****

**T.C.**

**ESKİŞEHİR OSMANGAZİ UNIVERSITY**

**Faculty of Engineering and Architecture**

**Department of Computer Engineering**

**2022-2023 Fall Semester**

**152117127 - Introduction to Heuristic Algorithms**

**The Traveling Salesman Problem**

**using**

**Combinatorial Optimization**

**Prepared by**

Hakam CHEDO – 152120181096

**Course Instructor**

Asst. Prof. Dr. Emrah ATILGAN

January 2023

**Contents**

[**INTRODUCTION** 2](#_Toc123899665)

[**BRUTE-FORCE SEARCH** 2](#_Toc123899666)

[**GENETIC ALGORITHM** 4](#_Toc123899667)

[**HILL CLIMBING ALGORITHM** 5](#_Toc123899668)

# **INTRODUCTION**

The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" The traveling salesman problems abide by a salesman and a set of cities. The salesman must visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

The most direct solution would be to try all [permutations](https://en.wikipedia.org/wiki/Permutation) (ordered combinations) and see which one is cheapest (using [brute-force search](https://en.wikipedia.org/wiki/Brute-force_search)). The running time for this approach lies within a polynomial factor of {\displaystyle O(n!)}Օ(n!), the [factorial](https://en.wikipedia.org/wiki/Factorial) of the number of cities, so this solution becomes impractical even for only 20 cities.

In that case, problems can be solved exactly using Branch and Bound techniques. However, in other cases no exact algorithms are feasible, and randomized search algorithms must be employed, such as:

* Genetic Algorithm
* Simulated Annealing Algorithm
* Hill Climbing Algorithm
* Tabu Search Algorithm
* Ant Colony Optimization

These five algorithms will be used to solve traveling salesman problem in this project.

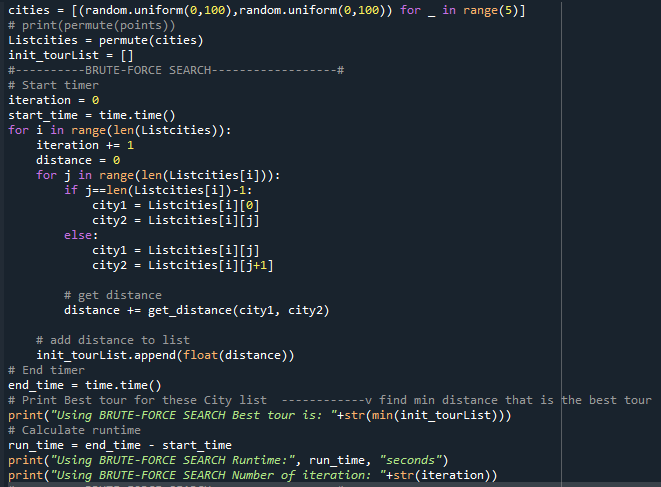
# [**BRUTE-FORCE SEARCH**](https://en.wikipedia.org/wiki/Brute-force_search)

Brute-force search, also known as exhaustive search, is a method of solving a problem by trying all possible solutions and selecting the best one. In the case of the Traveling Salesman Problem (TSP), a brute-force search would involve checking all possible routes and selecting the shortest one.

The brute-force approach is not practical for solving large instances of the TSP, as the number of possible routes grows exponentially with the number of cities. For example, there are approximately 20! (20 factorial) possible routes for a TSP with 20 cities. This is a very large number, and it would take an impractical amount of time to check all of them.

However, the brute-force approach can be useful for solving small instances of the TSP and as a baseline for comparing the performance of other algorithms. It is also sometimes used as a subroutine in more complex algorithms for the TSP.

This is the brute-force python code that takes a list of randomly city location (represented as tuples of x and y coordinates) and display the shortest route through all the points, as well as its runtime and number of iterations. The permute function is a helper function that to permutate all possible way to ordered combinations. Also, the get\_distance function is a helper function that calculates the distance between two cities.

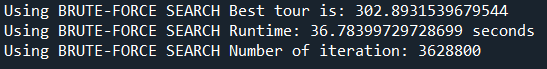


This is the result for 5 cities. It doesn’t take much time, and the number of iterations is just 120

Text

Description automatically generated

But with the 10 number of cities, it does an iteration 10! time that is 3,628,800 time and the runtime is increase very much from the example before.



In summary brute-force approach will work for small problem sizes, but it becomes impractical for larger sets of points because the number of possible permutations grows exponentially with the number of points. For instance, with just 10 points there are already over 3 million permutations to consider.

# **GENETIC ALGORITHM**

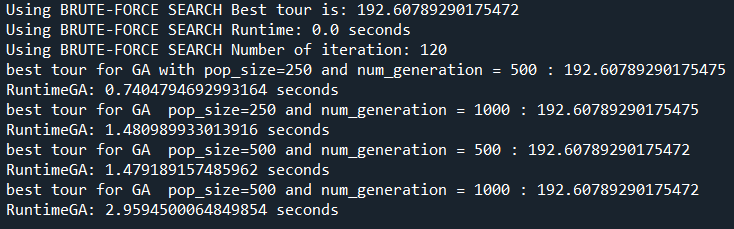
Genetic algorithms are heuristic optimization algorithms that are inspired by the principles of natural evolution, such as reproduction, mutation, and selection. They can be used to find good, but not necessarily optimal, solutions to the Traveling Salesman Problem (TSP).

To solve the TSP using a genetic algorithm, the following steps can be taken:

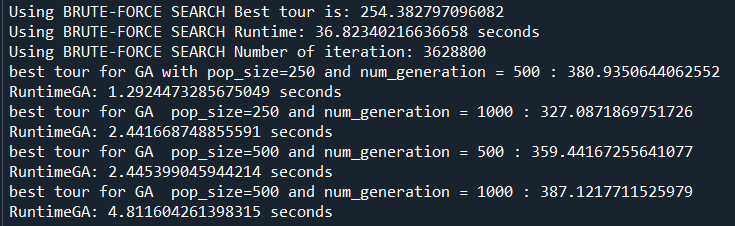
* Initialize the population: Generate a set of initial solutions to the problem, called "individuals," to serve as the starting point for the search.
* Evaluate the fitness of each individual: Calculate a measure of the quality of each solution in the population. This is typically done by calculating the total distance of the route represented by the solution.
* Select the best individuals: Choose a subset of the most fit individuals to serve as the parents for the next generation.
* Generate offspring: Use the selected parents to produce a new set of individuals through processes such as crossover (recombining pieces of the parents' solutions) and mutation (randomly altering parts of the solutions).
* Repeat the process: Evaluate the fitness of the new individuals, select the best ones, and generate offspring from them. This process is repeated until a satisfactory solution is found or a predetermined termination condition is reached.

The genetic algorithm python code implementation is in GA.py attachment file.

Result of 5 cities compare to the Brute force search is we get the exact values to the Brute force search, but it takes a little time due to the number of generation or iteration



Result of 10 cities compare to the Brute force search is we get values that are close to Brute force search, but in much less time.



As a result, Genetic algorithms can be effective for solving small to medium sized instances of the Travelling Salesman Problem, but they may not scale well to larger instances due to the computational cost of generating and evaluating large populations of solutions.

# **SIMULATED ANNEALING ALGORITHM**

Simulated annealing is an optimization algorithm that is inspired by the annealing process in metallurgy. It can be used to find good, but not necessarily optimal, solutions to the Traveling Salesman Problem (TSP).

To solve the TSP using simulated annealing, the following steps can be taken:

* Initialize the current solution: Choose a starting point for the search, such as a randomly generated solution or a simple solution such as visiting the cities in order of their indices.
* Generate a new solution: Create a new solution by making a random change to the current solution. For example, the new solution could be obtained by swapping the positions of two cities in the tour.
* Accept or reject the new solution: Calculate the difference in quality between the new solution and the current solution. If the new solution is better, it is accepted as the current solution. If it is worse, it may still be accepted with a certain probability, based on a cooling schedule.
* Repeat the process: Generate a new solution and accept or reject it. The probability of accepting worse solutions is decreased over time according to the cooling schedule, which helps the algorithm escape from local minima and explore the search space more extensively.

The simulated annealing algorithm python code implementation is in SA.py attachment file.

Result of 5 cities compare to the Brute force search is we get the exact values to the Brute force search, and the runtime is as same as using the Brute force search.

Text

Description automatically generated

Result of 10 cities compare to the Brute force search is we get values that are close to Brute force search, but in much less time.

Text

Description automatically generated

In summary, Simulated annealing can be effective for solving small to medium sized instances of the TSP, but it can be computationally expensive if the quality of each new solution must be calculated. It can also be sensitive to the choice of cooling schedule.

# **HILL CLIMBING ALGORITHM**

Hill climbing is a heuristic optimization algorithm that can be used to find good, but not necessarily optimal, solutions to the Traveling Salesman Problem (TSP). It works by starting with an initial solution and iteratively improving it by making small changes and selecting the best solution at each step.

To solve the TSP using a hill climbing algorithm, the following steps can be taken:

* Initialize the current solution: Choose a starting point for the search, such as a randomly generated solution or a simple solution such as visiting the cities in order of their indices.
* Generate a set of neighboring solutions: Create a set of solutions that are similar to the current solution, but with one or more small changes. For example, the neighboring solutions of a solution that visits the cities in order might be obtained by swapping the positions of two cities in the tour.
* Select the best neighbor: Evaluate the quality of each neighboring solution and select the one with the lowest total distance.
* Repeat the process: Set the current solution to the best neighbor and generate a new set of neighbors. If a better solution is found, the process is repeated. Otherwise, the search terminates, and the current solution is returned as the result.

The Hill climbing algorithm python code implementation is in HC.py attachment file.

Result of 5 cities compare to the Brute force search is we get the exact values to the Brute force search, and the runtime is as same as using the Brute force search.

Text

Description automatically generated

Result of 10 cities compare to the Brute force search is we get values that are close to Brute force search, but in much less time.

Text

Description automatically generated

In concluded, Hill climbing can be effective for solving small to medium sized instances of the TSP, but it may get stuck in local minima and fail to find the global optimum for the problem. It can also be computationally expensive if the set of neighbors must be generated and evaluated at each step of the search.

# **TABU SEARCH ALGORITHM**

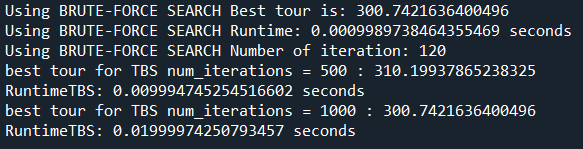
Tabu search is a metaheuristic optimization algorithm that uses a memory of past solutions to guide the search for a good solution to the Traveling Salesman Problem (TSP). It avoids revisiting solutions that were previously found to be poor.

To solve the TSP using tabu search, the following steps can be taken:

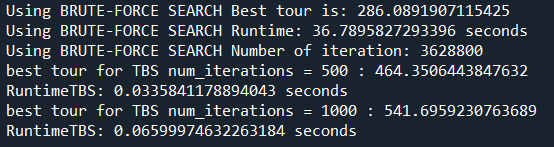
* Initialize the current solution: Choose a starting point for the search, such as a randomly generated solution or a simple solution such as visiting the cities in order of their indices.
* Generate a set of neighbors: Create a set of solutions that are similar to the current solution, but with one or more small changes. For example, the neighbors of a solution that visits the cities in order might be obtained by swapping the positions of two cities in the tour.
* Select the best neighbor: Evaluate the quality of each neighbor and select the one with the lowest total distance. If a neighbor has been visited recently, it is considered "tabu" and is not selected.
* Update the tabu list: Add the current solution to the tabu list, which is a memory of past solutions that are not allowed to be revisited.
* Repeat the process: Set the current solution to the best neighbor and generate a new set of neighbors. If a better solution is found, the process is repeated. Otherwise, the search terminates, and the current solution is returned as the result.

The Tabu search algorithm python code implementation is in TBS.py attachment file.

Result of 5 cities compare to the Brute force search is we get the values that are close to Brute force search, and the runtime is as same as using the Brute force search.



Result of 10 cities compare to the Brute force search is we get values that are not close to Brute force search, but it uses much less time.



In summary, the tabu search algorithm can’t be effective for solving medium to large size instanced of the TSP

# **ANT COLONY OPTIMIZATION**

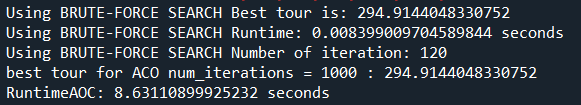
Ant Colony Optimization (ACO) is a metaheuristic optimization algorithm that is inspired by the foraging behavior of ant colonies. It can be used to find good, but not necessarily optimal, solutions to the Traveling Salesman Problem (TSP).

To solve the TSP using ACO, the following steps can be taken:

* Initialize the population of ants: Generate a set of "ants" that will search for solutions to the problem. Each ant is associated with a tour of the cities, which it will construct by moving from city to city.
* Place the ants on the cities: Assign each ant to a starting city for its tour.
* Allow the ants to construct tours: At each step, each ant selects the next city to visit based on the quality of the neighboring cities and a probabilistic rule that takes into account the amount of pheromone that has been deposited on the edges between the cities. Pheromone is a chemical substance that is used by ants to communicate and coordinate their behavior.
* Evaluate the tours: Calculate the total distance of each tour and use it as a measure of the quality of the solution.
* Update the pheromone levels: Increase the pheromone levels on the edges that are part of good tours and decrease the pheromone levels on the edges that are part of poor tours. This allows the ants to "remember" and exploit good solutions while avoiding poor ones.
* Repeat the process: Place the ants on the cities again and allow them to construct new tours. The process is repeated until a satisfactory solution is found or a predetermined termination condition is reached.

The Ant colony optimization python code implementation is in ACO.py attachment file.

Result of 5 cities compare to the Brute force search is we get the exact values to Brute force search, but the runtime is much more than using the Brute force search.



Text

Description automatically generated

Result of 10 cities compare to the Brute force search is we get values that are not close to Brute force search, but it uses less time.

As a result, ACO can be effective for solving small to medium sized instances of the TSP, but it can be computationally expensive if the tours of all the ants must be constructed and evaluated at each iteration. It can also be sensitive to the parameters of the algorithm, such as the evaporation rate of the pheromone.

# **SUMMARY**

In summary, the combinatorial optimization can reduce the time from using brute force search as much as possible but the values that we get is close to the actual value for some algorithm.