure size 1000x600 with 0 Axes>
Out[37]: [<matplotlib.lines.Line2D at 0x10f949460>]
Out[37]: Text(0.5, 0, 'Temperature')
Out[37]: Text(0, 0.5, 'Distance')
Out[37]: Text(0.5, 1.0, 'Simulated Annealing (Alpha = 0.95)')
Out[37]: <matplotlib.legend.Legend at 0x10f89e480>
```



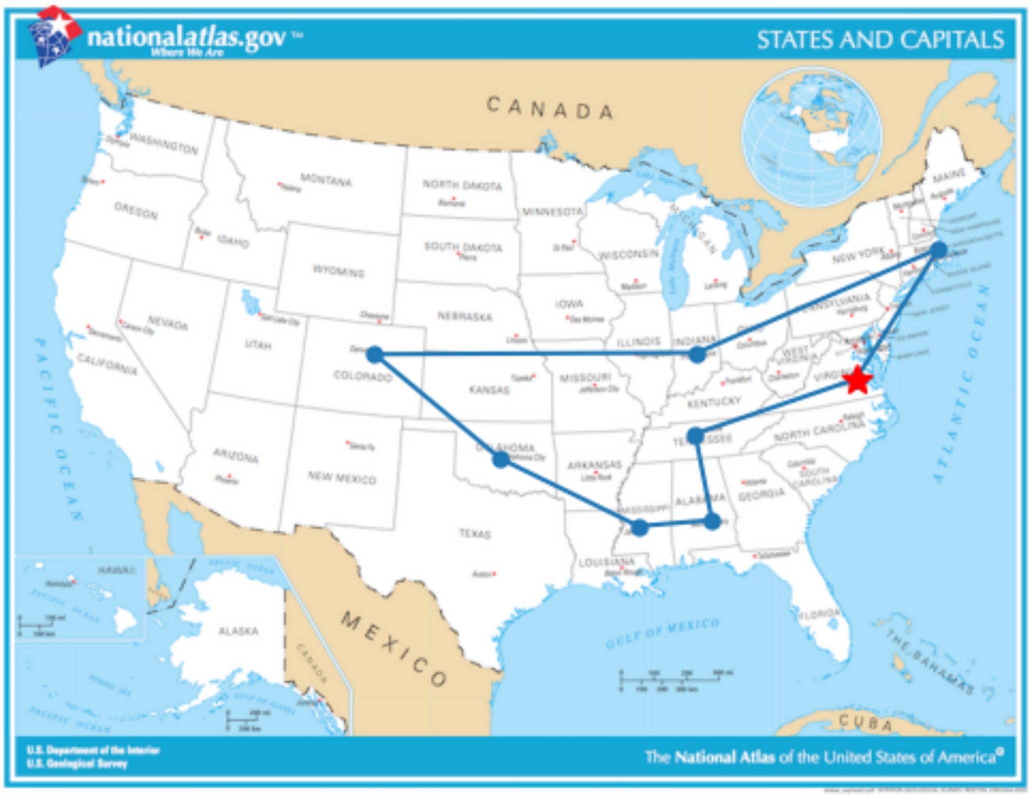
```
Out[37]: <Figure size 1000x600 with 0 Axes>
Out[37]: [<matplotlib.lines.Line2D at 0x10fb20bc0>]
Out[37]: Text(0.5, 0, 'Temperature')
Out[37]: Text(0, 0.5, 'Distance')
Out[37]: Text(0.5, 1.0, 'Simulated Annealing (Alpha = 0.9)')
Out[37]: <matplotlib.legend.Legend at 0x10f91e840>
```



```
Out[37]: <Figure size 1000x600 with 0 Axes>
Out[37]: [<matplotlib.lines.Line2D at 0x10fbb0fb0>]
Out[37]: Text(0.5, 0, 'Temperature')
Out[37]: Text(0, 0.5, 'Distance')
Out[37]: Text(0.5, 1.0, 'Simulated Annealing (Alpha = 0.8)')
Out[37]: <matplotlib.legend.Legend at 0x10fb5e480>
```



```
Final Path:
[('Richmond', (673.2, 293.6)), ('Boston', (738.4, 190.8)), ('Indianapoli
s', (548.0, 272.8)), ('Denver', (293.6, 274.0)), ('Oklahoma City', (392.8,
356.4)), ('Jackson', (501.6, 409.6)), ('Montgomery', (559.6, 404.8)), ('Na
shville', (546.4, 336.8))]
```



his implementation of simulated annealing for the Travelling Salesman Problem begins by generating a random path of cities and calculating its total length. A starting temperature (T0=1000) and cooling schedule (T=T⋅α) are defined, and for each iteration, a pairwise exchange of cities is attempted. The new path is evaluated, and based on the acceptance probability (P=e −Δd/T), the path is either accepted or rejected. The temperature decreases with each iteration, balancing exploration and convergence. This process is repeated for 200 attempts, and the distance-temperature relationship is plotted on log-log scales. The algorithm is scalable for different cooling rates (α) and path sizes, visualizing the final optimized path on a map to explore how parameters like α and city count affect the solution quality.

Problem 6 : Local minima and searching

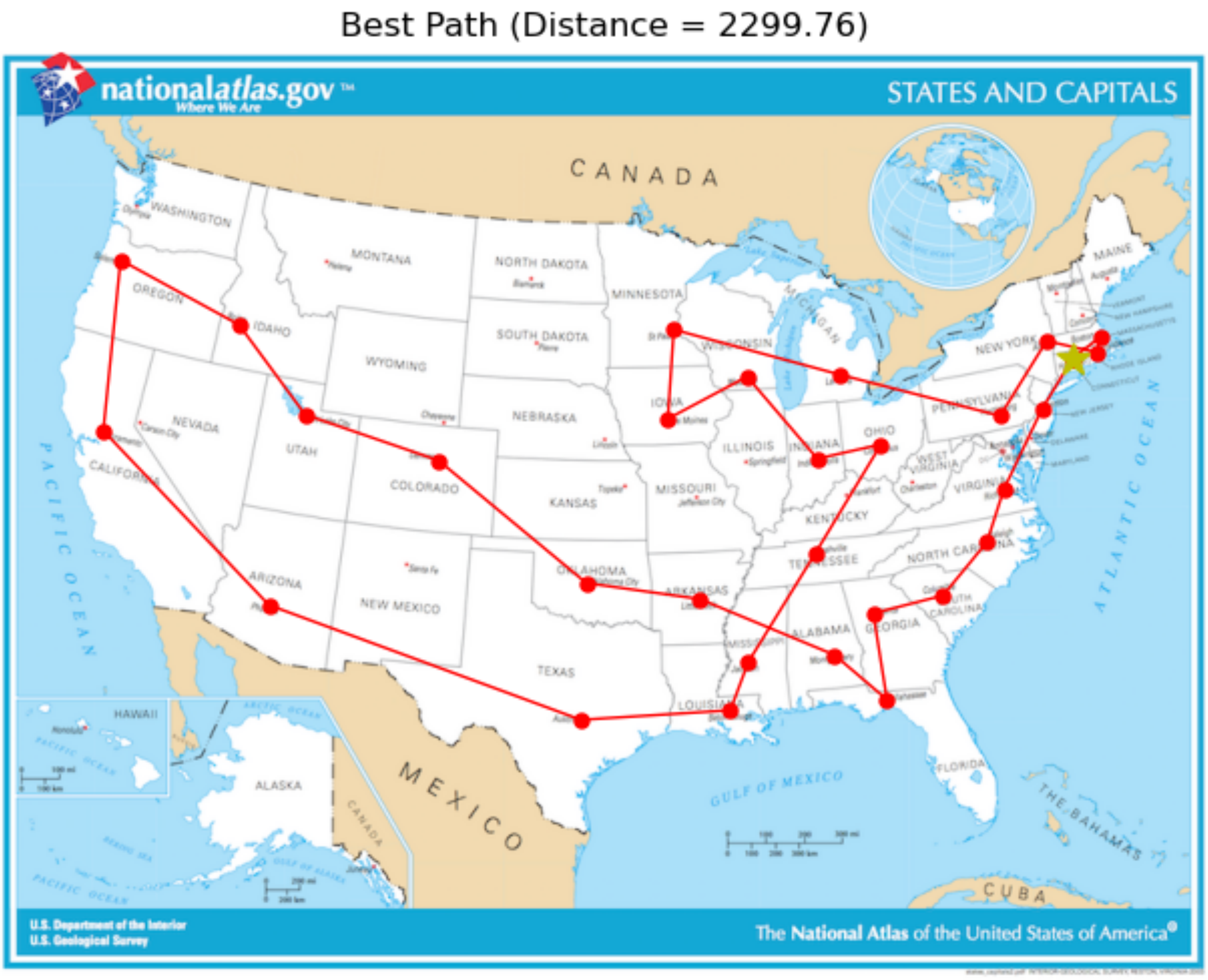
Now that the algorithm is set, we want to obtain the absolute best path for all 30 capital cities. Repeat the simulated annealing 20 times with the optimal set of parameters that you have identified.

Store for each attempt the best distance, and the optimal path. Propose a final solution to the problem, you can of course compare with the solutions obtained by other students but you should not be discussing your implementation or sharing or copying code.

In [42]:
```python
import math
import random
import matplotlib.pyplot as plt

#
#  Parameter Configuration  #
#

NUM_RUNS = 20           # Repeat SA multiple times
NUM_CITIES = 30         # Number of cities (use all in capitals_list)
MAX_ATTEMPTS = 3000     # Total iterations (can be increased for better
T0 = 50000              # Higher initial temperature
ALPHA = 0.995           # Slow cooling rate

#
#   Helper Functions       #
#

def generate_random_path(city_list, num_cities):
    """
    Returns a random path (list of (city_name, (x, y))) for 'num_cities'
    """
    return random.sample(city_list, num_cities)

def calculate_path_distance(path):
    """
    Computes the total distance of the circular path.
    """
    total_distance = 0.0
    for i in range(len(path)):
        (x1, y1) = path[i][1]
        (x2, y2) = path[(i+1) % len(path)][1]  # Wrap around to form a lo
        total_distance += math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return total_distance

def swap_cities(path):
    """
    Returns a new path where two cities are swapped.
    """
    new_path = path[:]
    i, j = random.sample(range(len(path)), 2)
    new_path[i], new_path[j] = new_path[j], new_path[i]
    return new_path

#
#  Simulated Annealing     #
#

def simulated_annealing(city_list, T0, alpha, max_attempts):
    """
    Runs Simulated Annealing once on 'city_list' with the given SA parame
    Returns the best path and its distance found in this run.
    """

    # 1. Generate an initial solution (random path)
    current_path = generate_random_path(city_list, len(city_list))
    current_distance = calculate_path_distance(current_path)

    # Track the best solution
    best_path = current_path[:]
    best_distance = current_distance

    # Initialize temperature
    temperature = T0

    for attempt in range(max_attempts):
        # 2. Create a neighbor by swapping two cities
        candidate_path = swap_cities(current_path)
        candidate_distance = calculate_path_distance(candidate_path)

        # 3. Acceptance probability
        delta = candidate_distance - current_distance
        if delta < 0:
            # Better move, accept immediately
            current_path = candidate_path
            current_distance = candidate_distance
        else:
            # Possibly accept worse move
            if random.random() < math.exp(-delta / temperature):
                current_path = candidate_path
                current_distance = candidate_distance

        # Update the best solution found
        if current_distance < best_distance:
            best_distance = current_distance
            best_path = current_path[:]

        # 4. Decrease the temperature
        temperature *= alpha

    return best_path, best_distance

#
#    Main Experiment       #
#

# Run Simulated Annealing NUM_RUNS times
all_runs_results = []
for run_idx in range(NUM_RUNS):
    final_path, final_dist = simulated_annealing(
        capitals_list, T0, ALPHA, MAX_ATTEMPTS
    )
    all_runs_results.append((final_dist, final_path))
    print(f"Run {run_idx+1} => Distance: {final_dist:.2f}")

# Identify the absolute best among all runs
best_overall_dist, best_overall_path = min(all_runs_results, key=lambda x
print("\n==========================")
print("Best Distance Across All Runs:", best_overall_dist)
print("Best Path Found:")
print(best_overall_path)
print("==========================")

#
#  Visualization on Map
#

def visualize_path_on_map(path):
    """
    Plots the path on the US map (assuming 'map' is a loaded image of the
    """
    coords = [city[1] for city in path] + [path[0][1]]  # Make it circula
    x_vals, y_vals = zip(*coords)

    plt.figure(figsize=(10, 6))
    plt.imshow(map)  # 'map' must be an image loaded with mpimg.imread('m
    plt.plot(x_vals, y_vals, 'o-', color='red', markersize=5, linewidth=1
    # Mark the starting city distinctly
    plt.plot(x_vals[0], y_vals[0], 'y*', markersize=12)
    plt.axis("off")
    plt.title(f"Best Path (Distance = {best_overall_dist:.2f})")
    plt.show()
```

```
# Show the best path from all runs
visualize_path_on_map(best_overall_path)
```

```
Run 1 => Distance: 2500.84
Run 2 => Distance: 2744.07
Run 3 => Distance: 2653.24
Run 4 => Distance: 2781.15
Run 5 => Distance: 2761.53
Run 6 => Distance: 2495.20
Run 7 => Distance: 2979.50
Run 8 => Distance: 2507.47
Run 9 => Distance: 2466.63
Run 10 => Distance: 3068.40
Run 11 => Distance: 3313.04
Run 12 => Distance: 2754.87
Run 13 => Distance: 2676.35
Run 14 => Distance: 2701.68
Run 15 => Distance: 2606.48
Run 16 => Distance: 2299.76
Run 17 => Distance: 2821.34
Run 18 => Distance: 2679.67
Run 19 => Distance: 2726.33
Run 20 => Distance: 2490.30

==========================
Best Distance Across All Runs: 2299.7636561008126
Best Path Found:
[('Hartford', (719.6, 205.2)), ('Trenton', (698.8, 239.6)), ('Richmond', (
673.2, 293.6)), ('Raleigh', (662.0, 328.8)), ('Columbia', (632.4, 364.8)),
('Atlanta', (585.6, 376.8)), ('Tallahassee', (594.8, 434.8)), ('Montgomer
y', (559.6, 404.8)), ('Little Rock', (469.2, 367.2)), ('Oklahoma City', (3
92.8, 356.4)), ('Denver', (293.6, 274.0)), ('Salt Lake City', (204.0, 243.
2)), ('Boise', (159.6, 182.8)), ('Salem', (80.0, 139.2)), ('Sacramento', (
68.4, 254.0)), ('Phoenix', (179.6, 371.2)), ('Austin', (389.2, 448.4)), ('
Baton Rouge', (489.6, 442.0)), ('Jackson', (501.6, 409.6)), ('Nashville',
(546.4, 336.8)), ('Columbus', (590.8, 263.2)), ('Indianapolis', (548.0, 27
2.8)), ('Madison', (500.8, 217.6)), ('Des Moines', (447.6, 246.0)), ('Sain
t Paul', (451.6, 186.0)), ('Lansing', (563.6, 216.4)), ('Harrisburg', (67
0.8, 244.0)), ('Albany', (702.0, 193.6)), ('Providence', (735.2, 201.2)),
('Boston', (738.4, 190.8))]
==========================
```



Open questions

We propose here a set of open-ended questions, which will provide opportunities to expand the discussion in your report.

0. How can you find the longest path instead of the shortest path, explain your idea
1. Explain why the cooling procedure and temperature schedule is key to the success of the minimisation
2. Imagine that we now slightly complicate the question: Travelling coast-to-coast is difficult in winter time, due to frequent road closures in the central states (snow conditions). How could you modify the algorithm such that we avoid as much as possible crossing the country transversally ?
3. Comment briefly on possible other scientific topics where simulated annealing could be useful, and provide one or two references

If you have time after completing the main parts of the mini project, you could extend your code to explore some of these open-ended questions.

Answers :

- ....
- ....
- ....