TRAVELING THIEF PROBLEM USING GENETIC PROGRAMMING

Under guidance of Dr Dinesh Reddy Vemula, *Assistant Professor, Department of Computer Science Engineering*, SRM UNIVERSITY AP [dineshreddy.v@srmap.edu.in](mailto:dineshreddy.v@srmap.edu.in)

Sellamuthu Abishek Bobba Nikhila Dhanush Kamal Ede Sighakolli Chinmaya Datta

*Department of CSE* *Department of CSE Department of CSE Department of CSE*

SRM UNIVERSITY AP SRM UNIVERSITY AP SRM UNIVERSITY AP SRM UNIVERSITY AP

[abishek\_sellamuthu@srmap.edu.in](mailto:abishek_sellamuthu@srmap.edu.in) [nikhila\_bobba@srmap.edu.in](mailto:nikhila_bobba@srmap.edu.in) [dhanushkamal\_ede@srmap.edu.in](mailto:dhanushkamal_ede@srmap.edu.in) [Chinmaya\_malleswara@srmap.edu.in](mailto:Chinmaya_malleswara@srmap.edu.in)

**Abstract**

*Many real-world applications require dealing with a large problem with many Subproblems with interrelated subproblems. In order to study the interdependence between subproblems, the traveling thief problem (TTP) was set as a standard problem. It is a combination of the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP), two famous subproblems. Although each subproblem has been intensively studied, the related combinations have proven to be difficult and impossible to solve by solving subproblems separately.*

*The TSP and the (KP) is an efficient approach that has better scalability and solution quality, including steps for improvement and item selection.*

**Key Words:**

*Travel thief problem, genetic programming, Order 1 crossover, Swap mutation, Roulette Wheel Technique, Real-world optimization problems.*

1. **INTRODUCTION:**

**1.1 Genetic algorithm (GA):**

It is a method of intelligent computation. It is a research technique used to find approximate solutions to combinable optimization problems in computer science. Genetic algorithm is more properly considered an optimization technique based on natural evolution. best-fit idea algorithm. The approach here is to first guess the possible solutions and then to generate new solutions by combining similar solutions and make it better than the previous one. To prevent any crashes or malfunction we include a random mutation factor. Following components are used during the compilation process:

1 *Encoding:* 1. Encoding: For our problem, a suitable encoding is determined so that each conceivable solution has a unique encoding, which is some type of a code string.

2. Evaluation: The initial population is then chosen, usually at random, though heuristic-based strategies have also been proposed. The fitness of each individual in the population is then calculated, i.e., how well the individual fits the problem and whether it is close to the optimum output in comparison to the other individuals in the population.

3. Crossover: The fitness is utilized to determine an individual's likelihood of crossover. When two individuals are recombined to form new individuals, they are replicated into the next generation.

4. Mutation:  The next mutation takes place. Some people are picked at random to be mutated, and then a mutation point is chosen at random. The character in the string's corresponding position is altered.

5. Decoding: Once this is completed, a new generation is created, and the process will be repeated until a stopping point is reached. The process is complete when the individuals who are closest to the optimal are decoded.

**1.2 The Traveling Salesman problem (TSP):**

The Traveling Salesman Problem (TSP) is a well-known method for solving problems that are simple to formulate but tremendously difficult to solve.

This is an NP-hard problem, meaning it cannot be solved in polynomial time. Many accurate and heuristic strategies have been developed in the field of operations research to tackle this difficulty (OR). The objective is to find the shortest path between n vertices while visiting each one exactly once.

TSPs are divided into symmetric and asymmetrical groups based on the structure of the cost matrix. TSP is symmetric if cij = cji, for all i, j and asymmetric otherwise. There are (n)! for the asymmetrical TSP ncity. One conceivable option, one or more cost-effective solutions Even for fairly big n, the number of solutions can become exceedingly enormous in some circumstances, necessitating a full search.

In a fraction of the time, a genetic algorithm can find a solution. While it may not identify the optimum answer, in less than a minute, it can find a near-perfect solution for 100 city tours. To solve TSP with GA, better than one of the two basic procedures. To begin, make a population out of a large number of random hits. This algorithm starts with a population of cities that are close to each other and prioritizes them greedily. Second, pick two parents who will be seeing the population's best (shortest) child and combine their trips to create two additional visits. This is done to avoid the ensemble's towers appearing to be identical. Two of the longer circuits were replaced by child circuits, which were newly introduced. Newly introduced child circuits replaced two of the longer circuits. The population size remained the same at. New circuit links for the children are made again until the required target is reached. In TSP the accuracy of the problem depends on the factors :

1. Speed

2. Population Size

The best answer will be picked by comparing these factors in each solution, and the new shortest path will be presented in each iteration. As the population grows larger, the work of making comparisons and then describing the solution becomes more complicated.

The evolution of life inspired genetic algorithms, which are heuristic search algorithms. To carry out generation, the algorithm is designed to replicate natural selection, i.e. survival of the fittest. The conventional genetic algorithm is broken down into five stages:

The initial population is being created.

Fitness assessment

Choosing the most advantageous genes.

Crossing the border.

Mutation is used to introduce variants.

These algorithms can be used to solve a variety of optimization problems. A salesperson is given a set of towns and must discover the shortest path to visit each city exactly once before returning to the starting city, according to the problem. In the following implementation, cities are employed as genes, and the string formed using these characters is referred to as a chromosome. while a fitness score generated is equal to the path length of all the cities given is used for targeting a population group. The Fitness Score is calculated using the length of the gene's reported route. The better the gene, the shorter the path length. The gene pool's fittest genes pass the population test and progress to the next iteration.

**1.3 Knapsack Problem:**

KP is a basic combinatorial optimization problem which is concerned with increasing or decreasing the value as in a cost or benefit. A Knapsack that can only hold a certain amount of volume or weight can hold a range of items but has a total volume limit. The question is how many of these items are to collect in order to maximize total profit or reduce total cost. Elements in integer type KP can have arbitrary integer values, whereas KP limits the number and type to 0 or 1. KP 0–1 does not allow users to carry multiple copies of the same element. 0–1 is a special case where each entry may or may not be loaded into a knapsack.

There are other variations of KP such as multidimensional KP 0–1 or multiple knapsack problem 0– first. To produce the best solution, dynamic programming utilizes the optimization principle. KP 0–1 and its variations are well-known Hard problems that can be solved using dynamic programming techniques in pseudo-molecular time.

**2. IMPLEMENTATION of Traveling thief problem:**

*Definition:*

There are n cities with the distance matrix D=dij provided. There are also m items, each with a value pk and a weight wk. There is a thief who is going to visit these cities only once and pick up some items to fill his knapsack. The knapsack's maximum weight is W.The goal is to find a tour in which it visits all the cities only once and returns to the starting city while optimizing objective functions and without exceeding the total weight of the knapsack. It should be noted that the objective function(s) may be related to the time of travel and/or the total value gained by the thief from picking the items. The rest of this report we assume that the thief begins in the first city. Furthermore, the thief can only pick up items from the first city at the start.

**2.1 Travelling sales man sub-problem is the first step in this implementation.**

TSP – Genetic Problem

Solves the Traveling Sales Person Problem with a genetic algorithm that decodes chromosomes as travelling sales cycles (solutions). Order 1 crossover, Swap mutation, total generation replacement, Roulette Wheel Method for selecting parents, and the negative impact of cycling distance on fitness.

Variables are n is an integer that represents the number of points as well as the number of genomes in each chromosome. K is the variable that represents number of chromosomes in each population is an integer. Max generation indicates indicating the algorithm's maximal generation (iterations).

First implementation of TSP using genetic problem

**ALGORITHM:**

Pseudocode for the algorithm used to solve the TSP problem

1.Creating the initial population.

2. Set up the population

3. Set up the population maximum number of generations

4. Determine the population's fitness.

5. Finding the best individual so far.

6. Create another generation

7. Replace the current generation with the following generation.

8. Return the most suitable individual.

crossover probability: float in the range [0,1] indicating the likelihood of a crossover.

mutation probability: float in the range [0,1] indicating the likelihood of a mutation.

path: a string indicating the location of a text file containing n point coordinates.

1. *input read*

Reads the first N lines of the.txt file given by path, which contain the point coordinates in

(x1,y1) and (x2,y2).

Variables are path i.e., a string indicating the location of a text file with n-point coordinates. |Nx2| Numpy is the return parameter.

a collection of point coordinates

Return Parameter that indicates

|Nx2| numpy.array of coordinates of points

There are no function calls.

1. *population initiation:*

Modifies pop to create a population of K chromosomes with N genomes for each chromosome. Variables are k i.e., chromosomal number. N is the number of genomes per chromosome. Numpy.array of K chromosomes with N genomes for each return parameter is |KxN|. No Call to Functions

*4. calculate fitness of population*

Fit is modified to compute the fitness of each chromosome in the population. The fitness of each chromosome is proportional to its cycle distance. Parameters are pop: list/numpy |KxN|. K chromosomes, each with its own genome|Nx2| list/numpy o points

a collection of point coordinates. K is the chromosomal number. N is each chromosome's number of genomes

o fit, a |K| list/numpy as a return parameter

fit[k] = fitness of pop[k] is an array of floats.

Examples:

Distance is the Call to Function here.

5. *finding the best individual*

Chooses the best individual and the fitness of that individual in all the generations. Variables are pop |KxN| list/numpy.array of K chromosomes, each with N genome

fitness: |K| list/numpy.array of values indicating each chromosome's fitness· Best individual: |N| list/numpy.array indicating the best individual/chromosome up to this point, excluding the current population.

best fit is number indicating best individual's fitness Return Variable is best individual so far, best individual so far fitness. There are no function calls

6. *create new population*

By crossing and mutations over the current population, a new population of K chromosomes of N genomes is created. Variables are pop i.e., |KxN| list/numpy.array of K chromosomes, each with a N genome, indicating the current population

|K| list/numpy o fitness each chromosome's fitness in pop culture. N is number of genomes in each chromosome k is number of chromosomes. Crossover probability and mutation probability are the floats indicating their respective probabilities in the range [0,1].

List/numpy.array of K chromosomes with N genomes for each chromosome indicating the new population

Call to Functions are \ select parent \decide to \ crossover\ mutate

7. *results plot*

The best last 15 chromosomes created across generations are plotted and displayed.

Pop is |KxN| list/numpy parameters. Best last 15 is array of K chromosomes with N genomes for each chromosome representing the current population. |Nx2| list/numpy of |M| deque/list of (A, B), where A is one of the best chromosomes, B is its cycle distance, and M=15 points is a collection of point coordinates. There is no parameter for return.

Plot individual path is a call to functions.

8. *distance*

Calculates a cycle's Euclidean distance.

Parameters are order: |N| list of ordering points |Nx2| list of point coordinates (zero indexed)

Parameter for Return are the points from point0 to point1 to... to point N back to point0 Euclidean distance

Examples:

*9. parent select*

Using the roulette wheel technique, select the index for parent based on each chromosome's fitness. There are no functions to call

Fitness is |K| a list/numpy.array of values reflecting each chromosome's fitness. K is fitness duration. Parameter for Return is

index of the parent who was chosen at random. Finding the cumulative distribution is the only function.

*10. decide to*

Returns True or False based on chance at random. P: float of [0,1]; probability that the return value will be True. True with a probability of p, and False with a probability of (1-p). There are no function calls.

*11. crossover*

By changing child1 and child2, it performs the Order 1 Crossover and produces two children. Parameters are child1 are |N| list/numpy.array representing the first child child2 is the N| list/numpy.array representing the second child. Father - |N| list/numpy.array Representing the father chromosome mother is |N| list/numpy.array representing the mother chromosome N is the total length of all input chromosomes.

*12. mutate:*

Parameters are ind are |N| chromosome /individual list/numpy.array

N is the ind length. There are No return parameter and calls to functions

*13. individual path plot*

Individual cycles are plotted in the index of a 3x5 display. Parameters are the individual list/numpy |N| |Nx2| list/numpy array of chromosomal points a collection of point coordinal testitle: The plot index is an integer in the range [1,15] that represents the plot's position in a 3x5 matplotlib subplot.

No calls to functions

14. cumulative distribution

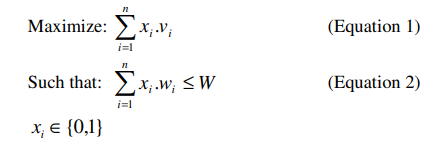
Calculates the cumulative distribution of arrays. Parameters are numpy. array of numbers |K| array. K is the array's total length. Return parameter is the cd[i] is the probability that a uniform random number in [0, arr.sum()] is less than arr[:i].

* 1. **KNAPSACK Problem:**

The knapsack problem mainly put the values and weights of n items in a knapsack with capacity W in order to obtain the maximum total value in the knapsack. Already given two arrays val[0..n-1] and wt[0..n-1], that represent the values and weights of n items, Find the maximum value subset of val[] whose sum of weights is less than or equal to W, given an integer W representing knapsack capacity.

Items in Knapsack cannot be broken, so the thief must either take the whole item or don’t take at all and this method is called the 0-1 Knapsack. Value of xi can be 0 or 1. 0-1 Greedy approach will not solve Knapsack. A greedy approach does not guarantee the best solution. In knapsack problem, some of the parameters used are parent size, mutation probability, generation limit, population size.

A collection of n objects, each with its own weight wi, value (profit) I v, and knapsack capacity W.



The problem is a simplified version of real-world constraint-based resource allocation problems. The greedy algorithm explained in [2] will always generate a solution that is more than half the optimum.  Core branch and bound algorithms [2] can also solve a large number of randomly generated instances in a very short time. But even so, as the correlation between the variables grows, so does the problem's difficulty.

* 1. **TRAVELLING THIEF:**

The TTP solution is built by providing the tour and picking plan. The workflow proceeds by first solving the Traveling Salesperson problem with a genetic algorithm, then the Knapsack problem, and finally the Traveling Thief problem with both problems.

Availability of item Ii in each city:

TTP can be solved by using both TSP and Knapsack problems

TSP gives n cities with a Dnn distance matrix. KP contains m items. For every item has a weight wi, a value bi, and an available city ai. The goal of a thief is to visit each city once, collect items along the way, and then come back to the starting city. To transport the items, the thief needs to rent a knapsack with a capacity of Q. The knapsack is available for rent at R per time unit.

The other TTP parameters are determined by the interdependence of the sub-problems (knapsack and tour). TTP's complete solution includes two vectors: tour & picking plan. For all I the plan is shown. Zi indicates in what city a item Ii must be selected from. Zi=0 indicates that item I was not selected at all.

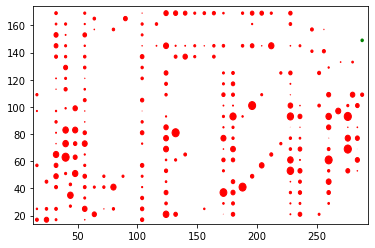
In generating genomes, we create random array of zeros and ones by keeping last half zero in starting gives early convergence, since we have less distance to travel. Then continue by the function of generating the population and fitness we need genomes, increasing weight, velocity while changing the cities, net profit and time for when to change the city. Profit of items in the current cities and weight addition one item at a time. The thief's speed decreases linearly as the total weight of items carried increases, as calculated by the formula as follows:

*v = vmax − νw, (1)*

*ν = vmax − vmin Q,*

Such that 0 = w = Q represents the present total weight of the chosen items. The speed (v = vmax) is maximised when the knapsack is empty (w = 0). When the knapsack is completely full (w = Q), the speed is reduced to a bare minimum (v = vmin). The total value of the selected items less the rent on the knapsack is then defined as the thief's profit.

Then select top k parent to generate child based on fitness and checking feasibility to get child and separating from the random index and mix the two parents. Mutation is performed where it is based on Mutation probability. The graph after re assigning profit gained from each city in the order of 0 to number cities:



**Input:**

KNAPSACK DATA TYPE: uncorrelated

DIMENSION: 280

NUMBER OF ITEMS: 2790

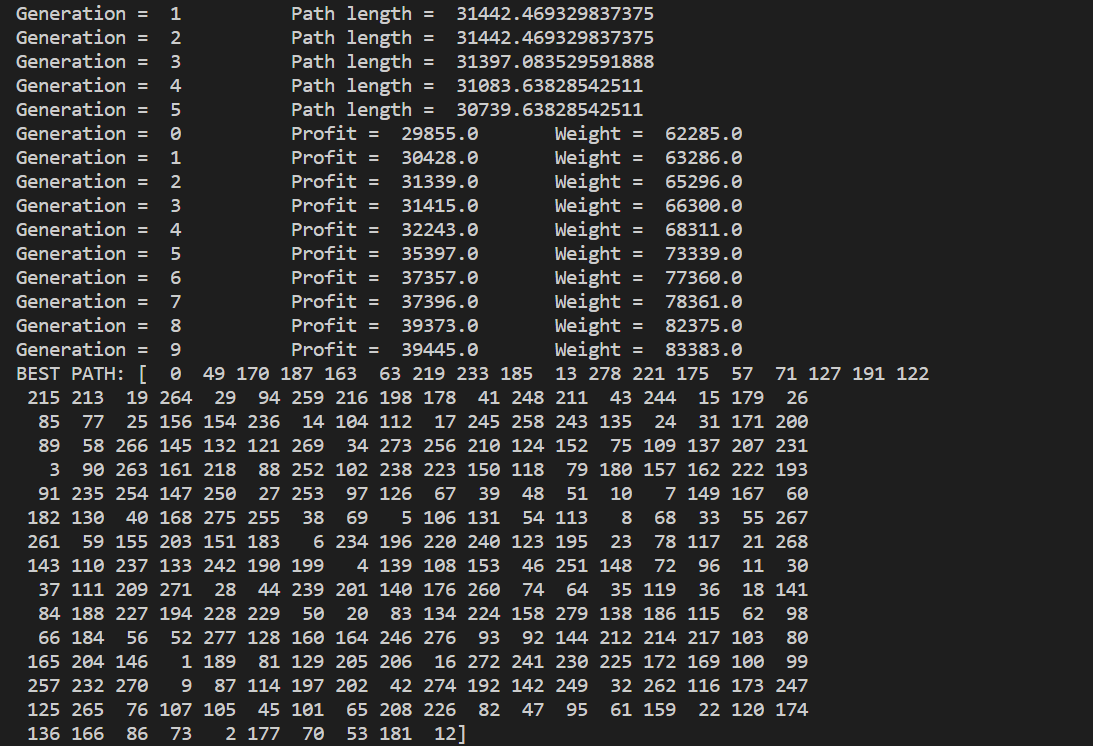
CAPACITY OF KNAPSACK: 1262022

MIN SPEED: 0.1

MAX SPEED: 1

RENTING RATIO: 208.53

**Output:**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time | Complexity | Advantage | Disadvantage |
| Genetic Algorithm | Exponential Time | O(Kmn) | Best Solution by using Fitness Criteria | Approximation of solution can be reached but not optimal |
| Greedy Approach | 5seconds for 15 cities | O(logn) | Solution accuracy | No best choices are considered so no accuracy |

**Conclusion:**

The Traveling Salesman Problem appears to be well-solved by genetic algorithms, although this is highly dependent on which mutation and crossover procedures are used to encode the problem We introduced a new crossover operator named after a genetic algorithm for the (TSP).

Experiments revealed that in terms of optimal solution, cost, and duration to solve, our crossover operator outperforms all other operators.

As a result, to improve the solution's quality. In order to prove the precise nature of the crossover operator, we also chose the highest probability of crossing. Only the mutation with the lowest likelihood is used in which it is needed.

We provided a comparison of the Greedy method and the Genetic Algorithm for TSP. It's difficult to say which moderate-sized instances our crossover operator didn't solve. As an outcome, implementing a optimal local search algorithm in to algorithm may fix all of the remaining cases, which can be researched furthermore as future research.

**REFERENCES:**

[1] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," Oper. Res., vol. 21, pp. 498-516, 1973.

[2] “Knapsack Problems Algorithms and Computer Implementations”, S. Martello and P. Toth, John Wiley and Sons (1990)

3] “Adaptation in natural and artificial systems”, MIT press, Cambridge, Massachusetts (1975)

[4] “Genetic Algorithm in Search, Optimization and Machine Learning”, D. E. Goldberg, Addison Wesley publishing company, Massachusetts (1989)

[5] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, Genetic programming: an introduction. Morgan Kaufmann San Francisco, 1998, vol. 1.

[6] “Optimization by simulated annealing”, S. Kirkpatrick, Science, Number 4598, 13 May 1983, volume 220, 4598, 671--680 (1983)

[7] “Theoretical and Numerical Constraint Handling Techniques used with Evolutionary Algorithms: A Survey of the State of the Art”, C. Coello, Computer Methods in Applied Mechanics and Engineering, 191 (11--12), 1245-1287, January 2002.