Learning Based Control HW2: Search and Optimization

Submitted By: Ashutosh Gupta
OSU ID: 934533517

I worked on this assignment on my own. All the work below along with the attached code is my own work. To clarify some doubts I worked with Raghav Thakar and Joshua Cook.

Contents

1 General Assumptions **2**

2 Simulated Annealing **2**
2.1 Algorithm . . . 2
2.2 Performance . . . 2

3 Evolutionary Search **3**
3.1 Algorithm . . . 3
3.2 Performance . . . 5

4 Stochastic Beam Search **5**
4.1 Algorithm . . . 5
4.2 Performance . . . 6

5 Discussion **6**

A Search algorithm code **8**
A.1 Base Search Algorithm . . . 8
A.2 Simulated Annealing . . . 12
A.3 Evolutionary Search . . . 15
A.4 Stochastic Beam Search . . . 18

B Experiments code **23**

List of Algorithms

1 Simulated Annealing . . . 3
2 Evolutionary Search . . . 4
3 Stochastic Beam Search . . . 7

1 General Assumptions

Some of the general assumptions and explanations about the algorithm implementations are mentioned below.

Solution Representation: Each solution in our representation is an ordered list of city indices in the order of visiting them. Each list will be of length 25 in the given problem, as that many cities exist in the problem. The solution does close the loop on the trip, travelling from the last city in the list to the starting city.

Cost Function: Cost function is simply the total euclidean distance of the path travelled. So each algorithm aims to minimize the total path length of a solution.

Initial Solution: An initial solution in each case is generated by a random shuffle of the city indices, this is to ensure no bias is added in the initial solution.

New Solution Generation: A new candidate solution is generated from the existing solution using a swapping operation. Where in we randomly select and swap 2 cities in the ordered list. This swapping operation can be performed any number of times on a given solution to generate a new solution.

Swapping cool down: The swapping operation, sometimes also referred to as the mutation operation helps generate new solutions from the existing one by searching around the current solution. This randomness is important in the start to better explore the search space as the initial solutions are not optimum. But as the algorithm progresses through the iterations, we would want to search more closer to the current good solutions for better convergence. Based on this logic each algorithm implements a cool down in the number of swaps to perform for generating a new solution. In each algorithm, the initial number of swaps are 10 and then after every $1/20$ of the iterations, number of swaps is reduced by 1.

2 Simulated Annealing

2.1 Algorithm

We start with an initial random solution generated by random shuffle. In each iteration we generate a candidate solution by the swap operation. If this solution has a better cost then our current solution, we accept it and move on to the next iteration. If the candidate solution has a worse cost, then we compute a probability using $e^{-\Delta E/T}$ and accept this bad solution with this probability. Otherwise continue with our current solution and move on to the next iteration. The temperature factor (T) is decayed every iteration by a decay factor. The parameters used in simulated annealing implementation are listed below:

- Iterations: $i = 5000$
- Temperature factor: $T = 15$
- Temperature decay factor: $\gamma = 0.9$

The algorithm in more detail is explained in Algorithm:1

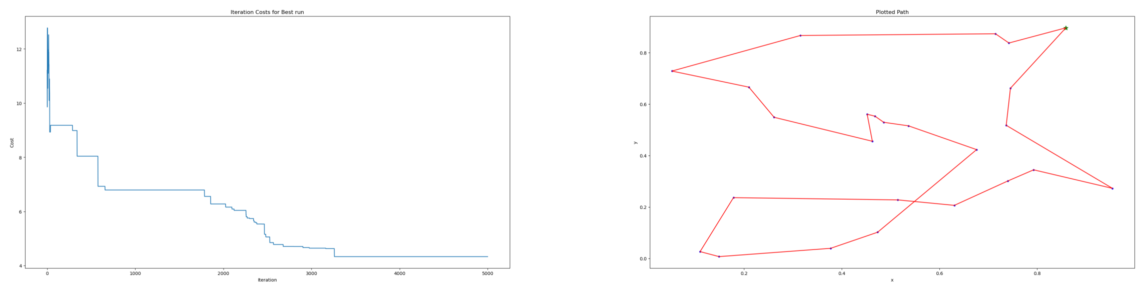
2.2 Performance

Simulated Annealing was run for 20 times to verify the repeatability of the algorithm. The overall best cost achieved was 4.3310. The mean of the best cost over the runs was 4.7528 with a standard deviation of 0.2762. Algorithm took on average 0.23 seconds to run.

Cost over iterations for the best run is shown in Fig:1a and the subsequent best solution path is plotted in Fig:1b. Cost over iterations for each run is shown in Fig:2.

Algorithm 1 Simulated Annealing

```
function SIMULATED_ANNEALING($i, T, \gamma$)
    $current = random\_shuffle$
    for i do
        $next = mutate(current)$
        if $cost(next) < cost(current)$ then
            $current \leftarrow next$
        else
            $p(current \leftarrow next) = e^{-(cost(next)-cost(current))/T}$    ▷ Accept the bad solution with a probability
        end if
        if $cost(current) < cost(best)$ then    ▷ Track the best solution
            $best \leftarrow current$
        end if
        $T \leftarrow \gamma * T$
    end for
    return $best$
end function
```



(a) Cost vs Iteration

(b) Path from the best solution (Start point is the star)

Figure 1: Best run from Simulated Annealing

3 Evolutionary Search

3.1 Algorithm

We start with an initial population of n generated by random shuffle. In each generation we first select the top k from the current generation based on their cost and mutate each of them to generate new k candidate solutions. Now from this current pool of $(n+k)$ solutions, we select the top n solutions based on their cost for the next generation. The parameters used in evolutionary search implementation are listed below:

- Generations: $i = 1000$
- Population size: $n = 100$
- Mutation size: $k = 90$

The algorithm in more detail is explained in Algorithm:2

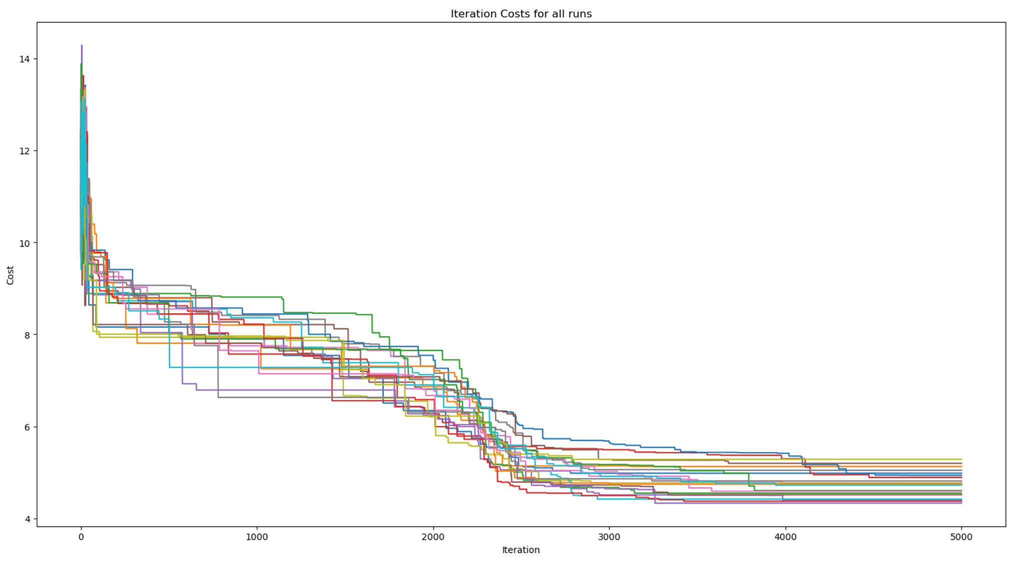


Figure 2: Cost vs Iteration for each run of Simulated Annealing

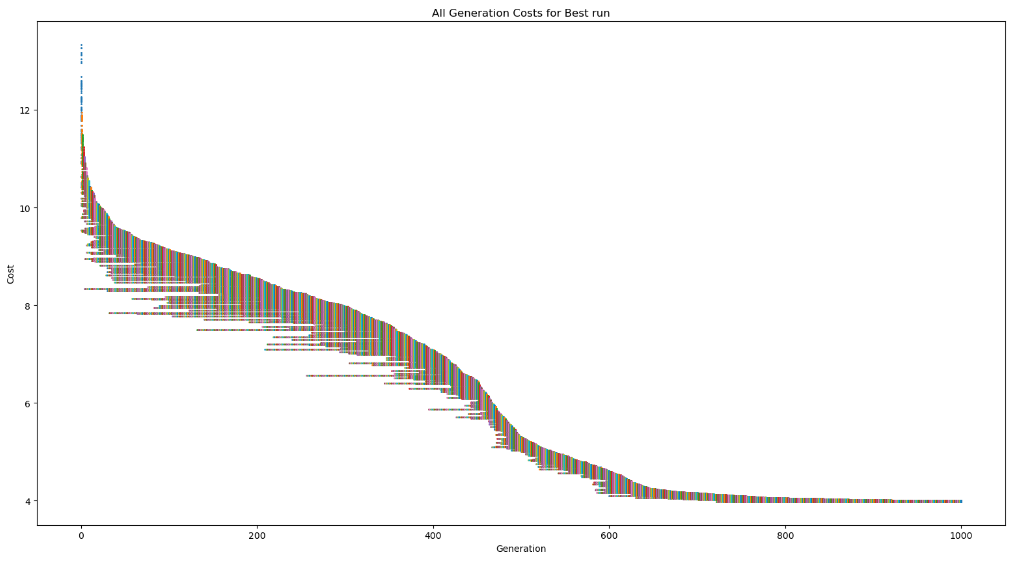
Algorithm 2 Evolutionary Search

```
function EVOLUTIONARY_SEARCH($i, n, k$)
    $current_n = random\_shuffle$                          ▷ Size of each list denoted in subscript
    for i do
        $to\_mutate_k = select\_top(current_n, k)$           ▷ Number of the solutions to select
        $next_k = mutate(to\_mutate_k)$
        $buffer_{n+k} = current_n + next_k$
        $current_n \leftarrow select\_top(buffer_{n+k}, n)$
        if $min(cost(current_n)) < best$ then             ▷ Track the best solution from all
            $best \leftarrow min(current_n)$
        end if
    end for
    return $best$
end function
```

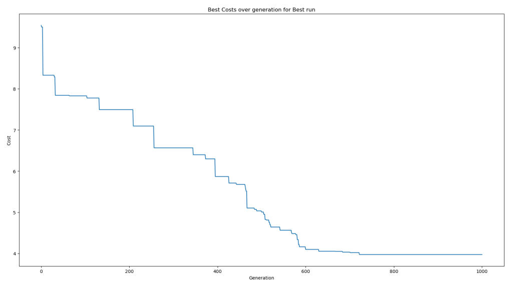
3.2 Performance

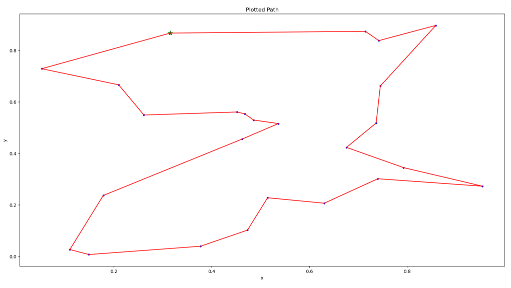
Evolutionary Search was run for 20 times to verify the repeatability of the algorithm. The overall best cost achieved was 3.9708. The mean of the best cost over the runs was 4.2461 with a standard deviation of 0.1688. Algorithm took on average 2.5 seconds to run.

Cost of all the agents in the population over all the generations for the best run is shown in Fig:3a, the best cost from each generation in the best run is shown in Fig:3b and the subsequent best solution path is plotted in Fig:3c. Best cost from each generation for all the runs is shown in Fig:4.



(a) Cost vs Generation for all agents





(b) Best Cost vs Generation

(c) Path from the best solution (Start point is the star)

Figure 3: Best run from Evolutionary Search

4 Stochastic Beam Search

4.1 Algorithm

We start with an initial population of n generated by random shuffle. In each iteration we mutate each of them to generate new n candidate solutions. Now from this current pool of $2n$ solutions, we need to select the next population of size n, this is done in a selection choice based on a stochastic factor λ. We first select a part of the next population as the top performing solutions from the current pool. Then the other part of the next population is randomly selected from the remaining current pool based on a probability depended on the cost of each agent. The stochastic factor λ, describes the fraction of n that is to be selected randomly. We start

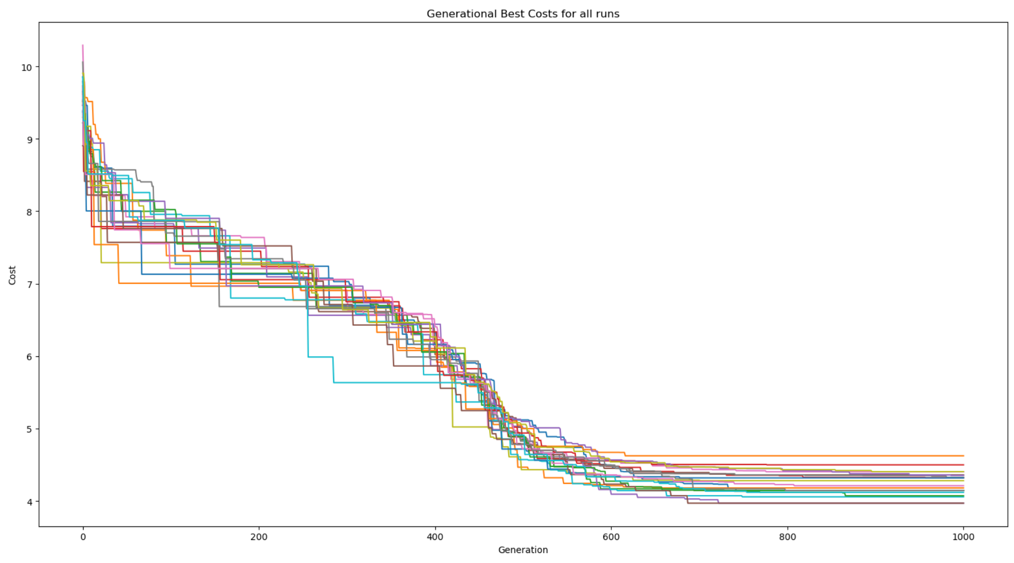


Figure 4: Best Cost vs Generation for each run of Evolutionary Search

with a higher stochastic factor λ, to encourage more exploration at the start and then reduce it by a decay factor γ every $1/100$ of total iterations. The parameters used in stochastic beam search implementation are listed below:

- Iterations: $i = 1000$
- Population/Beam size: $n = 25$
- Stochastic factor: $\lambda = 1.0$
- Decay factor: $\gamma = 0.9$

The algorithm in more detail is explained in Algorithm:3

4.2 Performance

Stochastic Beam Search was run for 30 times to verify the repeatability of the algorithm. The overall best cost achieved was 4.0582. The mean of the best cost over the runs was 4.4407 with a standard deviation of 0.2459. Algorithm took on average 0.92 seconds to run.

Cost of all the agents in the population over all the iterations for the best run is shown in Fig:5a, the best cost from each iteration in the best run is shown in Fig:5b and the subsequent best solution path is plotted in Fig:5c. Best cost from each iteration for all the runs is shown in Fig:6.

5 Discussion

Solution Space

- Total possible solutions for the travelling salesperson problem with 25 cities is $25! \approx 10^{25}$. Which is a very big solution search space.
- Simulated Annealing searches for just 1 solution in each iteration, so effectively it just searches 5000 solutions.

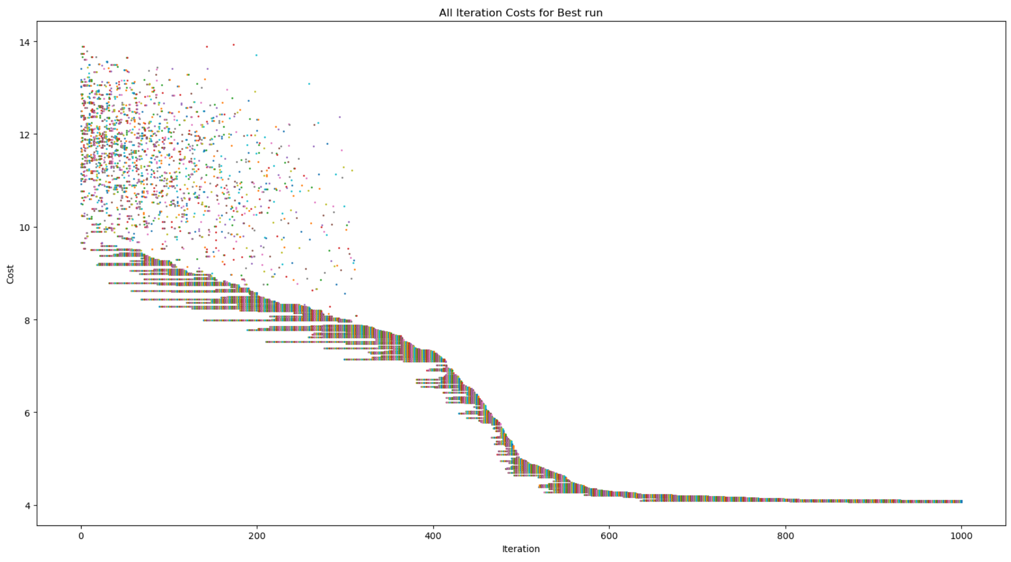
Algorithm 3 Stochastic Beam Search

```
function STOCHASTIC_BEAM_SEARCH($i, n, \lambda, \gamma$)
    $current_n = random\_shuffle$                         ▷ Size of each list denoted in subscript
    for i do
        $next_n = mutate(current_n)$
        $buffer_{2n} = current_n + next_n$
        $r = int(\lambda * n)$
        $t = n - num\_random$
        $top_t \leftarrow select\_top(buffer_{2n}, t)$            ▷ Number of the solutions to select
        $random_r \leftarrow select\_random((buffer_{2n} - top_t), r)$
        $current_n \leftarrow top_t + random_r$
        if $min(cost(current_n)) < best$ then              ▷ Track the best solution from all
            $best \leftarrow min(current_n)$
        end if
        Every $i/100 : \lambda \leftarrow \gamma * \lambda$
    end for
    return $best$
end function
```

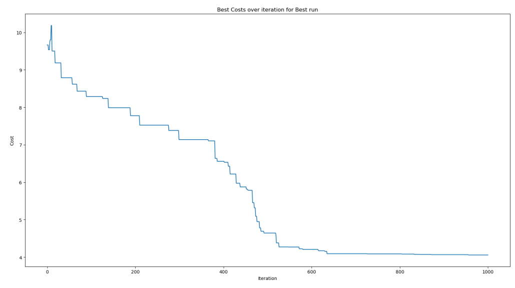
- Evolutionary algorithm searches for 90 new solutions in each generation, so it approximately searches 91000 solutions.
- Stochastic Beam algorithm searches for 25 new solutions in every iterations, so roughly searching through 25000 solutions.
- Each of the algorithm searches nearly close to a 0% of the solution space but still converge to a reasonably good solution.

Benefits and difficulties of the algorithm

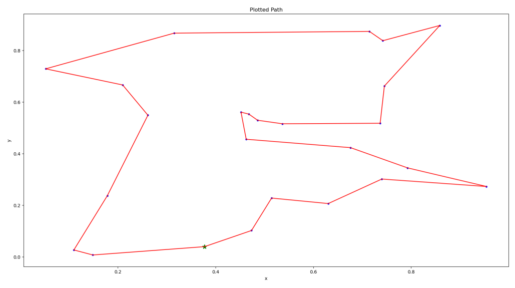
- Evolutionary and Stochastic beam being a population search algorithm searches through more of the solution space and converge to a better solution faster. Due to more search they don't require rather extensive parameter tuning, only the population size is tuned well.
- Simulated Annealing is a faster algorithm, but due to being a single solution search it doesn't converge all that well. There is lesser search coverage and the algorithm is more prone to get stuck in local minima. It requires more extensive parameter tuning of its Temperature factor. And it often results in more sub-optimal solutions.
- The algorithms don't always find the optimal solution as there is a lot of randomness injected in initial solution and subsequent solution generation. So they might not always converge to the same solution and result in some non-optimal or sub-optimal solution.



(a) Cost vs Iteration for all agents



(b) Best Cost vs Iteration



(c) Path from the best solution (Start point is the star)

Figure 5: Best run from Stochastic Beam Search

A Search algorithm code

A.1 Base Search Algorithm

```
import csv
import numpy as np
import matplotlib.pyplot as plt

class BaseSearchAlgorithm():
    """Base class for search algorithms with common functions"""

    def _generate_dist_matrix(self, csv_path:str):
        """Read the csv file and generate the distance matrix between cities

            Args:
            csv_path (str): path to the csv file
        """

        # Read the csv file
```

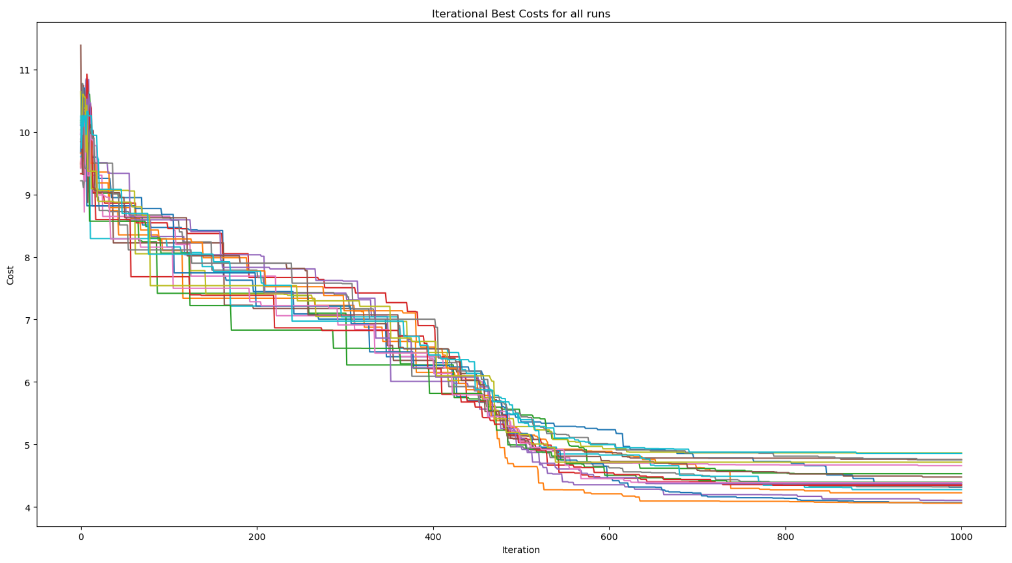


Figure 6: Best Cost vs Iteration for each run of Stochastic Beam Search

```
        with open(csv_path, 'r') as f:
            reader = csv.reader(f)
            data = list(reader)

        # Convert the data to numpy array
        self._data = np.array(data, dtype=np.float32)

        # Generate the distance matrix
        self.dist_matrix = np.zeros((self._data.shape[0], self._data.shape[0]))

        for i in range(self._data.shape[0]):
            for j in range(self._data.shape[0]):
                self.dist_matrix[i, j] = np.sqrt(np.sum((self._data[i] -
                ↪  self._data[j])**2))


    def _get_cost(self, path:list) -> float:
        """Compute the total distance of the path"""

        cost = 0.0

        # Add the distance between each city
        for i in range(len(path) - 1):
            cost += self.dist_matrix[path[i], path[i+1]]

        # Add the distance from the last city to the first city
        cost += self.dist_matrix[path[-1], path[0]]

        return cost
```

```
def _is_valid(self, path:list) -> bool:
    """Check if the path is valid
    Should visit all cities
    Have no duplicate cities"""

    # Check if the path has visited all cities
    if len(set(path)) != self.dist_matrix.shape[0]:
        return False

    # Check if the path has duplicate cities
    if len(path) != self.dist_matrix.shape[0]:
        return False

    return True

def _mutate_swap(self, path:list, num_swaps:int) -> list:
    """Mutate the path by performing swapping operation and return only valid
    ↪  path

    Args:
        path (list): path to be mutated
        num_mutations (int): number of swapping operations
    """

    # Check if the path is valid
    if not self._is_valid(path):
        print("Invalid path passed to mutate_path()")
        return path

    new_path = path.copy()

    # Perform swapping operation
    for _ in range(num_swaps):
        valid_swap = False
        while not valid_swap:
            # Randomly select two indices
            idx1 = np.random.randint(0, len(new_path))
            idx2 = np.random.randint(0, len(new_path))

            # If the two indices are the same, skip
            if idx1 == idx2:
                continue

            # Swap the two cities
            new_path[idx1], new_path[idx2] = new_path[idx2], new_path[idx1]
```

```
                valid_swap = True

        return new_path

    def _select_population(self, population:list, population_cost:list,
     ↪  num_select:int) -> (list, list, list):
        """Select the top population from the current population

            Args:
            population (list): population to select from
            population_cost (list): cost of the population
            num_select (int): number of agents to select

            Returns:
            select_population (list): selected population
            select_population_cost (list): cost of the selected population
            idx_chosen (list): index of the selected population from the original
             ↪  population
        """

        # Iniialize the selected population and cost list
        select_population = []
        select_population_cost = []
        idx_chosen = []

        # Get the top population
        order = np.argsort(population_cost)

        for i in range(num_select):
            select_population.append(population[order[i]])
            select_population_cost.append(population_cost[order[i]])
            idx_chosen.append(order[i])

        return (select_population, select_population_cost, idx_chosen)

    def _plot_path(self, path:list):
        """Plots the given path using the data points"""

        plt.figure("Path")
        plt.title("Plotted Path")

        # Plot the city points
        plt.scatter(self._data[:, 0], self._data[:, 1], s=10, c='b')

        # Only plot the path if it is valid
        if self._is_valid(path):
```

```
        # Plot the path
        plt.plot(self._data[path, 0], self._data[path, 1], c='r')
        # Complete the path circle
        plt.plot([self._data[path[-1], 0], self._data[path[0], 0]],
         ↪  [self._data[path[-1], 1], self._data[path[0], 1]], c='r')

        # Plot the starting point as green star
        plt.scatter(self._data[path[0], 0], self._data[path[0], 1], s=100,
         ↪  c='g', marker='*')

    plt.xlabel("x")
    plt.ylabel("y")
```

A.2 Simulated Annealing

```
import os
import numpy as np
import matplotlib.pyplot as plt

from base_search_algorithm import BaseSearchAlgorithm

class SimulatedAnnealing(BaseSearchAlgorithm):
    """Simulated Annealing algorithm for Traveling Salesman Problem"""

    def __init__(self, csv_path:str, iterations:int=1000, temperature:float=0.9,
     ↪  temperature_decay:float=0.9, num_swaps:int=10):
        """Initializes the Simulated annealing algorithm class

            Args:
            csv_path (str): path to the csv file containing the cities'
             ↪  coordinates
            iterations (int): maximum number of iterations
            temperature (float): initial temperature parameter for probability
             ↪  calculation
            temperature_decay (float): decay factor for temperature update
            num_swaps (int): number of swaps to perform to generate new path
        """

        # Generate the distance matrix
        self._generate_dist_matrix(csv_path)

        # Initialize the parameters
        self.iterations = iterations
        self.temperature = temperature
        self.temperature_decay = temperature_decay
        self.num_swaps = num_swaps
```

```python
        # Initialize the path
        self.path = np.arange(self.dist_matrix.shape[0])
        np.random.shuffle(self.path)
        self.path = list(self.path)

        # Initialize the cost
        self.cost = self._get_cost(self.path)

        # Initialize the best path and best cost
        self.best_path = self.path
        self.best_cost = self.cost

        # Initialize the cost history
        self.cost_history = [self.cost]

    def _get_probability(self, new_cost:float) -> float:
        """Calculate the probability of accepting the new path

            Args:
            new_cost (float): cost of the new path
        """

        # Calculate the probability
        probability = np.exp(-np.divide((new_cost - self.cost), self.temperature))

        return probability

    def algorithm(self, verbose:bool=True):
        """Simulated Annealing algorithm"""

        for i in range(self.iterations):
            # Generate the new path
            new_path = self._mutate_swap(self.path, self.num_swaps)

            # Calculate the new cost
            new_cost = self._get_cost(new_path)

            # Accept the new path if it has lower cost
            if new_cost < self.cost:
                self.path = new_path
                self.cost = new_cost

                # Update the best path and best cost
                if self.cost < self.best_cost:
                    self.best_path = self.path
                    self.best_cost = self.cost
            # Accept the new path if it has higher cost with a probability
            else:
                probability = self._get_probability(new_cost)

                # Select with a probability otherwise reject the new path and keep
                ↪  the old path
                if np.random.rand() < probability:
                    self.path = new_path
                    self.cost = new_cost

            # Update the temperature
            self.temperature *= self.temperature_decay

            # Update the cost history
            self.cost_history.append(self.cost)

            # Update number of swaps every 1/20 of the iterations
            if i % (self.iterations // 20) == 0 and i != 0 and self.num_swaps > 1:
                self.num_swaps -= 1

            # Print the progress
            if verbose:
                if i % (self.iterations // 10) == 0 or i == self.iterations - 1:
                    print(f"Iteration {i+1}/{self.iterations} - Cost:
                    ↪  {self.cost:.5f} - Temperature: {self.temperature}")

        if verbose:
            # Print the best path and best cost
            print(f"Best path: {self.best_path}")
            print(f"Best cost: {self.best_cost}")

            # Plot the best path
            self._plot_path(self.best_path)

    def plot_cost_history(self):
        """Plot the cost history of the algorithm"""

        plt.figure("Simulated Annealing")
        plt.plot(np.arange(self.iterations + 1), self.cost_history)
        plt.title("Cost History")
        plt.xlabel("Iteration")
        plt.ylabel("Path length")
        plt.show()
```

```
if __name__ == "__main__":

    # csv file path
    csv_path = os.path.join(os.getcwd(), "hw2.csv")

    # Parameters
    iterations = 5000
    temperature = 10.0
    temperature_decay = 0.9
    num_swaps = 10

    # Initialize the algorithm
    sa = SimulatedAnnealing(csv_path, iterations, temperature, temperature_decay,
     ↪  num_swaps)

    # Run the algorithm
    sa.algorithm()

    # Plot the cost history
    sa.plot_cost_history()
```

A.3 Evolutionary Search

```
import os
import numpy as np
import matplotlib.pyplot as plt

from base_search_algorithm import BaseSearchAlgorithm

class EvolutionarySearch(BaseSearchAlgorithm):
    """Evolutionary Search algorithm for Traveling Salesman Problem"""

    def __init__(self, csv_file:str, population_size:int=100, iterations:int=1000,
     ↪  num_swaps:int=10, mutation_size:int=25):
        """Initialize the evolutionary search algorithm class

            Args:
            csv_file (str): path to the csv file containing the cities'
             ↪  coordinates
            population_size (int): size of the population
            iterations (int): maximum number of iterations
            num_swaps (int): number of swaps to perform to generate new path
            mutation_size (int): number of paths to mutate
        """

        # Generate the distance matrix
        self._generate_dist_matrix(csv_file)
        # Initialize the parameters
        self.population_size = population_size
        self.iterations = iterations
        self.num_swaps = num_swaps
        self.mutation_size = mutation_size

        # Initialize the population
        self.population = []
        for i in range(self.population_size):
            path = np.arange(self.dist_matrix.shape[0])
            np.random.shuffle(path)
            self.population.append(list(path))

        # Initialize the cost history
        self.population_cost = []
        self.generational_cost = []

        # Get the cost of the initial population
        for path in self.population:
            self.population_cost.append(self._get_cost(path))

        # Store the initial generation cost
        self.generational_cost.append(self.population_cost)

        # Initialize the best path and best cost
        self.best_path = self.population[np.argmin(self.population_cost)]
        self.best_cost = np.min(self.population_cost)

        # Iniialize the top mutation_size population and cost
        self.top_population = []
        self.top_population_cost = []

    def algorithm(self, verbose:bool=True):
        """Evolutionary Search algorithm"""

        for i in range(self.iterations):
            # Select the top population from current population for mutation
            self.top_population, self.top_population_cost,_ =
             ↪  self._select_population(self.population, self.population_cost,
             ↪  self.mutation_size)

            # Mutate the top population
            mutated_population = []
            mutated_population_cost = []
            for path in self.top_population:
                new_path = self._mutate_swap(path, self.num_swaps)
                mutated_population.append(new_path)
                mutated_population_cost.append(self._get_cost(new_path))

            # Get the new population
            population_buffer = self.population + mutated_population
            population_cost_buffer = self.population_cost + mutated_population_cost

            # Select the next generation
            self.population, self.population_cost,_ =
             ↪  self._select_population(population_buffer, population_cost_buffer,
             ↪  self.population_size)

            # Store the cost history
            self.generational_cost.append(self.population_cost)

            # Update the best path and best cost
            if np.min(self.population_cost) < self.best_cost:
                self.best_path = self.population[np.argmin(self.population_cost)]
                self.best_cost = np.min(self.population_cost)

            # Reduce the swap number every 1/20 of the iterations
            if i % (self.iterations // 20) == 0 and i != 0 and self.num_swaps > 1:
                self.num_swaps -= 1

            # Print the progress
            if verbose:
                if i % (self.iterations // 10) == 0 or i == self.iterations - 1:
                    print(f"Iteration {i+1}/{self.iterations} - Cost:
                     ↪  {self.best_cost:.5f}")

        if verbose:
            # Print the best path and best cost
            print(f"Best path: {self.best_path}")
            print(f"Best cost: {self.best_cost}")

            # Plot the best path
            self._plot_path(self.best_path)

    def plot_cost_history(self):
        """Plot the cost history of the algorithm"""

        plt.figure("Evolutionary Search")

        # Plot cost of each member of the population in each generation
        plt.subplot(1, 2, 1)
        for i in range(len(self.generational_cost)):
            plt.scatter(np.repeat(i, self.population_size),
             ↪  self.generational_cost[i], s=1)
        plt.xlabel("Generation")
        plt.ylabel("Path length")
        plt.title("Cost of each member of the population")

        # Plot the best cost in each generation
        plt.subplot(1, 2, 2)
        plt.plot(np.arange(len(self.generational_cost)),
         ↪  np.min(self.generational_cost, axis=1))
        plt.xlabel("Generation")
        plt.ylabel("Path length")
        plt.title("Best cost in each generation")

        plt.show()


if __name__ == "__main__":

    # Get the path to the csv file
    csv_file = os.path.join(os.getcwd(), "hw2.csv")

    # Parameters
    population_size = 100
    iterations = 1000
    num_swaps = 10
    mutation_size = 90

    # Initialize the evolutionary search algorithm
    es = EvolutionarySearch(csv_file, population_size, iterations, num_swaps,
     ↪  mutation_size)

    # Run the algorithm
    es.algorithm()

    # Plot the cost history
    es.plot_cost_history()
```

A.4 Stochastic Beam Search

```
import os
import numpy as np
import matplotlib.pyplot as plt

from base_search_algorithm import BaseSearchAlgorithm
```

```
class StochasticBeamSearch(BaseSearchAlgorithm):
    """Stochastic Beam Search algorithm for Traveling Salesman Problem"""

    def __init__(self, csv_file:str, beam_width:int=25, iterations:int=1000,
    ↪  num_swaps:int=10, stochastic_factor:float=0.75, cooling_rate:float=0.9):
        """Initializes Stochastic Beam Search algorithm

            Args:
            csv_file (str): path to the csv file containing the cities'
            ↪  coordinates
            beam_width (int): beam width
            iterations (int): maximum number of iterations
            num_swaps (int): number of swaps to perform to generate new path
            stochastic_factor (float): factor to determine the number of paths to
            ↪  select randomly from the population with probability
            cooling_rate (float): cooling rate for the stochastic factor
        """

        # Generate the distance matrix
        self._generate_dist_matrix(csv_file)

        # Initialize the parameters
        self.beam_width = beam_width
        self.iterations = iterations
        self.num_swaps = num_swaps
        self.stochastic_factor = stochastic_factor
        self.cooling_rate = cooling_rate

        # Initalize the population
        self.population = []
        for i in range(self.beam_width):
            path = np.arange(self.dist_matrix.shape[0])
            np.random.shuffle(path)
            self.population.append(list(path))

        # Initialize the cost history
        self.population_cost = []
        self.iterational_cost = []

        # Get the cost of the initial population
        for path in self.population:
            self.population_cost.append(self._get_cost(path))

        # Store the initial population cost
        self.iterational_cost.append(self.population_cost)
        # Initialize the best path and best cost
        self.best_path = self.population[np.argmin(self.population_cost)]
        self.best_cost = np.min(self.population_cost)

    def _get_probabilities(self, population_cost:list) -> list:
        """Returns the probabilities of each member of the population

            Args:
            population_cost (list): cost of each member of the population

            Returns:
            list: probabilities of each member of the population
        """

        # Get the probabilities
        probabilities = []
        for cost in population_cost:
            probabilities.append(1 / cost)

        # Normalize the probabilities
        probabilities = probabilities / np.sum(probabilities)

        return probabilities

    def algorithm(self, verbose:bool=True):
        """Stochastic Beam Search algorithm"""

        for i in range(self.iterations):
            # Initialize the new population and new population cost
            new_population = []
            new_population_cost = []

            # Generate the new population
            for path in self.population:
                new_path = self._mutate_swap(path, self.num_swaps)
                new_population.append(new_path)
                new_population_cost.append(self._get_cost(new_path))

            # Get the new population
            population_buffer = self.population + new_population
            population_cost_buffer = self.population_cost + new_population_cost

            # Select the population from the buffer population depending on the
             ↪  stochastic factor
            num_random = int(self.stochastic_factor * self.beam_width)
            num_top = self.beam_width - num_random
            # Get the top population
            self.population, self.population_cost, idx_top =
                ↪  self._select_population(population_buffer, population_cost_buffer,
                ↪  num_top)

            # Remove the top population from the buffer population
            population_buffer = [population_buffer[idx] for idx in
             ↪  range(len(population_buffer)) if idx not in idx_top]
            population_cost_buffer = [population_cost_buffer[idx] for idx in
             ↪  range(len(population_cost_buffer)) if idx not in idx_top]

            # Get the random population
            probabilities = self._get_probabilities(population_cost_buffer)
            idx_random = np.random.choice(a=len(population_buffer),
             ↪  size=num_random, replace=False, p=probabilities)
            self.population += [population_buffer[idx] for idx in idx_random]
            self.population_cost += [population_cost_buffer[idx] for idx in
             ↪  idx_random]

            # Store the iteration cost
            self.iterational_cost.append(self.population_cost)

            # Update the best path and best cost
            if np.min(self.population_cost) < self.best_cost:
                self.best_path = self.population[np.argmin(self.population_cost)]
                self.best_cost = np.min(self.population_cost)

            # Update the stochastic factor every 1/100 of the iterations
            if i % (self.iterations // 100) == 0 and i != 0:
                self.stochastic_factor *= self.cooling_rate

            # Reduce the number of swaps for every 1/20th of the iterations
            if i % (self.iterations // 20) == 0 and i != 0 and self.num_swaps > 1:
                self.num_swaps -= 1

            # Print the progress
            if verbose:
                if i % (self.iterations // 10) == 0 or i == self.iterations - 1:
                    print(f"Iteration {i+1}/{self.iterations} - Cost:
                     ↪  {self.best_cost:.5f}")

        if verbose:
            # Print the best path and best cost
            print(f"Best path: {self.best_path}")
            print(f"Best cost: {self.best_cost}")
```

```
            # Plot the best path
            self._plot_path(self.best_path)

    def plot_cost_history(self):
        """Plots the cost history of the algorithm"""

        plt.figure("Beam Search")

        # Plot cost of each member of the population in each iteration
        plt.subplot(1, 2, 1)
        for i in range(len(self.iterational_cost)):
            plt.scatter(np.repeat(i, self.beam_width), self.iterational_cost[i],
             ↪  s=1)

        plt.xlabel("Iteration")
        plt.ylabel("Path length")
        plt.title("Cost of each member of the population in each iteration")

        # Plot the best cost in each iteration
        plt.subplot(1, 2, 2)
        plt.plot(np.arange(len(self.iterational_cost)),
         ↪  np.min(self.iterational_cost, axis=1))
        plt.xlabel("Iteration")
        plt.ylabel("Path length")
        plt.title("Best cost in each iteration")

        plt.show()


if __name__ == "__main__":

    # Get the path to the csv file
    csv_file = os.path.join(os.getcwd(), "hw2.csv")

    # Parameters
    beam_width = 25
    iterations = 1000
    num_swaps = 10
    stochastic_factor = 1.0
    cooling_rate = 0.9

    # Initialize the beam search algorithm
    sbs = StochasticBeamSearch(csv_file, beam_width, iterations, num_swaps,
     ↪  stochastic_factor, cooling_rate)
```

```
# Run the algorithm
sbs.algorithm()

# Plot the cost history
sbs.plot_cost_history()
```

B Experiments code

The pdf generated from the Jupyter notebook is attached below.

experiments

October 24, 2023

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

from simulated_annealing import SimulatedAnnealing
from evolutionary_search import EvolutionarySearch
from stochastic_beam_search import StochasticBeamSearch

%matplotlib qt
```

[2]:
```
# csv file path
csv_path = os.path.join(os.getcwd(), "hw2.csv")
```

[20]:
```
## Simulated Annealing experiments
# Parameters
iterations = 5000
temperature = 15.0
temperature_decay = 0.9
num_swaps = 10
runs = 20

# Results
best_paths = []
best_cost = []
iteration_costs = []
time_to_run = []

# Run simulated annealing for number of runs
for i in range(runs):
    # Initialize simulated annealing
    sa = SimulatedAnnealing(csv_path, iterations, temperature,␣
 ↪temperature_decay, num_swaps)

    # Run simulated annealing
    start = time.time()
    sa.algorithm(verbose=False)
    end = time.time()

    # Save results
    best_paths.append(sa.best_path)
    best_cost.append(sa.best_cost)
    iteration_costs.append(sa.cost_history)
    time_to_run.append(end - start)
```

[35]:
```
# Find best run
best_run_idx = np.argmin(best_cost)

# Plot iteration costs for best run
plt.figure("Iteration costs for best run")
plt.plot(range(len(iteration_costs[best_run_idx])),␣
 ↪iteration_costs[best_run_idx])
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Iteration Costs for Best run")

# Plot best path for best run
sa._plot_path(best_paths[best_run_idx])

# Plot best cost for each run
plt.figure("Best cost for each run")
plt.scatter(range(1, runs+1), best_cost, s=20)
plt.xticks(range(1, runs+1))
plt.xlabel("Run")
plt.ylabel("Best Cost")
plt.title("Best Cost for each run")

# Plot iteration costs for each run
plt.figure("Iteration costs for each run")
for i in range(runs):
    plt.plot(range(len(iteration_costs[i])), iteration_costs[i])

plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Iteration Costs for all runs")

plt.show()
```

[38]:
```
print("For Simulated Annealing:")

# Print the time to run for each run
print(f"Time taken for each run in sec: {time_to_run}")

# Print the best cost achieved
```

```
print(f"Best cost achieved: {best_cost[best_run_idx]}")

# Print the mean and standard deviation of the best cost
print(f"Mean best cost: {np.mean(best_cost)}")
print(f"Standard deviation of best cost: {np.std(best_cost)}")
```

```
For Simulated Annealing:
Time taken for each run in sec: [0.27204298973083496, 0.22841525077819824,
0.22993993759155273, 0.2314624786376953, 0.22998046875, 0.22643208503723145,
0.22549843788146973, 0.2241065502166748, 0.22742605209350586,
0.22686243057250977, 0.22619199752807617, 0.22486424446105957,
0.22822117805480957, 0.2328319549560547, 0.2298872470855713,
0.23218560218811035, 0.22817301750183105, 0.2301175594329834,
0.2245655059814453, 0.22311997413635254]
Best cost achieved: 4.331013482064009
Mean best cost: 4.7527596110478045
Standard deviation of best cost: 0.2761799610804234
```

[40]:
```
## Evolutionary Search experiments
# Parameters
population_size = 100
iterations = 1000
num_swaps = 10
mutation_size = 90
runs = 20

# Results
best_paths = []
best_cost = []
iteration_costs = []
time_to_run = []

# Run evolutionary search for number of runs
for i in range(runs):
    # Initialize evolutionary search
    es = EvolutionarySearch(csv_path, population_size, iterations, num_swaps,␣
 ↪mutation_size)

    # Run evolutionary search
    start = time.time()
    es.algorithm(verbose=False)
    end = time.time()

    # Save results
    best_paths.append(es.best_path)
    best_cost.append(es.best_cost)
    iteration_costs.append(es.generational_cost)
    time_to_run.append(end - start)
```

[42]:
```
# Find best run
best_run_idx = np.argmin(best_cost)

# Plot generation costs for best run
plt.figure("Generation costs for best run")
for i in range(len(iteration_costs[best_run_idx])):
    plt.scatter(np.repeat(i, population_size),␣
  ↪iteration_costs[best_run_idx][i], s=1)
plt.xlabel("Generation")
plt.ylabel("Cost")
plt.title("All Generation Costs for Best run")

# Plot best costs in each generation for the best run
plt.figure("Best costs for best run")
plt.plot(range(len(iteration_costs[best_run_idx])), np.
  ↪min(iteration_costs[best_run_idx], axis=1))
plt.xlabel("Generation")
plt.ylabel("Cost")
plt.title("Best Costs over generation for Best run")

# Plot best path for best run
es._plot_path(best_paths[best_run_idx])

# Plot best cost for each run
plt.figure("Best cost for each run")
plt.scatter(range(1, runs+1), best_cost, s=20)
plt.xticks(range(1, runs+1))
plt.xlabel("Run")
plt.ylabel("Best Cost")
plt.title("Best Cost for each run")

# Plot generational best costs for each run
plt.figure("Generational best costs for each run")
for i in range(runs):
    plt.plot(range(len(iteration_costs[i])), np.min(iteration_costs[i], axis=1))
plt.xlabel("Generation")
plt.ylabel("Cost")
plt.title("Generational Best Costs for all runs")

plt.show()
```

[43]:
```
print("For Evolutionary Search:")

# Print the time to run for each run
print(f"Time taken for each run in sec: {time_to_run}")
```

```
# Print the best cost achieved
print(f"Best cost achieved: {best_cost[best_run_idx]}")

# Print the mean and standard deviation of the best cost
print(f"Mean best cost: {np.mean(best_cost)}")
print(f"Standard deviation of best cost: {np.std(best_cost)}")
```

```
For Evolutionary Search:
Time taken for each run in sec: [3.8853065967559814, 3.604876756668091,
2.3146116733551025, 2.3303232192993164, 2.3131256103515625, 2.3504374027252197,
2.3750338554382324, 2.2965402603149414, 2.3313534259796143, 2.3383419513702393,
2.3685531616210938, 2.3826637268066406, 2.356966972351074, 2.4064712524414062,
2.37369441986084, 2.38724422454834, 2.372828960418701, 2.3999781608581543,
2.3677496910095215, 2.386867046356201]
Best cost achieved: 3.9708135314285755
Mean best cost: 4.2460881766863166
Standard deviation of best cost: 0.16885564554528337
```

[45]:
```
## Stochastic Beam Search experiments
# Parameters
beam_width = 25
iterations = 1000
num_swaps = 10
stochastic_factor = 1.0
cooling_rate = 0.9
runs = 20

# Results
best_paths = []
best_cost = []
iteration_costs = []
time_to_run = []

# Run evolutionary search for number of runs
for i in range(runs):
    # Initialize Stochastic Beam Search
    sbs = StochasticBeamSearch(csv_path, beam_width, iterations, num_swaps,␣
 ↪stochastic_factor, cooling_rate)

    # Run evolutionary search
    start = time.time()
    sbs.algorithm(verbose=False)
    end = time.time()

    # Save results
    best_paths.append(sbs.best_path)
    best_cost.append(sbs.best_cost)
    iteration_costs.append(sbs.iterational_cost)
    time_to_run.append(end - start)
```

[46]:
```
# Find best run
best_run_idx = np.argmin(best_cost)

# Plot iteration costs for best run
plt.figure("Iteration costs for best run")
for i in range(len(iteration_costs[best_run_idx])):
    plt.scatter(np.repeat(i, beam_width), iteration_costs[best_run_idx][i], s=1)
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("All Iteration Costs for Best run")

# Plot best costs in each iteration for the best run
plt.figure("Best costs for best run")
plt.plot(range(len(iteration_costs[best_run_idx])), np.
  ↪min(iteration_costs[best_run_idx], axis=1))
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Best Costs over iteration for Best run")

# Plot best path for best run
sbs._plot_path(best_paths[best_run_idx])

# Plot best cost for each run
plt.figure("Best cost for each run")
plt.scatter(range(1, runs+1), best_cost, s=20)
plt.xticks(range(1, runs+1))
plt.xlabel("Run")
plt.ylabel("Best Cost")
plt.title("Best Cost for each run")

# Plot iterational best costs for each run
plt.figure("Iterational best costs for each run")
for i in range(runs):
    plt.plot(range(len(iteration_costs[i])), np.min(iteration_costs[i], axis=1))
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.title("Iterational Best Costs for all runs")

plt.show()
```

[47]:
```
print("For Stochastic Beam Search:")

# Print the time to run for each run
```

```python
print(f"Time taken for each run in sec: {time_to_run}")

# Print the best cost achieved
print(f"Best cost achieved: {best_cost[best_run_idx]}")

# Print the mean and standard deviation of the best cost
print(f"Mean best cost: {np.mean(best_cost)}")
print(f"Standard deviation of best cost: {np.std(best_cost)}")
```

```
For Stochastic Beam Search:
Time taken for each run in sec: [1.2976090908050537, 1.2390804290771484,
1.2067830562591553, 1.2253077030181885, 1.2317626476287842, 1.1565864086151123,
0.8249421119689941, 0.8102307319641113, 0.7752177715301514, 0.759474515914917,
0.7691755294799805, 0.7858595848083496, 0.7851295471191406, 0.7573051452636719,
0.7707412242889404, 0.7716658115386963, 0.8011846542358398, 0.7934019565582275,
0.7988109588623047, 0.787672758102417]
Best cost achieved: 4.058206062763929
Mean best cost: 4.440742810815573
Standard deviation of best cost: 0.24586903082127312
```