ECE 457A: Course Project

Project # 9: GROCERY SHOPPING OPTIMIZATION PROBLEM

University of Waterloo

Spring 2015

Ahson Khan - aa26khan

Peter Chau - p2chau

Baglan Daribayev - bdaribay

Ruobin Li - r47li

Cheuk-Hang (Angela) Ko - c2ko

Table of Contents

List of Tables 4
List of Figures 5
Summary 6
Chapter 1: Introduction..... 7
Chapter 2: Literature Review..... 9
  Simulated Annealing..... 9
  Genetic Algorithm..... 9
  Tabu Search.....10
  Ant Colony System.....11
  Particle Swarm Optimization.....11
  Artificial Bee Colony 12
  Conclusion 13
Chapter 3: Problem Formulation and Modeling 14
  Problem Formulation 14
  Reduced Problem.....15
  Real Life Problem.....16
Chapter 4: Proposed Solution 17
  Objective Function.....17
  Reduced Version of the Problem.....17
  Simulated Annealing.....19
    Algorithm and Implementation.....19
    Adaptation.....19
    Termination Criteria 20
    Hand Iterations 20
  Genetic Algorithm.....22
    Algorithm and Implementation.....22
    Operators and Parameters.....23
    Adaptation.....23
    Termination Criteria 23
    Hand Iterations 24
  Tabu Search.....27
    Algorithm and Implementation.....27
    Adaptation.....28
    Hand Iterations 29
  Ant Colony System.....33

Algorithm and Implementation....33
Adaptation....34
Hand Iterations....34
Particle Swarm Optimization....39
Algorithm and Implementation....39
Operators....39
Parameters and Tuning....40
Adaptation....41
Termination Criteria....42
Hand Iterations....42
Artificial Bee Colony....51
Algorithm and Implementation....52
Adaptation....52
Termination Criteria....53
Chapter 5: Performance Evaluation....54
Simulated Annealing....55
Genetic Algorithm....56
Tabu Search....59
Ant Colony System....61
Particle Swarm Optimization....62
Artificial Bee Colony....64
Result Comparison....65
Chapter 6: GUI and Usability Guide....67
Chapter 7: Conclusions and Recommendations....69
References....70
Appendices....71
Appendix 1: SA solution....71
Appendix 2: GA Solution....72
Appendix 3: TS Solution....73
Appendix 4: ACS Solution....74
Appendix 5: PSO Solution....75
Appendix 6: ABC Solution....76

List of Tables

Table 1: TS – Initial Tabu List.......30
Table 2: TS – Iteration 1 Tabu List.......31
Table 3: TS – Iteration 2 Tabu List.......33
Table 4: PSO – Current solution for Particle 1.......43
Table 5: PSO – Current solution for Particle 2.......43
Table 6: PSO – Current global best.......44
Table 7: PSO – Global best solution at the end of iteration 1.......47
Table 8: PSO – Global best solution at the end of iteration 2.......51
Table 9: Real user's shopping list.......54
Table 10: SA – Experimental Results for Real Data.......56
Table 11: GA – Experimental Results for Real Data with Population of 20.......58
Table 12: GA – Experimental Results for Real Data with Population of 10.......58
Table 13: TS – Experimental Results for Real Data with Maximum Iterations 100.......60
Table 14: TS – Experimental Results for Real Data with Maximum Iterations 200.......60
Table 15: ACS – Experimental Results for Real Data.......62
Table 16: PSO – Experimental Results for Real Data with Population of 10.......63
Table 17: ABC – Experimental Results for Real Data.......65
Table 18: Results Comparison for All Six Algorithms.......66
Table 19: SA Solution for Real Data.......71
Table 20: GA Solution for Real Data.......72
Table 21: TS Solution for Real Data.......73
Table 22: ACS Solution for Real Data.......74
Table 23: PSO Solution for Real Data.......75
Table 24: ABC Solution for Real Data.......76

List of Figures

Figure 1: Graph of Simulated Annealing Solution Cost vs Number of Iterations 55
Figure 2: Graph of Genetic Algorithm Solution Cost vs Number of Generations 57
Figure 3: Graph of Tabu Search Solution Cost vs Number of Runs 59
Figure 4: Graph of Ant Colony Optimization Solution Cost vs Number of Iterations 61
Figure 5: Graph of Particle Swarm Optimization Solution Cost vs Number of Iterations 63
Figure 6: Graph of Artificial Bee Colony Solution Cost vs Number of Iterations 64
Figure 7: Example GUI of the PSO algorithm showing parameters and results 68

Summary

This report examines a shopping list plan problem. A customer is aiming to purchase a list of groceries, but doesn't know which stores to visit for them to pay and travel as little as possible. The problem is very similar to the Travelling Salesman Problem (TSP) in that there is a route generated and an attempt is made to generate a minimum distance cost. However, it is different in that it brings a second variable of price into the problem and also, that there are different stores that sell the same items at different prices, causing there to be a large combination of routes to be generated and evaluated and exponentially increasing the search space.

The report attempts to solve this problem by implementing six metaheuristic techniques: Simulated Annealing, Genetic Algorithms, Tabu Search, Ant Colony Optimization, Particle Swarm Optimization, and Artificial Bee Colony. Because there are no concrete data sets with known optimal solutions, all of our comparisons will be relative to the performance of the other algorithms. The alternatives will be compared based on the total time taken to converge and output their generated solution and the objective function cost of the final solution.

The report finds that Simulated Annealing, Tabu Search, Artificial Bee Colony and gave relatively poor or mediocre solutions in a moderate amount of time. Algorithms that provided the best objective function solutions were the Ant Colony System and Particle Swarm Optimization. These two algorithms however take a long time to compute their solution. As a result, we found that the Genetic Algorithm implementation which gave the fastest solutions with relatively good solution costs is best suited for practical use with real customers.

Chapter 1: Introduction

Grocery shopping is a common activity that people do quite often. A lot of time is spent to try to save money by scanning flyers from various stores for deals. For some, this may be a fun task, but for many others, it is quite tedious and many savings are lost due to incomplete information and lack of time. The goal of this report is to make this task less tedious by automating the process and return a result in a reasonable amount of time.

Given a user's shopping list and initial starting location, the alternatives will try to find which stores to visit to purchase their items and the route to take to spend the least amount of time possible, as well as the least cost possible. The program will have access to stores, their inventory, and their prices. It will also have access to the travel distance between each store locations.

The grocery shopping problem solution where a pseudo-optimal solution is generated is an extremely sought after solution within the software industry as there are several applications within the mobile phone markets that attempt to fill this user need. Some applications for grocery tracking include Checkout 51, RedFlagDeals, Flipp, and Grocery IQ, to name a few, and many have millions of users (for instance Flipp has over one million downloads, from the Android Play Store) [8]. Many of these applications have limited algorithmic capabilities and usually only display the available product data (scoped to certain geo-locations), and allow the users to track their lists while mobile. They lack the computation necessary to generate a solution that is both viable and helpful to the user, which, in the end significantly reduces the user's search time and makes shopping a lot more convenient while cutting the grocery bill immensely. This is the key motivating factor to tackle this crucial problem and find a useful solution, and if packaged correctly can be successful start-up product that is different from the competition and can lead to market success.

The problem is a multi-objective problem that is trying to minimize both price and distance. Because there are an extremely large amount of solutions to explore, a brute force method would take too long for a shopper to wait. This report will use six metaheuristic algorithms: Simulated Annealing (SA), Genetic Algorithm (GA), Tabu Search (TS), Ant Colony System (ACS), Particle Swarm Optimization (PSO), and Artificial Bee Colony (ABC) in an attempt to solve the problem. Although these algorithms will most likely not find the optimal solution, they will generate a near optimal solution in a much shorter amount of time that a shopper would be willing to wait. All of the implemented algorithms will use a combined weighted objective function of half of the total price cost and half of the total distance to simplify the process.

The solutions will be tested against randomly generated data of various sizes and with various inputs along with the real data set to generate useful results. Solutions will be implemented in MATLAB. Each program will be run on the same machine and timed by using the tic-toc functionality in MATLAB. Programs will be run multiple times and the average of results will be taken to help avoid variance due to the stochastic nature of each algorithm. Solutions will be evaluated by their optimality and their time cost.

Chapter 2 is an analysis of previous solutions to similar problems and a description of how the solution to this problem expands on those techniques or how it differs from them. The problem is formulated and modeled in chapter 3. In chapter 4, a solution for each algorithm is shown, with hand iterations on a reduced problem set. The results and performance evaluation is done in chapter 5. The explanation of the GUI and how to generate results for each of the algorithm is provided in Chapter 6 (along with the README files). Chapter 7 shows the conclusions and recommendations of the report.

Chapter 2: Literature Review

The problem is somewhat similar to the traveling salesman problem (TSP), since there is the aspect of minimizing the distance between stores. However, the main difference is that there is the second variable of the products that are required, which means that the TSP problem itself can change depending on which stores are chosen for a certain purchase. The addition of extra dimensions in the problem space makes this problem very challenging and worthwhile as the conclusions are equivalently more valuable.

TSP is a well-documented problem with multiple solutions. Various solutions for TSP will be analyzed since it is the most similar to our problem. Different algorithms and techniques are discussed below, under the categories of each algorithm.

Simulated Annealing

We found a paper that uses some unique ideas with SA to try to solve TSP such as introducing a demon creditor to work with acceptance of new solutions [4]. They concluded with various results, most notably that with large data sets, plain SA seems to work the best compared to the demon variations and that smaller and medium variations tend to favour the demon rather than the regular acceptance formula. This report will be attempting plain SA because when adding in the factor of selecting a different store, many different possible TSP routes can be considered and the number of combinations will be huge.

Genetic Algorithm

Genetic Algorithms (GA) have been used to optimize solutions for the TSP, although the representation of the problem can be difficult [6]. Issues arise in using genetic algorithms in binary representation as crossover and mutation often causes illegal tour paths to be generated. A possible alternate representation is through using ordinal representation, however the results after crossover are generally random and not usually improving. Other solutions maybe the adjacency representation, which requires the crossover operator to be modified from the classical crossover operator. Another option is to have crossover on sub routes rather than the entire route, to ensure validity of the solution [6].

A similar approach is used in our solution, where the crossover operator is more sophisticated, so that the path chosen for the solution is always valid. However, since the TSP part of the problem is dictated by the stores chosen for each purchase, the crossover operator works on swapping the stores for a particular item rather than the order of purchase.

Tabu Search

We considered a paper that discussed the Tabu Search heuristic approach for TSP [2]. The journal article describes the undirected selective traveling salesman problem (STSP) as a graph problem and how to take an optimal profitable subset of the vertices and make the shortest distance Hamiltonian cycle. STSP is an NP-hard combinatorial type problem where the difficulty comes from the fact that the cost and distance are independent parameters. Optimal solutions to one of the parameters is not necessarily optimal to the other. The paper recognized that although several heuristics exist that can solve a small set of TSP, none were consistently good. They were quick in performance, but did not provide enough diversification in exploring the whole solution space. The paper was confident that the proposed tabu search method is the near-optimal solution to solve STSP type of problems. The TS method used inserting/removal as the neighbourhood structure operator, but the unique part was that this was applied to a cluster of vertices at a time, rather than exclusively at one vertex at a time. Partitions of vertices into clusters used a proximity measure. For each possible move, removal is evaluated as a ratio of distance reduced to cost increased. The insertion move is evaluated as a ratio of cost reduced to distance increased. The tabu status was assigned to move with the length which was randomly determined by a discrete uniform distribution. The aspiration criterion was that if insertion/removal move yields a feasible solution, it is allowed regardless of the tabu status. The test results were shown to be near optimal [2].

Ant Colony System

In [1] a similar problem of solving TSP is discussed. The algorithm has several parameters that are used to tweak the speed of convergence or exploration rate. There are two updating rules applied - local-updating and global-updating. For each edge, when choosing a certain path to follow, the algorithm decides whether to pick the greedy one (exploitation) or pick randomly (exploration). The parameter that configures that is known as q0 (where q0 is between 0 and 1). For 0, the next edge might be chosen randomly, and for 1, the choice is greedy (i.e. chose the best next edge).

Particle Swarm Optimization

The paper listed in [7] discusses the PSO-based algorithm for a multi-objective problem and it proposes a hybrid algorithm based on particle swarm optimization for a multi-objective permutation flow shop scheduling problem. This problem is a typical NP-hard combinatorial optimization problem similar to our grocery shopping problem. It suggests the utilization of several adaptive local search methods to perform exploitation. First, a ranked-order value (ROV) rule based on a random key technique is used to convert position values of particles to job permutations. A similar approach is taken to convert probabilities to store permutations within each particle solution for the grocery problem (to generate the velocity vector 1). Second, a multi-objective local search is applied based on the Nawaz-Enscore-Ham heuristic which was not applicable to our problem due to its increased complexity. The proposed algorithm achieves diversity by randomizing the weight vectors near the end of each iteration and this objective is achieved with the adaptation of the inertia weight in our implementation. In summary, the proposed algorithm sequentially applies these functions one after the other within each iteration, by first using PSO operations, followed by the ROV permutations, and then the randomization of the weight vectors. Lastly, it does the multi-objective NEH-based local search. Inspiration from this methodology was taken while implementing our current PSO algorithm to solve the grocery shopping problem.

Since our problem has a component which is very similar to the travelling salesman problem, another paper was consulted which discusses the application of PSO to the TSP [3]. Particle swarm optimization is an evolutionary computation technique inspired by social particles (like birds or ants). An effective PSO approach for TSP is to consider distinct velocity operators. The velocity vector is defined as probabilities of exchanging two elements of the permutation vector using a pairwise exchange. Other approaches involve constructing the PSO algorithm for discrete optimization problems where the search space is defined along with a physical neighbourhood which maps to the logical neighbour of particles within the algorithm. The paper suggests applying new velocity operators for the discrete PSO where probabilities are defined for whether the particle should move towards the local best or the global best. This approach was followed rudimentarily in our algorithm. As the algorithm moves along, one of the probabilities is decreased while the other is increased as part of the adaptation, where the probabilities towards the global best become much higher. This rationale was implemented by starting off with a slightly higher probability to tend towards the global best solution.

Artificial Bee Colony

With regards to Bee Colony optimization for TSP, a paper done by Karaboga and Gorkemli was found [5]. Their neighbourhood exploration operator was limited to swapping a city’s index with the previous index or the index afterwards based on the random phi value generated. Since our problem is slightly different and has two type of operators instead, we decided to make our operators randomly decide to swap stores or swap locations instead. They also used a dampened fitness function to help the weaker solutions gain more traction from the onlooker bees. Instead, this report uses adaptive parameters to help weaker solutions live longer and get the chance to be visited more often. They concluded with near-optimal performance in very large datasets (150 and 200 places) and outperformed many other genetic algorithm variations. As a result, we believe that ABC will be a good candidate algorithm to solve our problem and chose it as our self-study algorithm.

Conclusion

While experiments have been made on similar problems, the problem we have chosen has unique constraints and optimization factors that require a slightly different approach in every algorithm. However, much of the information gained from these papers can be used in the implementation of our algorithms as explained within each of the algorithm specific sections.

Chapter 3: Problem Formulation and Modeling

Problem Formulation

Given a list from the user of desired products they want to purchase, P, and their initial location starting location, L, we want to find the minimum cost defined by the objective function:

$$minimize\ f(x) = 0.5 * D(x) + 0.5 * C(x)$$

In the optimization equation above, $D(x)$ is the total distance for the return trip route taken to visit all the stores to purchase all the desired items and $C(x)$ is the total cost to purchase all the items desired from the stores visited.

The vector x will be a solution generated by the algorithms that contains a mapping to the list of items to be purchased and the stores at which each of those items should be purchased. The order of the vector x dictates the route, i.e., the order in which to visit each of the stores.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} item_1, store_1 \\ item_2, store_2 \\ \dots \\ item_n\ store_n \end{pmatrix}$$

The data set that is required to solve this problem includes the distance matrix between each location and store as well as a product catalogue of each store which contains price information. The distance matrix is of the form:

$$Distance = \begin{bmatrix} store_1 \\ store_2 \\ \dots \\ store_j \\ location_1 \\ location_2 \\ \dots \\ location_k \end{bmatrix} \begin{bmatrix} 0 & dist_{1,2} & \dots & \dots & \dots & \dots & \dots & dist_{1,j+k} \\ dist_{2,1} & 0 & dist_{2,3} & \dots & \dots & \dots & \dots & dist_{2,j+k} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ dist_{j,1} & dist_{j,2} & \dots & dist_{k,(j-1)} & 0 & dist_{k,j+1)} & \dots & dist_{k,j+k} \\ dist_{j+1,1} & dist_{j+1,2} & \dots & \dots & \dots & 0 & \dots & \dots \\ dist_{j+2,1} & dist_{j+2,2} & \dots & \dots & \dots & \dots & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ dist_{j+k,1} & \dots & \dots & \dots & \dots & \dots & \dots & 0 \end{bmatrix}$$

Where each $dist_{a,b}$ is the distance between address and address b (either store or user location) and there is a diagonal of 0 (since the distance between an address and itself is 0. The values of $dist_{a,b}$are equal to$dist_{b,a}$.

The inventory matrix is of the form:

Store 1	Item 1	Price$_{\text{item1}}$	Item 2	Price$_{\text{item2}}$	…
Store 2	…				
…					
Store j	…				

$D(x)$ is calculated by using the distance matrix and adding the distance values for each of the stores visited as defined in the vector x, along with adding the location for the user as the first and last node in the route.

$C(x)$ is calculated by using the inventory matrix and adding the cost of each of the items in the purchase list at the store it was purchased from as defined in the vector x, along with multiplying that item cost with the quantity of that item purchased.

The algorithms will have access to the list of stores and what products they sell with prices (inventory matrix). It will also know the distances between each store and the distances between the stores and the user's starting location, L (distance matrix).

The total distance of the route, $D(x)$ can be modelled similar to the travelling salesman problem.

Reduced Problem

The required data for the reduced problem can be found in the RandomData directory.

RandomData / outputInventorySmall.txt

RandomData / outputDistanceSmall.txt

The reduced problem is used for the hand iterations of each of the algorithms. The data set used contains only 10 different stores to choose from and each store sells very few items with relatively small distance values. The stores, items they sell, and the distances between them are randomly generated from an initial set of data. The price and distance data is modified by a multiplier that is added randomly to each of the items in the data set. The hand iterations start with the user at location_0 who has a purchase list containing only 3 items (duck, chicken, and apples) where only one 'unit' of each item is purchased.

Real Life Problem

The required data for the real life problem can be found in the RandomData directory.

RandomData / REAL_inventory.txt

RandomData / REAL_distances.txt

The real problem data set is used to measure the performance and results of each of the algorithms. The data set used contains several grocery stores from the city of Waterloo and includes some stores with branches located around the city. Along with the stores, the items that are sold from each store are scraped from the store websites and flyers to get accurate and realistic product catalogue and price data. The distance matrix between each store is generated based on the travel distance by car (gathered from the Google Maps APIs). The user's location are defined to one of three locations based on the heavily populated areas (University of Waterloo, RIM Park, and the Grand River Hospital). These locations are chosen based on what would occur most commonly and encapsulate the surrounding regions as well. The user's shopping list is composed of 15 items that are commonly found in a typical shopping list. These items were chosen since many stores sell them at various costs so that there are sufficiently interesting results. All price values are in cents and all distance values are in meters.

Chapter 4: Proposed Solution

Objective Function

Minimize $f(x)$, where

$$f(x) = w_d * sum\ of\ round\ trip\ distances + w_c * total\ cost\ of\ purchasing\ items$$

Where w_d is the weight of the total distance, and w_c is the weight of the total cost of purchase.

The sum of round trip distances is the sum of the distance between the starting location and the first store to visit, the distance between that store and the next store in the route, and so on, with the last distance being the distance between the last store to visit and the starting location.

The total cost of the purchase is the sum of the cost of each item purchased at a given store multiplied by how many of those items that are purchased.

Reduced Version of the Problem

Each algorithm is tested with the reduced problem set defined below. The neighbourhood operator, the initial solution, and solving strategy are specific to each algorithm.

A small dataset is randomly generated with 10 stores and 10 different items.
User requires: one of chicken, apples, and duck
User start location: location_0

Stores and locations distance matrix (each row i is the distance between store i and all other stores and locations, hence the 0 in the diagonal).

RandomData / outputDistanceSmall.txt

```
Store_0 0 673 242 95 867 729 399 729 283 223 867 375
Store_1 673 0 54 489 65 144 546 5 493 582 606 194
Store_2 242 54 0 586 606 960 792 128 457 863 982 54
Store_3 95 489 586 0 39 381 654 993 438 453 187 741
Store_4 867 65 606 39 0 725 178 651 827 927 921 344
Store_5 729 144 960 381 725 0 364 83 701 2 592 626
Store_6 399 546 792 654 178 364 0 994 627 796 448 305
Store_7 729 5 128 993 651 83 994 0 530 711 697 557
Store_8 283 493 457 438 827 701 627 530 0 963 449 908
Store_9 223 582 863 453 927 2 796 711 963 0 826 463
Location_0 867 606 982 187 921 592 448 697 449 826 0 682
Location_1 375 194 54 741 344 626 305 557 908 463 682 0
```

The purchase or inventory catalogue of each store along with the price (in cents).

RandomData / outputInventorySmall.txt

```
Store_0 Duck 536
Store_1 Chicken 359
Store_2 Chicken 323 Duck 488 Apples 40 Pork 347
Store_3 Chicken 325 SomethingElse 1006
Store_4 Duck 481
Store_5 VeryExpensiveItem 853
Store_6 MediumItem 454 Utensils 106
Store_7 Pork 350 Grapes 19 Apples 49
Store_8 Utensils 97 Chicken 323
Store_9 Wings 286 Duck 427 Oranges 79 Pork 312
```

Simulated Annealing

Algorithm and Implementation

SA works very similarly to hill climbing in that it first starts out with a current solution and then tries to find a better solution by performing a neighbourhood operator on the current solution. Hill climbing only accepts improved solutions whereas SA will conditionally accept solutions based on a temperature parameter that gradually decreases as iterations of the algorithm pass.

The temperature is used for accepting solutions that did not improve as an attempt to avoid being stuck in local optima. The formula to accept non-improving solutions is

probability = e^(-(costNew - costOld))/(temperature)

The temperature of the algorithm begins at a value that is based on the problem set. Depending on the size of the values we are expecting, the temperature may start at a higher or lower value to help properly reflect the probability. We found that this value accepted a good range of solutions at the beginning to help the exploration of the search space. The implemented algorithm has a geometric cool down schedule (Temperature = Temperature * alpha where alpha is 0.9). The cool down schedule runs every few iterations and the number of iterations required per cool down increases as the algorithm progresses.

Adaptation

SA searches for new solutions by either swapping the order of items bought, which swaps the route, or by buying an item from a different store (than the store currently selected). The implemented algorithm first begins with a large number of swaps per iteration and decreases if the algorithm continuously has good results to try to intensify in the specific search space. Once it detects stagnation after many iterations, the algorithm adapts and tries to slowly reheat and increase the temperature and number of swaps to try to explore in other areas.

Termination Criteria

The algorithm terminates under two conditions. One is running for the max number of iterations slotted for the algorithm. The other is if no solution has been accepted for a large number of runs even with reheating and the increased number swaps operators, the algorithm stops.

Hand Iterations

Pick SA parameters
alpha = 0.9
temperature = 200
Cool down schedule geometric and decrease per iteration
temperature = temperature * alpha;

Goal:
Start location = Location_0
Buy Duck, Chicken, Apples

Generating Random Initial Solution:

Duck, Chicken, Apples bought at...
Store_0, Store_1, Store_2

Distance = Location_0, Store_0, Store_1, Store_2, Location_0
= 867 + 673 + 54 + 982
= 2576
Price = 536 + 359 + 40

= 935

ObjectiveFcn = 0.5*2576 + 0.5*935

= 1755.5

Iteration 1:

Randomly decide to swap buying Duck and Chicken first

And no store swaps

Chicken, Duck, Apples bought at...

Store_1, Store_0, Store_2

Distance = Location_0, Store_1, Store_0, Store_2, Location_0

= 606 + 673 + 242 + 982

= 2503

Price = 359 + 536 + 40

= 935

ObjectiveFcn = 0.5*2503 + 0.5*935

= 1719

This solution is an improvement from the best solution so far (1755.5) so we accept.

Cool down...

temperature = temperature * alpha

= 200 * 0.9

= 180

Iteration 2:

Randomly decide to buy Chicken at Store_3 instead of Store_1

And no order swaps

Chicken, Duck, Apples bought at...
Store_3, Store_0, Store_2

Distance = Location_0, Store_3, Store_0, Store_2, Location_0
= 187 + 95 + 242 + 982
= 1506
Price = 325 + 536 + 40
= 901
ObjectiveFcn = 0.5*1506 + 0.5*901
= 1203.5

This solution is an improvement from the best solution so far (1719) so we accept.
Cool down...
temperature = temperature * alpha
= 180 * 0.9
= 162

Genetic Algorithm

Algorithm and Implementation

In brief, a genetic algorithm uses a population of solutions, and recombines or mutates them, with some preference for better solutions and a penalty for poor solutions. In this case, the best solution of each generation is kept without modification for the next generation and the worst solution is removed.

One of the main issues in genetic algorithms is the encoding of the problem. In this case, both value encoding and permutation encoding was used for the genetic algorithm. The permutation allows for the TSP part of the problem to be optimized, while the value encoding allows the cost to be optimized. Binary encoding of the store order would not be appropriate, as mutations or crossover could lead to invalid solutions.

Instead stores themselves are swapped around. Each solution is encoded as an ordered list of items and an ordered list of stores at which each item is purchased.

Operators and Parameters

There are two operators on the neighbourhood in genetic algorithms, the crossover and the mutation. Crossover swaps stores to buy an item from between two solutions. Mutation swaps the order of buying items and swaps a store with another random store to buy an item at. The items and stores are randomly chosen.

In the actual implementation, the population is set to 20 and the crossover rate to 0.9, with the mutation rate at 0.1. The population size is chosen so that the removal of the single worst solution and keeping the best solution can affect the overall fitness of the population, but still have enough candidate solutions to properly explore and intensify. A crossover rate of 0.85 - 0.9 is recommended for the genetic algorithm, which is why it is chosen to be 0.9.

Adaptation

The algorithm is set to be adaptive, with the mutation rate changing depending on how long it has been since the best solution last improved. If the solution has not improved in a set number of generations, the mutation rate will increase, so that the algorithm becomes more exploratory.

Termination Criteria

The algorithm terminates after a certain number of mutation rate increases or the number of generations goes over 500.

Hand Iterations

In this reduced problem, probability of crossover is chosen as 0.5. In the actual solution it should be around 0.8-0.9. As well, for this reduced problem size the population size is chosen to be 5, but in the actual solution population should be larger.

1. Generate initial population candidate solutions, as shown below

```
for each CS (current solution):
    randomize buying order
    for each required item
        randomly choose store for each item
    end
end
```

Initial population:

CS1: ['Duck', 'Chicken', 'Apples' ;
'Store_9', 'Store_1', 'Store_2']
CS2: ['Chicken', 'Apples' 'Duck';
'Store_3', 'Store_2', 'Store_9']
CS3: ['Chicken', 'Duck', 'Apples' ;
'Store_2', 'Store_4', 'Store_7']
CS4: ['Duck', 'Chicken', 'Apples' ;
'Store_0', 'Store_2', 'Store_7']
CS5: ['Apples', 'Duck', 'Chicken' ;
'Store_7', 'Store_2', 'Store_2']

2. Evaluate each solution:

CS1: 0.5(826 + 582 + 54 + 982) + 0.5(427 + 359 + 40) = 1635
CS2: 0.5(187 + 586 + 863 + 826) + 0.5(325 + 40 + 427) = 1627
CS3: 0.5(982 + 606 + 651 + 697) + 0.5(323 + 481 + 49) = 1894.5

CS4: 0.5(982 + 606 + 651 + 697) + 0.5(323 + 481 + 49) = 1421

CS5: 0.5(697 + 128 + 0 + 982) + 0.5(49 + 488 + 323) = 1333.5

3. Initial changes:

remove worst solution : CS3 removed ->

keep a copy of best solution: CS5

nextGenSolution1 = NS1 = CS5

4. randomly select half of {CS1, CS2, CS4, CS5} for crossover

CS1, CS3

randomly choose item to swap stores for: ‘Chicken’

NS2: [‘Duck’, ‘Chicken’, ‘Apples’ ;
‘Store_9’, ‘Store_2’, ‘Store_2’]

NS3: [‘Chicken’, ‘Duck’, ‘Apples’ ;
‘Store_1’, ‘Store_4’, ‘Store_7’]

5. select remaining for mutation: CS2, CS5

choose random item to swap stores for: ‘Duck’: store_9 → store_4

choose random items to swap orders: ‘Apples’ ‘Duck’

NS4: [‘Chicken’, ‘Duck’, ‘Apples’;
‘Store_3’, ‘Store_4’, ‘Store_2’]

choose random item to swap stores for: ‘Chicken’ store_2 → store_8

choose random items to swap orders: ‘Duck’ ‘Apples’

NS5: [‘Duck’, ‘Apples’, ‘Chicken’ ;
‘Store_2’, ‘Store_7’, ‘Store_8’]

FINISH 1st ITERATION

NEXT ITERATION

1. Take last generation as new current solution CS = NS

2. Evaluate each solution:

    CS1: 0.5(697 + 128 + 0 + 982) + 0.5(49 + 488 + 323) = 1333.5

    CS2: 0.5(826 + 863 + 0 + 982) + 0.5(427 + 323 + 40) = 1730.5

    CS3: 0.5(606 + 65 + 651 + 697) + 0.5(359 + 481 + 49) = 1454

    CS4: 0.5(187 + 39 + 606 + 982) + 0.5(325 + 481 + 40) = 1330

    CS5: 0.5(982 + 128 + 530 + 449) + 0.5(488 + 49 + 323) = 1474.5

3. Initial changes:

    remove worst solution : CS2

    keep a copy of best solution: CS4

    nextGenSolution1 = NS1 = CS4

4. randomly select half of {CS1, CS3, CS4, CS5} for crossover

    CS4, CS5

    randomly choose item to swap stores for: ‘Duck’

        NS2: [‘Chicken’, ‘Duck’, ‘Apples’;

        ‘Store_3’, ‘Store_2’, ‘Store_2’]

        NS3: [‘Duck’, ‘Apples’, ‘Chicken’ ;

        ‘Store_4’, ‘Store_7’, ‘Store_8’]

5. select remaining for mutation: CS1, CS3

    choose random item to swap stores for: ‘ Apples’: store_7 → store_2

    choose random items to swap orders: ‘Apples’ ‘Duck’

        NS4: [‘Duck’, ‘Apples’, ‘Chicken’ ;

        ‘Store_2’, ‘Store_2’, ‘Store_2’]

    choose random item to swap stores for: ‘Chicken’ store_8 → store_1 choose random items to swap orders: ‘Chicken’, ‘Apples’

NS5: [‘Duck’, ‘Chicken’, ‘Apples’ ;
‘Store_2’, ‘Store_1’, ‘Store_7’]

FINISH 2nd ITERATION

Final solutions:
CS = NS

evaluate each solution:
CS1: 0.5(187 + 39 + 606 + 982) + 0.5(325 + 481 + 40) = 1330
CS2: 0.5(187 + 586 + 0 + 982) + 0.5(325 + 488 + 40) = 1304
CS3: 0.5(921 + 651 + 530 + 449) + 0.5(481 + 49 + 323) = 1702
CS4: 0.5(982 + 0 + 0 + 982) + 0.5(488 + 40 + 323) = 1407.5
CS5: 0.5(982 + 54 + 5 + 697) + 0.5(488 + 359 + 49) = 1317

The best solution is CS2, 1304. The best solution has improved by approximately 3% over 2 generations.

Tabu Search

Algorithm and Implementation

Tabu Search (TS) for the problem statement can be solved using a similar strategy as the classic traveling salesman problem [1]. Tabu search is a stochastic trajectory type algorithm. The problem setup is such that visiting stores that are selling items is analogous to visiting cities. There’s only one shopper/traveler making visiting a sequence of destination nodes and then return home to the point of origin. The first node does not count as a store and is the point of origin, a neighbourhood example of 5-nodes route sequence can look like this [1, 2, 4, 5, 3, 1].

The neighbourhood operator is a pairwise swap of two stores to generate new solution [1, 3, 4, 5, 2, 1]. After the swap, the new objective function solution is calculated. All possible swap pairs are checked per iteration. Termination condition is when a pre-specified number of iterations have been performed.

The optimization criteria is to minimize the total distance of shopping route and the cost of shopping list items. The objective function will evaluate the sum of distance and total cost of the route sequence, with their associated weights. The lowest value result from the objective function is considered the best solution. The best indices of pairwise swap neighbourhood solutions were kept as tabu memory entry, which become forbidden swap moves in the following iterations for as many as the number of iterations as specified by the tabu length.

The tabu memory matrix keeps track of which swap moves were marked for tabu. The indices of the elements represent possible swaps between stores. If the element value is greater than zero, then it is a tabu move. After each iteration, all tabu moves are decremented by 1. For recency criteria, when a matrix element gets decrement back down to zero, then it means the move is no longer recent enough to consider a tabu move.

Adaptation

For the adaptive criteria, the cost solution from previous iterations was recorded in short term memory. If previous iteration solution had greater values than the current iteration, the result improved, and the tabu length is increased by 1 to encourage intensification. If previous iteration solutions had lower value than the current iteration, the result deteriorated, and the tabu length is decreased by 1 to encourage diversification.

Aspiration criteria was used to permit a swap move, even if it belonged in the tabu memory, as long as the swapped solution turn out to be better than the current best solution overall. The tabu memory element for that swap move was then revoked. A long term memory current best solution variable was used to keep track of best solution, which updated itself when a new best solution move was discovered.

Hand Iterations

Setup Initial TS parameters
tabu_length = 5
currentBestSol = inf
tabu_mem = zeros matrix
prev_iter_sol= 0

Goal: user input
Start location = Location_0
Buy Duck, Chicken, Apples

Generating Random Initial Solution:

Duck, Chicken, Apples bought at...
Store_0, Store_1, Store_2

Distance = Location_0, Store_0, Store_1, Store_2, Location_0
= 867 + 673 + 54 + 982
= 2576
Price = 536 + 359 + 40
= 935
ObjectiveFcn = 0.5*2576 + 0.5*935 = 1755.5

Tabu memory:

Table 1: TS – Initial Tabu List

	store 0	store 1	store 2	store 3	store 4	store 5	store 6	store 7	store 8	store 9
store 0	0	0	0	0	0	0	0	0	0	0
store 1	0	0	0	0	0	0	0	0	0	0
store 2	0	0	0	0	0	0	0	0	0	0
store 3	0	0	0	0	0	0	0	0	0	0
store 4	0	0	0	0	0	0	0	0	0	0
store 5	0	0	0	0	0	0	0	0	0	0
store 6	0	0	0	0	0	0	0	0	0	0
store 7	0	0	0	0	0	0	0	0	0	0
store 8	0	0	0	0	0	0	0	0	0	0
store 9	0	0	0	0	0	0	0	0	0	0

Iteration 1:

Swap visiting random stores by pairs, find the minimum objective function solution.

buy Chicken, duck, apples

Store_0, Store_1, Store_2

Distance = Location_0, Store_1, Store_0, Store_2, Location_0

= 606 + 673 + 242 + 982 = 2503

Price = 359 + 536 + 40 = 935

ObjectiveFcn = 0.5*2503 + 0.5*935 = 1719

Distance = Location_0, Store_2, Store_0, Store_1, Location_0

= 54 + 242 + 673+ 194 = 1163

Price = 40 + 536 + 359 = 935

ObjectiveFcn = 0.5*1163 + 0.5*935 = 1517

Distance = Location_0, Store_0, Store_2, Store_1, Location_0

= 375 + 242+ 54 + 194= 865

Price = 359 + 536 + 40 = 935

ObjectiveFcn = 0.5*865 + 0.5*935 = 900

top 3 shopping route objective function solution: 1719, 1517, 900.

Minimum value: 900, a swap between store_2, and store_1 from initial stores route.

insert tabu_length = 5 into tabu matrix at position (1,2) and (2,1)

update currentbestsol = 900

update prev_iter_sol =900

Table 2: TS – Iteration 1 Tabu List

	store 0	store 1	store 2	store 3	store 4	store 5	store 6	store 7	store 8	store 9
store 0	0	0	0	0	0	0	0	0	0	0
store 1	0	0	5	0	0	0	0	0	0	0
store 2	0	5	0	0	0	0	0	0	0	0
store 3	0	0	0	0	0	0	0	0	0	0
store 4	0	0	0	0	0	0	0	0	0	0
store 5	0	0	0	0	0	0	0	0	0	0
store 6	0	0	0	0	0	0	0	0	0	0
store 7	0	0	0	0	0	0	0	0	0	0
store 8	0	0	0	0	0	0	0	0	0	0
store 9	0	0	0	0	0	0	0	0	0	0

Iteration 2:

Randomly decide to buy Chicken at Store_3 instead of Store_1

Chicken, Duck, Apples bought at...

Store_3, Store_0, Store_2

Distance = Location_0, Store_3, Store_0, Store_2, Location_0

= 187 + 95 + 242 + 982

= 1506

Price = 325 + 536 + 40 = 901

ObjectiveFcn = 0.5*1506 + 0.5*901 = 1203.5

Distance = Location_0, Store_0, Store_3, Store_2, Location_0

= 375+ 95 + 586+ 54 = 1110

ObjectiveFcn = 0.5*1110 + 0.5*901 = 1005.5

Distance = Location_0, Store_2, Store_0, Store_3, Location_0

= 54+ 242+ 95+ 741= 1132

ObjectiveFcn = 0.5*1132 + 0.5*901 = 1016.5

Distance = Location_0, Store_3, Store_2, Store_0, Location_0

= 741+ 586+ 242+ 375= 1944

ObjectiveFcn = 0.5*1944 + 0.5*901 = 1422.5

top 3 shopping route objective function solution: 1203.5, 1005.5, 1016.5.

Minimum value: 1005.5, a swap between store_0, and store_3 from initial stores route.

insert tabu_length = 5 into tabu matrix at position (0,3) and (3,0)

decrement by 1 at tabu matrix element at position (1,2) and (2,1) down to 4

no change for currentbestsol = 900

tabu_length decreased by 1 to 4

update prev_iter_sol =1005.5

Table 3: TS – Iteration 2 Tabu List

	store 0	store 1	store 2	store 3	store 4	store 5	store 6	store 7	store 8	store 9
store 0	0	0	0	5	0	0	0	0	0	0
store 1	0	0	4	0	0	0	0	0	0	0
store 2	0	4	0	0	0	0	0	0	0	0
store 3	5	0	0	0	0	0	0	0	0	0
store 4	0	0	0	0	0	0	0	0	0	0
store 5	0	0	0	0	0	0	0	0	0	0
store 6	0	0	0	0	0	0	0	0	0	0
store 7	0	0	0	0	0	0	0	0	0	0
store 8	0	0	0	0	0	0	0	0	0	0
store 9	0	0	0	0	0	0	0	0	0	0

Ant Colony System

Algorithm and Implementation

Ants Colony System is an algorithm that has a similar behaviour as a real ants' colony in nature. Several tasks search for an optimal least-cost solution and then, after each iteration, we compare which task has given the better result. To simulate a pheromone that real ants use to choose a better path to a source of food, Ants Colony System algorithm uses pheromone vector or matrix to remember which choices resulted in a better solution. After each iteration previous results evaporate, whereas an iteration's best path is added more weight so that it's preferable to be selected by ants in next iterations. To avoid an intensification in local minimum, there are several parameters that we use to encourage exploration of new paths. Since we are dealing with a multivariable problem, there are two pheromone matrices being used. One is for distances, and another one is for prices. Distance pheromone matrix is NxN matrix where N is a total number of locations (including location_0, location_1). D_{ij} is a distance from store i to store j. Price pheromone matrix is PxN matrix where P is a total number of products a person wants to buy. P_{ij} is a price of a product i in store j. P = [Duck, Chicken, Apples, Pork, Grapes, Wings, Oranges]. Initially, these matrices are initialized to ones. Each iteration step, we evaporate previous values in matrices. Ant that has the route and products that lie in an iteration's best solution will add pheromones to corresponding values in a matrix. So, expensive solutions' routes and prices pheromone values are more likely to essentially converge to 0.

Adaptation

When an algorithm's best solution does not improve, parameters such as β and evaporation rate are changed to do more exploration rather than intensification. When a solution does not improve after several changes in parameters, the algorithm stops.

Hand Iterations

Let's assume we want to buy 7 items, and there are 10 stores available + 2 locations that we start our trip from.

So, in 1st step of the run of ACS, we have distance and price pheromone matrices that look like these:

$$\begin{bmatrix} 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&1 \end{bmatrix} \text{ is D and } \begin{bmatrix} 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1 \end{bmatrix} \text{ is a P}$$

ACS generates solutions incrementally. At each step it constructs a route base on a transition rule. A store is chosen depending on probability values. The probability $p_{ij,}$ where i is a store we are at, and j is a store we'd like to visit, is

$\frac{\tau_{ij}^{\alpha} * \mu_{ij}^{\beta}}{\sum_{u \in N} \tau_{iu}^{\alpha} * \mu_{iu}^{\beta}}$, where α is a parameter to control the influence of pheromone τ, η is the desirability of state transition from i to j. β is a parameter to control the influence of the desirability. For our problem there are two factors of desirability. One is a product's price, and another one is a distance between two stores. For price desirability we use $\frac{1}{price_{ij}}$, and similarly for the distance's desirability we use $\frac{1}{distance_{ij}}$

Other Parameters:

β = 1, evaporation rate ρ = 0.3 (the smaller value - the more exploration will happen).

Q_p = 1, and Q_d = 10 (since distances are usually large numbers we want to scale it up a little bit) are parameters used to compute delta for pheromone updates - $\Delta\tau_{ij} = Q/cost$.

For goal: start location = location_0, products to buy = Duck, Chicken, Apples, these are hand iterations.

Randomly generate an order of products we are buying – Chicken, Duck, Apples. Then, look at stores that sell Chicken. These are Store_1, Store_2, Store_3 and Store_8.

Compute desirability of prices and distances.

Distance(Location_0, Store_1) = 606, Price(Chicken, Store_1) = 359

Distance(Location_0, Store_2) = 982, Price(Chicken, Store_2) = 323

Distance(Location_0, Store_3) = 187, Price(Chicken, Store_3) = 325

Distance(Location_0, Store_8) = 449, Price(Chicken, Store_8) = 323

Distance_desirability(Location_0, Store_1) = 1/606 = 0.0016, Price_desirability(Chicken, Store_0) = 0.0028

Distance_desirability(Location_0, Store_2) = 1/982 = 0.0010, Price_desirability(Chicken, Store_1) = 0.0031

Distance_desirability(Location_0, Store_3) = 1/187 = 0.0054, Price_desirability(Chicken, Store_3) = 0.0031

Distance_desirability(Location_0, Store_8) = 1/449 = 0.0022, Price_desirability(Chicken, Store_8) = 0.0031.

Compute probabilities:

p(Location_0, Store_1) = $\frac{1*0.0016}{1*0.0165+1*0.0102+1*0.0535+1*0.0223} * \frac{1*0.0028}{1*0.0028+1*0.0031+1*0.0031+1*0.0031}$

= 0.1788 * 0.2314 = 0.0414

p(Location_0, Store_2) = $\frac{1*0.0102}{1*0.0165+1*0.0102+1*0.0535+1*0.0223} * \frac{1*0.0031}{1*0.0028+1*0.0031+1*0.0031+1*0.0031}$

= 0.1105 * 0.2562 = 0.0283

p(Location_0, Store_3) = $\frac{1*0.0535}{1*0.0165+1*0.0102+1*0.0535+1*0.0223} * \frac{1*0.0031}{1*0.0028+1*0.0031+1*0.0031+1*0.0031}$

= 0.5797 * 0.2562 = 0.1485

p(Location_0, Store_8) = $\frac{1*0.0223}{1*0.0165+1*0.0102+1*0.0535+1*0.0223} * \frac{1*0.0031}{1*0.0028+1*0.0031+1*0.0031+1*0.0031}$

= 0.2416 * 0.2562 = 0.0619

Calculate a factor of a probability:

Factor = $factor = \frac{1}{0.0414+0.0283+0.1485+0.0619} = 3.5702$

Random an integer (either 0 or 1) for r_0 (used to determine if a choice is greedy or not). If it's greedy we pick the best probability which is Store_3 (exploitation). If it's not, then we roll a number (b/w 0 and 1) and choose the probability that fits the number (exploration). So, p(store_1) becomes p(Location_0, Store_1) * factor = 0.1478, and so on. If we roll 0.15, then we choose Store_1. Let's say we chose Store_1 after we figured out that we choose to explore. After choosing Store_1, we add it to a route – [Location_0, Store_1]. Now, search for stores that have Duck. These are Store_0, Store_2, Store_4. Repeat previous steps:

Distance_desirability(Store_1, Store_0) = 1/673 = 0.0015, Price_desirability(Duck, Store_0) = 1/536 = 0.0019

Distance_desirability(Store_1, Store_2) = 1/54 = 0.0185, Price_desirability(Duck, Store_1) = 1/488 = 0.0021

Distnace_desirability(Store_1, Store_4) = 1/65 = 0.0154, Price_desirability(Duck, Store_3) = 1/481 = 0.0021

p(Store_1, Store_0) = $\frac{1*0.0015}{1*0.0015+1*0.0185+1*0.0154}$ * $\frac{1*0.0019}{1*0.0019+1*0.0021+1*0.0021}$ = 0.0421 * 0.3114 = 0.0131

p(Store_1, Store_2) = $\frac{1*0.0185}{1*0.0015+1*0.0185+1*0.0154}$ * $\frac{1*0.0021}{1*0.0019+1*0.0021+1*0.0021}$ = 0.5233 * 0.3443 = 0.1802

p(Store_1, Store_4) = $\frac{1*0.0154}{1*0.0015+1*0.0185+1*0.0154}$ * $\frac{1*0.0021}{1*0.0019+1*0.0021+1*0.0021}$ = 0.4346 * 0.3443 = 0.1496

Suppose we choose to pick greedy this time. We pick Store_2 now.

Search for stores with Apples – Store_2, Store_7.

Distance_desirability(Store_2, Store_2) = 1/1 = 1 (if the store is already in a route we use 1 for a value of distance to avoid a division by zero), Price_desirability(Apples, Store_2) = 1/40 = 0.025

Distance_desirability(Store_2, Store_7) = 1/5 = 0.2, Price_desirability(Apples, Store_1) = 1/49 = 0.02

p(Store_2, Store_2) = $\frac{1*1}{1*1+1*0.2}$ * $\frac{1*0.025}{1*0.025+1*0.02}$ = 0.8333 * 0.5556 = 0.463

p(Store_2, Store_7) = $\frac{0.2*1}{1*1+1*0.2}$ * $\frac{1*0.025}{1*0.025+1*0.02}$ = 0.1667 * 0.4444 = 0.0741

$$\begin{bmatrix} 0.3 & 0.3 & 0.3061 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3061 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3061 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix}$$

We are likely to pick Store_2 for its probability. Our route became – [Location_0, Store_1, Store_2, Location_0]. Distance price of a cost is 606 + 54 + 982 = 1642. Price cost is 359 + 488 + 40 = 887. Total cost is 1642*0.5 + 887*0.5 = 1264.5.

Instead of repeating all the steps above for the 2nd ant, assume 2nd ant's order of products is Duck, Apples, Chicken and the route it chose is – [Location_0, Store_4, Store_2, Store_1, Location_0]. Distance price of a cost is 921 + 606 + 54 + 606 = 2187. Price cost is 481 + 40 + 359 = 880. Total cost is 1533.5. Since ant 1 gave us the best solution we choose it to deposit pheromone.

Update Pheromone matrix:

Multiply matrix by (1 – ρ). Then, out of all ants pick the best solution and deposit some pheromone onto the corresponding values. The additional deposit is $\frac{Q_p}{\text{price\_cost\_of\_a\_route}}$ = 1/887 = 0.0011, $\frac{Q_d}{\text{distance\_cost\_of\_a\_route}}$ = 10/1642 = 0.0061.

[Duck, Chicken, Apples, Pork, Grapes, Wings, Oranges]

$$\begin{bmatrix} 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3011 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3011 & 0.3 \\ 0.3 & 0.3011 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3011 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3011 & 0.3011 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 & 0.3 \end{bmatrix}$$ is D and

After that, the algorithm continues in a similar fashion for a supplied number of iterations.

Particle Swarm Optimization

Algorithm and Implementation

The particle swarm optimization is based on the behaviour of a colony or swarm of particles (such as insects, birds, and fish). It mimics the behaviour of these social organisms. Each particle attempts to emulate the success of the neighbouring individuals and once a particle finds a good path, the rest of the swarm will be able to follow it. The permutation particle swarm optimization algorithm (synchronous) for two dimensions is used to solve our problem. First the parameter values are set (to their default values or modified by the user) followed by the parameters for the adaptation, which are not tweaked. For each of the particles, random initial solutions are generated which randomly pick a store to purchase each item from and a random route between those stores. We track the local best for each particle along with the global best and use those in the main loop to find the next solutions. Once the termination criteria are met, the loop ends along with the algorithm. The fitness value of each particle is calculated during the main loop of each iteration and if the fitness value is better than the current local best, the local best is updated. Then the global best is updated to be the minimum of all the local best solutions.

Within the main loop, for each iteration, there are two velocity vectors that are generated using the operators defined below. The first velocity vector aims to change the stores where the items are purchased towards the global or local best while the second velocity vector aims to change the order in which the stores are visited. This is done for each particle. After all the particles have generated new solutions, the global best is updated (synchronous). At the end of the iteration, the inertia weight is adapted based on the current condition of the algorithm, as specified in the adaptation section.

Operators

Since there are two dimensions to this problem, for the PSO algorithm, there are two velocity function calculations. The store modification utilizes the usual mathematical operators with probabilities to decide whether or not to investigate purchasing items from different stores. The route modification utilizes the permutation PSO operators which are redefined from the normal semantics.

The velocity1 of a particle is defined as a set of swaps to be performed on the solution which translates to bringing in different stores (randomly) to purchase certain items from.

We have the FindDiff function (synonymous with subtraction) and the GetDiffStore function (synonymous with addition). Multiplication is the same as the arithmetic operation. The FindDiff function compares the current solution and the local/global best solution to see which stores are different and which match. The objective here is to move the particle towards either the global best or the local best solution. The GetDiffStore function uses the velocity vector and modifies the particle to add different stores depending on that velocity vector.

The velocity2 of a particle is defined as a set of swaps to be performed on the solution which translates to modifying the order in which the stores are visited. We have the Adding, Subtracting, and Multiply functions. The Adding function adds a velocity (set of swaps) to a position (store list). The Subtracting function compares two positions (store lists) and produces a velocity (set of swaps) which would transform position1 into position2. The Multiply function changes the length of the velocity vector according to the constant c. If c is 0, the length is set to 0. If c is less than 1, the velocity is truncated to the new size (which equals size*c), and if c is more than 1, then the velocity is augmented to the new size (which equals size*c).

Parameters and Tuning

For the PSO algorithm, these are the tunable parameters:

- Inertia weight - default value set to 0.6, min 0.5, max 0.9
- Cognitive acceleration coefficient - default value set to 1.1, min 0.5, max 3.0
- Social acceleration coefficient - default value set to 2.9, min 1.0, max 3.5

- The number of particles - default set to 10, min 5, max 50
- The maximum number of iterations - default set to 5000, min 1000, max 10000

The inertia weight impacts the inertia component of the velocity calculation and balances the exploration and exploitation of the algorithm. The c1 and c2 (cognitive and social acceleration coefficients) balance the weights to trust the particle's own experience versus the swarm experience. The maximum number of iterations impact the termination conditions along with the adaptation parameters. The number of particles is the same as the swarm and neighbourhood size as there is only one neighbourhood, with no cooperation. A value of 10 particles was chosen to allow for a relatively quick iteration while still giving improving results. Through experimentation, a default value of 5000 was chosen for the maximum number of iterations. If modified too much, it would result in very early convergence on a poor solution or take unnecessarily long. The inertia weight was set to 0.6 (slightly lower than the usual 0.792) to allow for reasonable probability of convergence within the maximum iterations. The acceleration coefficient always sum to 4 and hence only one of these (cognitive) is tunable by the user. There was more emphasis on the social acceleration coefficient (leaning towards a selfless model) to allow for reasonable exploration but more frequent convergence. This is a side effect of implementing a permutation based PSO with multiple variables and hence some inherent exploration is built within the second velocity function (which modifies the route).

Adaptation

The main parameter that is updated as part of the adaptation for the PSO algorithm is the inertia weight, w. The inertia component accommodates the fact that a particle cannot suddenly change its direction or movement. The value w is important to balance exploration and exploitation and 0.6 is used as the recommended starting value for convergence. Large values of w promote exploration and small promote exploitation (allowing more control to cognitive and social components). If a solution is not improving for some iterations, we adapt the inertia weight and increase it by 0.1 (to a maximum of

0.9), so that we explore a larger search space. However, once the algorithm runs for several iterations, the inertia weight is decreased by 0.1 (to a minimum of 0.5), in an attempt to converge to a solution by intensifying. Each adaptation to the weight criteria is done every tenth of the termination condition (max number of iterations solution is not improving divided by 10).

Termination Criteria

The algorithm terminates under two conditions. The maximum number of iterations that the algorithm will run for is a modifiable parameter (recommended between 1000 and 10000). This is one of the termination condition for the algorithm. The other is if no solution has been accepted for a large number of iterations even with adapting the inertia weight, the algorithm stops. If the solution does not improve for a quarter of the maximum number of iterations, that implies we have converged to a relatively optimal solution and can stop early.

Hand Iterations

For solving the reduced problem, the user is chosen to start at location_0 and is purchasing: Duck, Chicken, and Apples

- The inertia weight is set to 0.6
- The cognitive acceleration coefficient is 1.1
- The social acceleration coefficient is 2.9
- There are two particles and two iterations

The initial particle solutions are chosen randomly.

Particle 1

Table 4: PSO – Current solution for Particle 1

Current solution:	
Start at	Location_0
Apples	Store_2
Duck	Store_0
Chicken	Store_1
End at	Location_0
Cost	1719

This is the local best for Particle 1. Cost = 0.5*(982+242+673+606)+0.5*(536+359+40)

Particle 2

Table 5: PSO – Current solution for Particle 2

Current solution:	
Start at	Location_0
Duck	Store_4
Chicken	Store_3
Apples	Store_2
End at	Location_0

This is the local best for Particle 2. Cost = 0.5*(921+39+586+982)+0.5*(325+481+40)

The global best is the minimum of all the local bests, i.e. cost of 1687. Hence, the global best is:

Table 6: PSO – Current global best

Start at	Location_0
Duck	Store_4
Chicken	Store_3
Apples	Store_2
End at	Location_0

ITERATION 1

The first set of random values is for the store selection (for velocity1).

The second set of random values is for the route selection (for velocity2).

rand11	0.3	c1*rand11	0.33
rand12	0.9	c2*rand12	2.61
rand21	0.4	c1*rand21	0.44
rand22	0.7	c2*rand22	2.03

Particle 1

inertia = inertia weight multiplied by previous velocity = 0.6*[0] = [0]

cognitive = difference between current solution and local best solution = c1*rand1*[0] = 0.33*[0] = [0]

social = difference between current solution and global best solution

Apples	Store_2	vs	Duck	Store_4
Duck	Store_0		Chicken	Store_3
Chicken	Store_1		Apples	Store_2

The shopping list differs for duck and chicken but matches the global best for apples

Therefore, social = [1, 1, 0]

social = social*c2*rand2 = [1, 1, 0]*2.61 = [2.61, 2.61, 0]

velocity1 = (1/3)*inertia + (1/3)*cognitive + (1/3)*social

velocity1 = [0] + [0] + (1/3)*[2.61, 2.61, 0]

velocity1 = [0.87, 0.87, 0]

Add the velocity to the current solution
Random to find different store: [0.7, 0.5, 0.3]
We look for new stores if random is less than velocity. This is true at index 1 since 0.7 < 0.87 and at index 2 since 0.5 < 0.87.
Therefore, we look for new stores for Duck and Chicken only.

We randomly pick store_2 to purchase duck and store_2 to purchase chicken from for the next solution.

Solution	Apples	Store_2
	Duck	Store_2
	Chicken	Store_2

inertia = Multiple(velocity, 0.6) = [0]*0.6 = [0]

cognitive = Multiply(difference between local best and current solution, c1*rand1) = Multiply([0], 0.44) = [0]

social = Multiply(difference between global best and current solution, c2*rand2)

Apples	vs	Duck
Duck		Chicken
Chicken		Apples

swap list = swaps required to go from the current solution to the global best = [1, 3; 1, 2]

social = Multiply([1, 3; 1, 2], 2.03) = [1, 3; 1, 2; 1, 3; 1, 2]

velocity2 = [1, 3; 1, 2; 1, 3; 1, 2]

Add the velocity to the current solution (i.e. do the swap operations)

Solution	Chicken	Store_2
	Apples	Store_2
	Duck	Store_2
	Cost	1407.5

This is the new local best for Particle 1

Particle 2

inertia = inertia weight multiplied by previous velocity = 0.6*[0] = [0]

cognitive = difference between current solution and local best solution = c1*rand1*[0] = 0.33*[0] = [0]

social = difference between current solution and global best solution = c2*rand2*[0] = 2.61*[0] = [0]

Duck	Store_4	vs	Duck	Store_4
Chicken	Store_3		Chicken	Store_3
Apples	Store_2		Apples	Store_2

The global best is equal to the current solution which equals the local best. Hence, the velocity is 0.

velocity1 = [0]

Therefore, our next solution has not changed.

Solution	Duck	Store_4
	Chicken	Store_3
	Apples	Store_2

inertia = Multiple(velocity, 0.6) = [0]*0.6 = [0]

cognitive = Multiply(difference between local best and current solution, c1*rand1) = Multiply([0], 0.44) = [0]

social = Multiply(difference between global best and current solution, c2*rand2) = Multiple([0], 2.03) = [0]

Duck	vs	Duck
Chicken		Chicken
Apples		Apples

velocity2 = [0]
Therefore, our next solution has not changed.

Solution	Duck	Store_4
	Chicken	Store_3
	Apples	Store_2
	Cost	1687

This is the new local best for Particle 2 (same as before).

The global best is the minimum of all the local bests, i.e. cost of 1407.5. Hence, the global best is:

Table 7: PSO – Global best solution at the end of iteration 1

Start at	Location_0
Chicken	Store_2
Apples	Store_2
Duck	Store_2
End at	Location_0

ITERATION 2

The first set of random values is for the store selection (for velocity1).
The second set of random values is for the route selection (for velocity2).

rand11	0.8	c1*rand11	0.88
rand12	0.5	c2*rand12	1.45
rand21	0.9	c1*rand21	0.99
rand22	0.2	c2*rand22	0.58

Particle 1
inertia = inertia weight multiplied by previous velocity = 0.6*[0.87, 0.87, 0] = [0.522, 0.522, 0]
cognitive = difference between current solution and local best solution = c1*rand1*[0] = 0.88*[0] = [0]
social = difference between current solution and global best solution = c2*rand2*[0] = 1.45*[0]= [0]

Chicken	Store_2	vs	Chicken	Store_2
Apples	Store_2		Apples	Store_2
Duck	Store_2		Duck	Store_2

The global best is equal to the current solution which equals the local best. Hence, the velocity is 0.
velocity1 = [0]
Therefore, our next solution has not changed.

Solution	Apples	Store_2
	Duck	Store_2
	Chicken	Store_2

inertia = Multiple(velocity, 0.6) = [1, 3; 1, 2; 1, 3; 1, 2]*0.6 = [1, 3; 1, 2]

cognitive = Multiply(difference between local best and current solution, c1*rand1) = Multiply([0], 0.99) = [0]

social = Multiply(difference between global best and current solution, c2*rand2) = Multiple([0], 0.58) = [0]

Apples	vs	Apples
Duck		Duck
Chicken		Chicken

velocity2 = [1, 3; 1, 2]

Add the velocity to the current solution (i.e. do the swap operations)

Solution	Apples	Store_2
	Duck	Store_2
	Chicken	Store_2
	Cost	1407.5

This is the new local best for Particle 1 (same as before).

Particle 2

inertia = inertia weight multiplied by previous velocity = 0.6*[0] = [0]

cognitive = difference between current solution and local best solution = c1*rand1*[0] = 0.88*[0] = [0]

social = difference between current solution and global best solution

Duck	Store_4	vs	Apples	Store_2
Chicken	Store_3		Duck	Store_2
Apples	Store_2		Chicken	Store_2

The shopping list differs for duck and chicken but matches the global best for apples
Therefore, social = [1, 1, 0]
social = social*c2*rand2 = [1, 1, 0]*1.45 = [1.45, 1.45, 0]
velocity1 = (1/3)*inertia + (1/3)*cognitive + (1/3)*social
velocity1 = [0] + [0] + (1/3)*[1.45, 1.45, 0]
velocity1 = [0.483, 0.483, 0]

Add the velocity to the current solution
Random to find different store: [0.6, 0.4, 0.9]
We look for new stores if random is less than velocity. This is true at index 2 since 0.4 < 0.483.
Therefore, we look for new stores for Chicken only.

We randomly pick store_8 to purchase chicken from for the next solution.

Solution	Duck	Store_4
	Chicken	Store_8
	Apples	Store_2

inertia = Multiple(velocity, 0.6) = [0]*0.6 = [0]
cognitive = Multiply(difference between local best and current solution, c1*rand1) = Multiply([0], 0.99) = [0]
social = Multiply(difference between global best and current solution, c2*rand2)

Duck	vs	Apples
Chicken		Duck
Apples		Chicken

swap list = swaps required to go from the current solution to the global best = [1, 3; 2, 3]
social = Multiply([1, 3; 2, 3], 0.58) = [1, 3]

velocity2 = [1, 3]

Add the velocity to the current solution (i.e. do the swap operations)

Solution	Apples	Store_2
	Chicken	Store_8
	Duck	Store_4
	Cost	2015.5

The local best for Particle 2 remains the same as before.

The global best is the minimum of all the local bests, i.e. cost of 1407.5. Hence, the global best is:

Table 8: PSO – Global best solution at the end of iteration 2

Start at	Location_0
Chicken	Store_2
Apples	Store_2
Duck	Store_2
End at	Location_0

Artificial Bee Colony

Artificial Bee Colony is a population based metaheuristic algorithm. It works by having groups of bees called employees, onlookers, and scouts. The algorithm is initialized with the number of employees and onlookers and each employee is assigned a randomly generated solution. As the solutions for the bee age without improvement, the solutions are abandoned and new solutions are randomly generated to explore new search spaces.

Algorithm and Implementation

At each iteration, employed bees try to find another solution in the area of their currently stored solution by using a neighbourhood operator. They compare that new solution with their stored solution and pick the solution with the better cost. Each onlooker bee then looks at all the solutions the employed bees hold and selects one at random using a roulette wheel based on the fitness score of each solution. Solutions with a better fitness have a higher probability of being selected by an onlooker bee. Each onlooker bee then acts similarly to an employee in that it tries to improve the solution that was taken through a neighbourhood operator. If the solution was improved, the age of the solution is reset.

After all onlookers are finished, the algorithm checks if any solutions have not been improved by any onlooker for the age limit. If there are, the employed bee discards the stored solution and becomes a scout. They randomly generate a new solution and become an employed bee again. All solutions are then aged before the beginning of the next iteration.

New solutions by scouts are completely randomly generated. They find random permutations of stores and items to be bought. Each employed bee and onlooker bee generates a new solution in the neighbourhood by rolling a random number between [-1, 1]. Negative numbers will cause the bee to swap the buying order route. Positive values will make the bee purchase items from different stores.

Adaptation

The algorithm adapts during the execution by changing the number of onlookers and the age limit threshold. When the algorithm detects that the number of abandoned solutions is too high, the number of onlooker bees and the age limit is increased in an attempt to help explore the worse solutions. When the algorithm detects that the number of explored solutions is too high, onlooker bees are removed and the age limit is decreased to help focus on the elite solutions.

Termination Criteria

The algorithm terminates under the fixed condition of hitting the maximum number of runs.

Chapter 5: Performance Evaluation

The algorithms are evaluated using the following metrics: performance, cost/time per iteration, number of iterations, and total time. To ensure performance, each solution was evaluated by the same evaluation function, which removes any store duplicates from the list. A problem set was constructed with real stores and prices around the waterloo area, determined through flyers of the stores and distances generated by google maps. The algorithms were tested on the problem set. Because of the stochastic nature of the algorithms, each algorithm is run 5 times and the results for each run are shown and averaged. The following list of items were purchased (with the quantity of each items purchased) with the user starting at the University of Waterloo:

Table 9: Real user's shopping list

Item	**Quantity**
fish fillet	5
Astro yogurt	10
boneless pork chop	1
shredded cheese	1
juice	5
coffee	1
grapes	1
Post cereal	1
Pepsi	6
cheese bar	1
PC chicken breast	1
entrée	1
water	6
salsa	1
salad	2

Simulated Annealing

A sample run of the Simulated Annealing algorithm is shown in Figure 1. The graph shows the currently selected solution that is used for generating the new solution. In the initial iterations, the graph is unstable and has many spikes. This is due to the high initial temperature and acceptance probability of non-improving solutions in the initial phases. As the algorithm progresses, it stabilizes as the temperature drops. Improvements to the solution cost seem to end at 25000 iterations, afterwards it seems like the algorithm has converged.

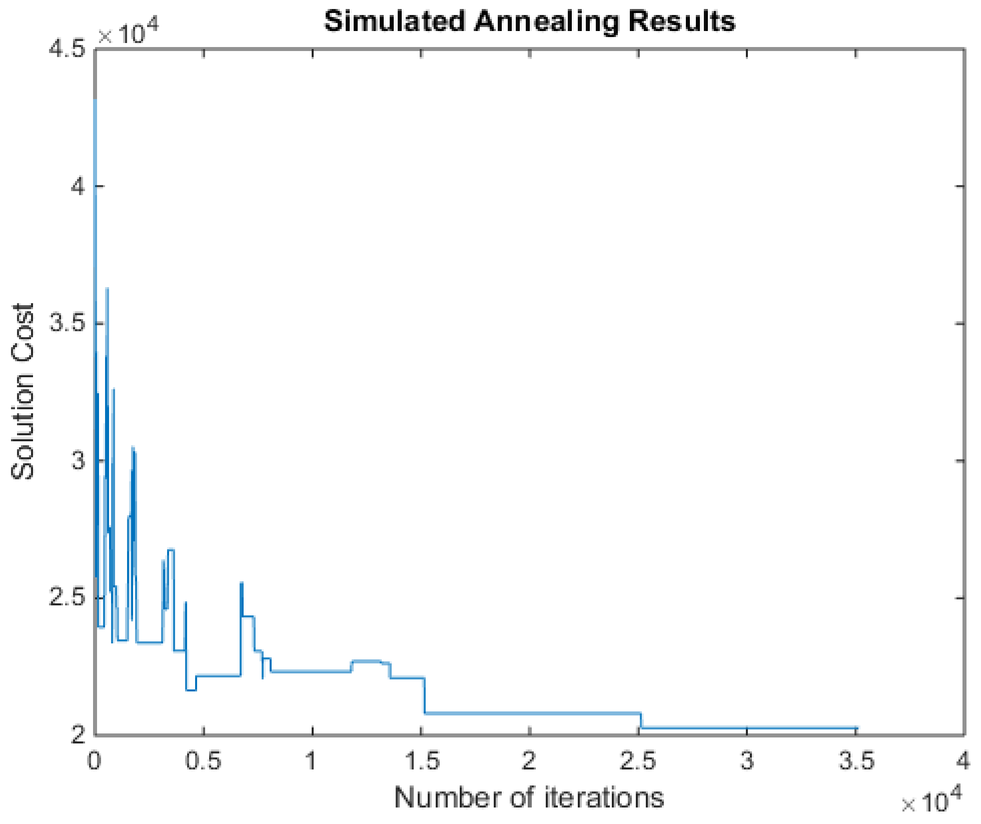


Figure 1: Graph of Simulated Annealing Solution Cost vs Number of Iterations

Table 10 was generated after running the algorithm 5 times. For simulated annealing, the cost per iteration is very low. However, the number of iterations required is quite high, leading the total time taken to be average. In general the solution is fairly consistent at around 20000. The best cost solution is 19287, shown in Appendix 1.

Table 10: SA – Experimental Results for Real Data

	cost per iteration(s)	number of iterations	best solution cost	total time (s)
	0.003995	31363	19287	125.2952
	0.003999	35140	20278	140.5249
	0.004002	21775	20418	87.14355
	0.004013	30891	19778	123.9656
	0.003999	27565	19757	110.2324
Average	**0.004002**	**29346.8**	**19903.6**	**117.4323**

Genetic Algorithm

A sample run of the GA with a population of 20 is shown in Figure 2 below. The blue line shows the best solution cost, while the red line shows the average cost of the population for each generation. The average solution quickly improves over the first 50 solutions. While the best solution improves quickly over the first 150 generations and less afterwards. Because of the adaptive algorithm which is set to increase mutation rates with extended periods where the best solution does not increase, the search diversifies more after 200 generations, meaning that the average solution cost increases.

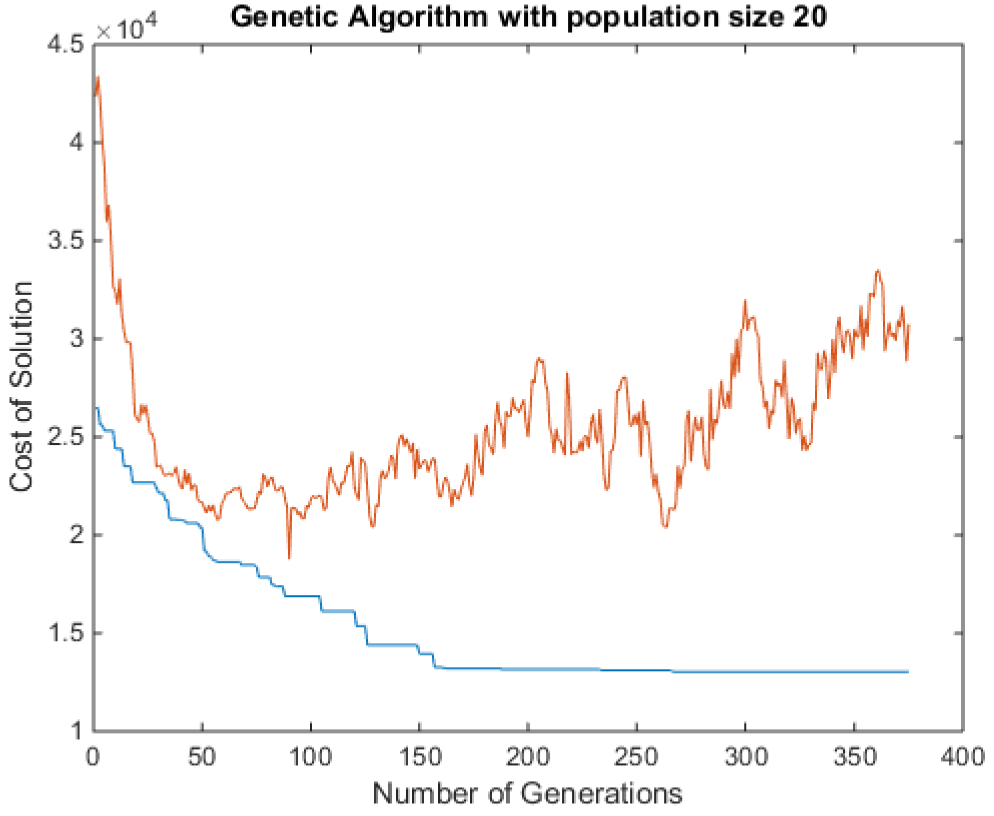


Figure 2: Graph of Genetic Algorithm Solution Cost vs Number of Generations

The genetic algorithm had fairly good performance. With a population of 20, the best solution was found with a cost of 12689, with an average of 14355.3 over 5 runs, as shown in Table 11. The average time was 25.54, and the average number of iterations was 353. The best solution found is shown in Appendix 2.

Table 11: GA – Experimental Results for Real Data with Population of 20

	cost per generation (s)	number of generations	best solution cost	total time (s)
	0.0782	394	14227	30.8108
	0.0692	396	14333.5	27.4032
	0.0781	289	12689	22.5709
	0.0682	372	14064.5	25.3704
	0.0685	315	16462.5	21.5775
Average	**0.07244**	**353.2**	**14355.3**	**25.54656**

The solution is somewhat worse with a population of 10, as shown in Table 12. The best solution found was at 14188.5, while the average solution was 17047.3. However, the average time was much quicker, at 11.6s. The solution for a population of 20 is chosen for comparison, since the results were significantly better.

Table 12: GA – Experimental Results for Real Data with Population of 10

	cost per generation (s)	number of generations	best solution cost	total time (s)
	0.0359	409	14188.5	14.6831
	0.0395	327	19681.5	12.9165
	0.0342	308	16999	10.5336
	0.0358	256	19898.5	9.1648
	0.0346	306	14469	10.5876
Average	**0.036**	**321.2**	**17047.3**	**11.6**

Tabu Search

A sample is shown in Figure 3, where it is clear that improvements are made fairly rarely. The Tabu search is terminated a certain number of runs if other criteria are not met. In this case, 100 and 200 was chosen because little improvement was found over larger iteration numbers. As seen in Table 13, the solutions widely vary, with relatively suboptimal solutions. This is likely due to the problem size, since TS is best suited for smaller sets of problems. As the problem size increases, TS must also increase its tabu memory size. Also, at every iteration, the algorithm needs to maintain short term memory to find minimal solution in our shopping problem from all possible neighbourhood swap operations. The best solution found in 5 runs is 29714, shown in Appendix 3.

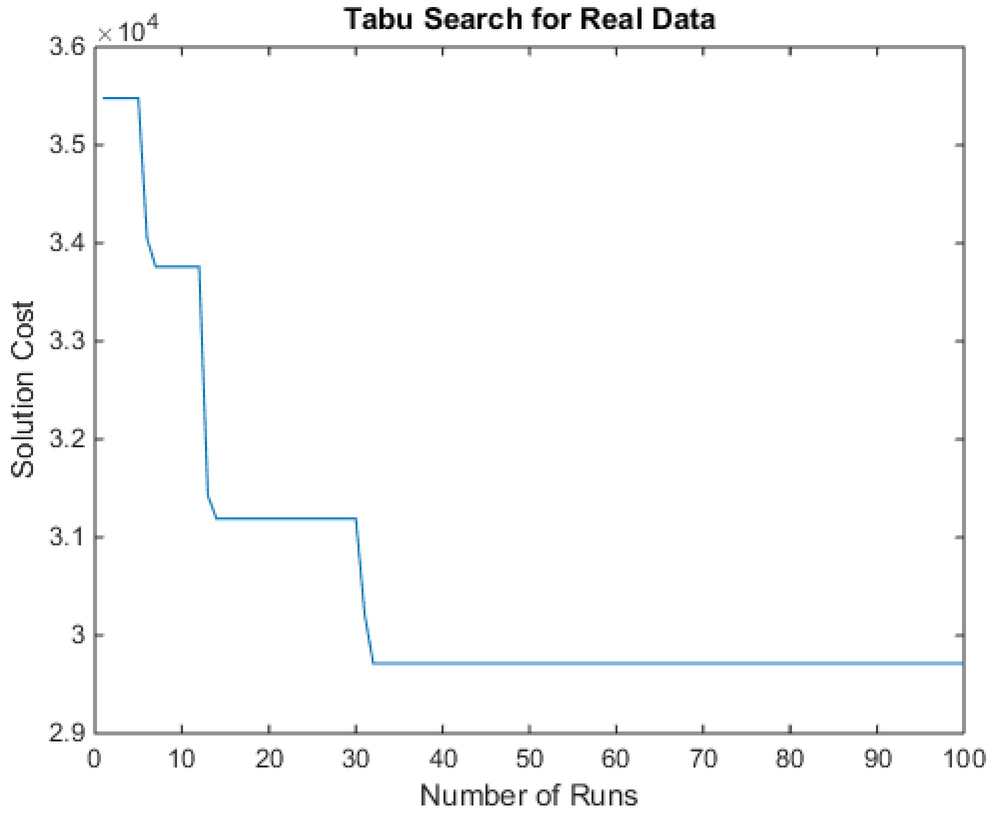


Figure 3: Graph of Tabu Search Solution Cost vs Number of Runs

Table 13: TS – Experimental Results for Real Data with Maximum Iterations 100

	cost per iteration(s)	number of iterations	best solution cost	total time (s)
	0.3746	100	29714	37.66
	0.4035	100	41134	36.46
	0.3632	100	33448	37.49
	0.4149	100	44329	41.49
	0.3775	100	26115	37.75
Average	**0.38674**	**100**	**36072**	**38.17**

Table 14: TS – Experimental Results for Real Data with Maximum Iterations 200

	cost per iteration(s)	number of iterations	best solution cost	total time (s)
	0.3766	200	35387	75.32
	0.3646	200	46778	72.92
	0.3749	200	33973	74.98
	0.3882	200	43122	77.64
	0.3951	200	41254	79.02
Average	**0.37988**	**200**	**40102.8**	**75.976**

Furthermore, running at high number of iterations didn’t necessarily improve best result, and sometimes the result is ~6500 after 1000 iterations for example. Therefore, TS is not suitable for large size problem running for high number of iterations due to memory and performance constraints.

Ant Colony System

A sample run of the ACS is shown in Figure 4. For ACS, the best solution cost is very good, and furthermore a similar value for the best solution is reached for each run. However, the time taken is over 400s, which means that the algorithm takes over 10 minutes to fully run. The best solution has a solution cost of 12800, shown in Appendix 4. ACS starts with exploring different candidates to pick from. The ant with the best solution deposits pheromone to an evaporated matrix. So, as you see from the figure, eventually the algorithm converges when it tends to pick a better route.

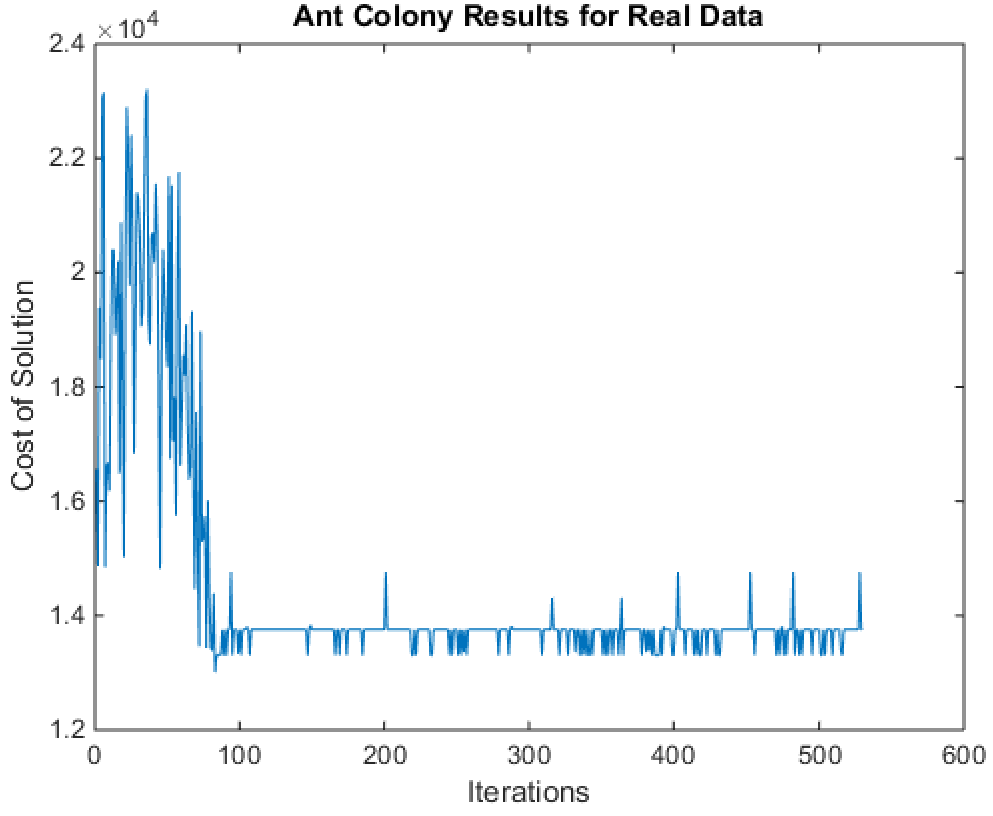


Figure 4: Graph of Ant Colony Optimization Solution Cost vs Number of Iterations

Table 15: ACS – Experimental Results for Real Data

	cost per iteration (s)	number of iterations	best solution cost	total time (s)
	0.8412	531	12800	446.7
	0.8235	543	12851	446.6
	0.79688	522	12925	416.0
	0.8336	485	13008	404.3
	0.7969	533	13249	424.8
Average	**0.8182**	**522.8**	**12966.6**	**427.7**

Particle Swarm Optimization

A sample run of PSO is shown below in Figure 5. The best solution is shown in blue, while the average cost per generation is shown in red. The solution fluctuates heavily from 0 to around 1700 generations, but afterwards the PSO converges to the best solution found. The initial fluctuations occurs since the inertia weight is increased every 100 iterations to explore more of the search space. After a thousand iterations, the inertia weight starts to drop to allow for convergence to a near-optimal solution. Table 16 shows the results after running the program 5 times.

Particle Swarm Optimization Results with Real Data

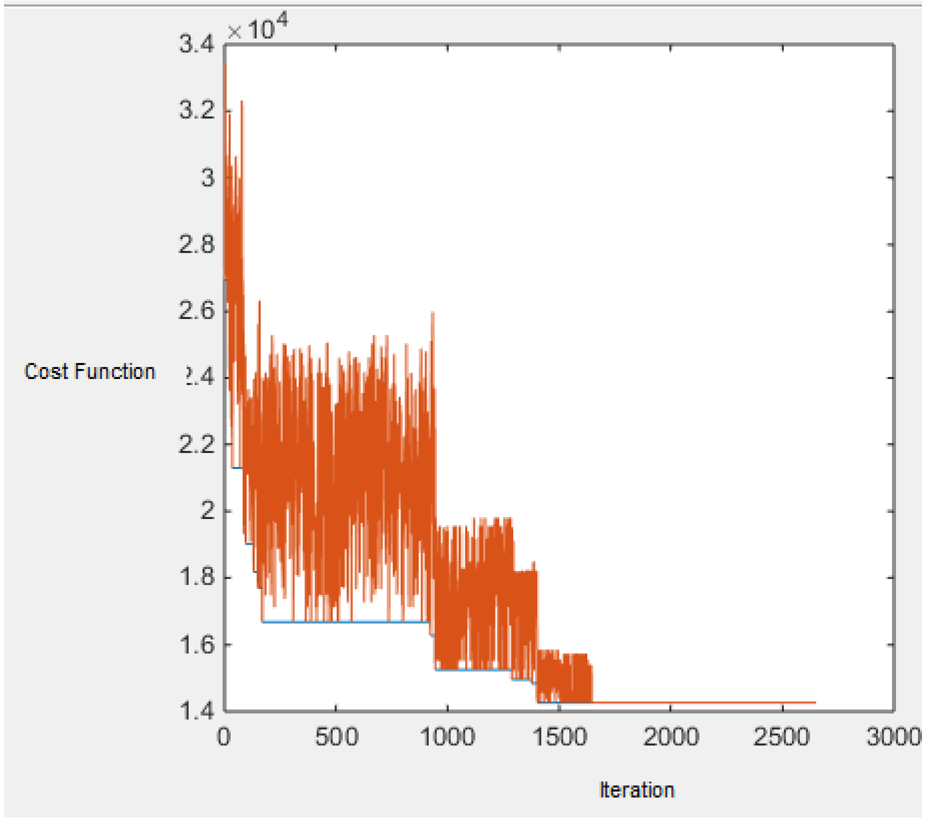


Figure 5: Graph of Particle Swarm Optimization Solution Cost vs Number of Iterations

Table 16: PSO – Experimental Results for Real Data with Population of 10

	cost per iteration(s)	number of iterations	best solution cost	total time (s)
	0.079499	4861	12451.5	386.4446
	0.084824	3646	13272	309.2683
	0.081477	4048	14188	329.8189
	0.080456	2173	15037	174.8309
	0.074017	3227	13071	238.8529
Average	**0.080055**	**3591**	**13603.9**	**287.8431**

Artificial Bee Colony

The graph in Figure 6 shows the optimal solution so far in the iterations of the bee colony algorithm. The colony finds many improving solutions initially. After around 200 iterations, it begins to taper off until a new discovery around 350 iterations. Afterwards, it stagnates and cannot find a better solution.

Artificial Bee Colony Results with Real Data

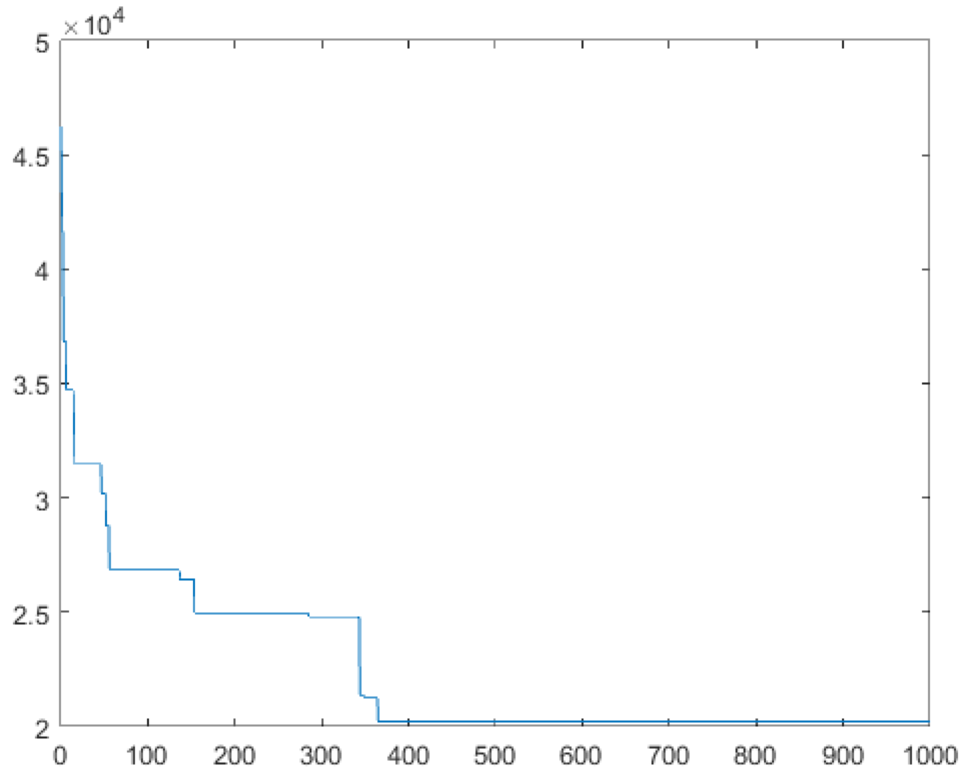


Figure 6: Graph of Artificial Bee Colony Solution Cost vs Number of Iterations

Table 17: ABC – Experimental Results for Real Data

	cost per generation (s)	number of generations	best solution cost	total time (s)
	0.07816	1000	25446	78.16
	0.09221	1000	19359	92.21
	0.08225	1000	22245	82.25
	0.08349	1000	22268	83.49
	0.07655	1000	23089	76.55
Average	**0.082532**	**1000**	**22481.4**	**82.532**

Result Comparison

Because of the different costs per generation/iteration for the algorithms, the number of generations and the cost can be misleading. For example, the simulated annealing cost per generation is only 4 milliseconds, however the number of iterations required is an order of magnitude higher than the other algorithms

However, the total time can be compared between the different algorithms, as well as the average best solution cost, which are shown in Table 18. The best solution here is the ACS, with an average cost of 12966.8, which is significantly better than all the other solutions. However, the ACS is also the slowest algorithm, taking an average of 411.7 seconds to finish. PSO is second in solution cost, and takes somewhat less time at 287.8 seconds. In terms of total time, GA is the best with only a 25.54s runtime and a reasonable solution of 14355.3. SA and TS, though are relatively faster than ACS and PSO, yielded worst costs with 19903.6 for SA and 36072.0 for TS. Surprisingly though, bee colony did not yield one of the best results for this problem set. However, it did show promising performance and results in some of the random data sets.

Table 18: Results Comparison for All Six Algorithms

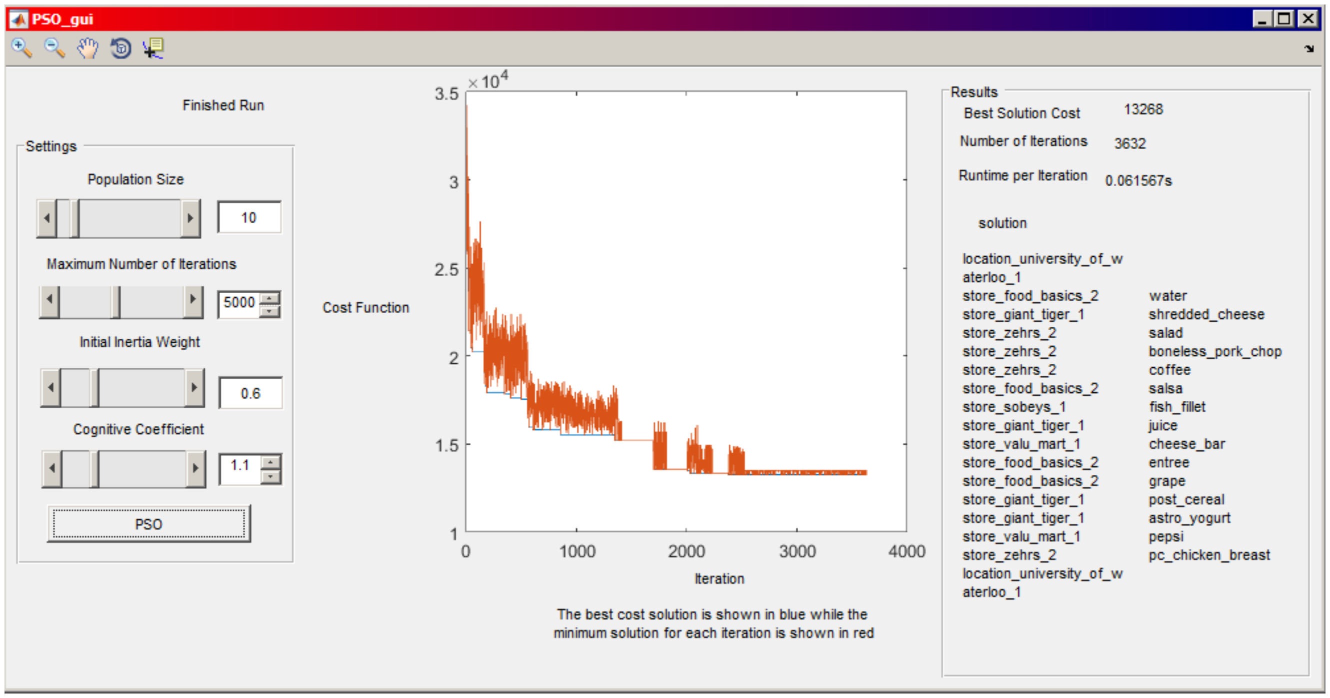
	cost per iteration(s)	number of iterations	best solution cost	total time (s)
ACS	0.8182	522.8	**12966.6**	427.7
GA	0.07244	353.2	14355.3	**25.54656**
PSO	0.08006	3591	13603.9	287.8
Bee Colony	0.08253	1000	22481.4	82.53
Tabu	0.3867	**100**	36072.0	38.2
SA	**0.004002**	29346.8	19903.6	117.4

Chapter 6: GUI and Usability Guide

Each of the algorithms is included as a component in their respective Graphical User Interface (GUI) which gives the user the ability to modify and tweak various parameters that relate to that algorithm before running it on the real problem data set. This makes the algorithms more interactive and by providing this modifiability, the user can appreciate the inner workings of the algorithms even more, without necessarily diving into the implementation details.

To run each algorithm, navigate to the folder for that algorithm in MATLAB and run the <algorithm name>_gui.m program (for example PSO_gui.m).

An example GUI of the particle swarm optimization is shown below in Figure 7. Each GUI is composed of three panels. The left panel contains the sliders and text boxes which allow the user to modify the different parameters within a restricted range for each of the algorithms before running them. These parameters are set to a default value and hence, the user can simply hit the button to start executing the algorithm (in this case, the 'PSO' button). The middle panel displays the graph of the algorithm performance and updates in real time. The x-axis shows the generation/iteration number while the y-axis shows the best cost that the algorithm has generated so far. In certain views, you see both the global best solution along with the best solution for each iteration as the algorithm evolves. Once the algorithm run is completed (based on the termination criteria and the maximum number of iterations), the results are displayed in the right panel. The results show the best solution cost, the number of iterations it took to reach that solution, how long it takes for each iteration to complete, and the problem solution itself in the form of a route which starts from the starting location, followed by a list of stores to visit, and ends with the starting location. It also provides which item to purchase at each of the stores visited. Even though duplicate stores are listed, the solution is optimized by ensuring that a store is only visited once and all the items that need to be purchased from that store are bought then before moving to the next unique store.



Finished Run

Settings

Population Size: 10

Maximum Number of Iterations: 5000

Initial Inertia Weight: 0.6

Cognitive Coefficient: 1.1

Results

Best Solution Cost 13268

Number of Iterations 3632

Runtime per Iteration 0.061567s

solution

```
location_university_of_w
aterloo_1
store_food_basics_2      water
store_giant_tiger_1      shredded_cheese
store_zehrs_2            salad
store_zehrs_2            boneless_pork_chop
store_zehrs_2            coffee
store_food_basics_2      salsa
store_sobeys_1           fish_fillet
store_giant_tiger_1      juice
store_valu_mart_1        cheese_bar
store_food_basics_2      entree
store_food_basics_2      grape
store_giant_tiger_1      post_cereal
store_giant_tiger_1      astro_yogurt
store_valu_mart_1        pepsi
store_zehrs_2            pc_chicken_breast
location_university_of_w
aterloo_1
```

The best cost solution is shown in blue while the minimum solution for each iteration is shown in red

Figure 7: Example GUI of the PSO algorithm showing parameters and results

Chapter 7: Conclusions and Recommendations

The report examines the application of six algorithms: TS, SA, GA, PSO, ACS, ABC on the shopping list problem. All algorithms were implemented in MATLAB analyzed based on their performance and average time taken. Although all algorithms were able to solve the problem, although there was a relatively large difference in performance in each of them in terms of time taken and objective function cost.

From our results, we found that ACS and PSO returned the best results. However, both algorithms took a relatively long time to converge. The best algorithm we found for practical use is GA. While performance is important, waiting for several minutes for a solution is a very poor experience for a customer waiting for their shopping route. GA takes a reasonable amount of time (around 25s) per run and provides a solution that is close to the other two top performing algorithm solutions. ABC and SA were in the middle of the pack in terms of time taken, but also gave results that were relatively poor compared to the other solutions. TS overall performed quite poorly as the algorithm often got stuck on local minima based on the initial guess. It is not very well suited to our problem as it is highly multimodal.

The algorithm implementations could do with further tuning and code optimization, which could reduce the running time. They could also be refined to be cooperative to utilize all the computational resources available to help improve the results of each algorithm by decreasing the computation time or by being allowed to search more through the search space without increasing the additional wait. This could make the better performing algorithms (PSO and ACS) better candidates as particles or ants could be processed in parallel. Some algorithms like Bee Colony however did follow a hard cap of the number of iterations and thus, maybe could've found better solutions if given more time to explore.

Another factor that could've changed the results was the objective function. A change in weight on price or distance could've possibly had an effect on the results found for each algorithm. Also, if we had attempted to solve for a Pareto-Front, giving a variety solutions rather than just a weighted objective function, it could've drastically changed the efficiency of each algorithm.

References

[1] M. Dorigo and L. Gambardella, “Solving Symmetric and Asymmetric TSPs by Ant Colonies”, in IEEE Conference on Evolutionary Computation, 1996, pp 622-628.

[2] M. Gendreau, G. Laporte and F. Semet, 'A tabu search heuristic for the undirected selective travelling salesman problem', *European Journal of Operational Research*, 1998, vol. 106, no. 2-3, pp. 539-545.

[3] E. Goldbarg, M. Goldbarg and G. de Souza, “Particle Swarm Optimization Algorithm for the Traveling Salesman Problem”, Travelling Salesman Problem, September 2008, ISBN 978-953-7619-10-7, pp. 202.

[4] B. Golden, J. Pepper, and E. Wasil, “Solving the Traveling Salesman Problem With Annealing-Based Heuristics: A Computational Study”, IEEE Transactions On Systems, Man, And Cybernetics - Part A: Systems And Humans, January 2002, vol. 32, No. 1, pp. 72-77.

[5] B. Gorkemli and D. Karaboga, “A Combinatorial Artificial Bee Colony Algorithm for Traveling Salesman Problem”, in Innovations Systems and Applications, 2011, pp. 50-53.

[6] J. Grefenstette et al., “Genetic Algorithms for the Traveling Salesman Problem ,” in Proc. of The First International Conference on Genetic Algorithms and their Applications, Pittsburgh, PA, 1985, pp. 160-177

[7] B. Li, B.Liu, and Ling Wang, “An Effective PSO-Based Hybrid Algorithm for Multiobjective Permutation Flow Shop Scheduling”, IEEE Transactions On Systems, Man, And Cybernetics - Part A: Systems And Humans, July 2008, vol. 38, No. 4, pp. 818-831.

[8] B. McKechnie (2014, July 11), “5 apps that will help you save money on groceries and more”. Available: http://globalnews.ca/news/1386892/5-apps-that-will-help-you-save-money-on-groceries-and-more

Appendices

All store locations are only visited once as all items will be purchased there. Duplicate store locations will be visited the first time it appears in the route. The charts below are the items that are bought from which store determined by the algorithm solutions.

Appendix 1: SA solution

Table 19: SA Solution for Real Data

Store	Item
location_university_of_waterloo	
store_zehrs_3	astro_yogurt
store_food_basics_2	salsa
store_super_store_1	entree
store_zehrs_3	coffee
store_sobeys_1'	pepsi
store_wholesale_...	water
store_zehrs_3	boneless_pork_chop
store_valu_mart_1	cheese_bar
store_zehrs_2	shredded_cheese
store_zehrs_3'	pc_chicken_breast
store_valu_mart_1	grape
store_sobeys_1	fish_fillet
store_zehrs_3	juice
store_valu_mart_1	post_cereal
store_zehrs_3'	salad
location_university_of_waterloo	

Appendix 2: GA Solution

Table 20: GA Solution for Real Data

Store	Item
location_university_of_waterloo	
store_zehrs_3	astro_yogurt
store_food_basics_2	water
store_sobeys_4	salad
store_valu_mart_1'	grape
store_food_basics_2	entree
store_zehrs_3	boneless_pork_chop
store_sobeys_4	fish_fillet
store_zehrs_3	coffee
store_sobeys_4	pepsi
store_zehrs_3'	shredded_cheese
store_valu_mart_1	cheese_bar
store_zehrs_3	post_cereal
store_zehrs_3	juice
store_zehrs_3	pc_chicken_breast
store_food_basics_2	salsa
location_university_of_waterloo	

Appendix 3: TS Solution

Table 21: TS Solution for Real Data

Store	Item
location_university_of_waterloo	
store_freshco_1	cheese_bar
store_food_basics_2	grape
store_sobeys_2	coffee
store_sobeys_3	pepsi
store_food_basics_2	entree
store_zehrs_1	salad
store_zehrs_1	pc_chicken_breast
store_sobeys_3	salsa
store_food_basics_2	water
store_sobeys_1	juice
store_sobeys_4	fish_fillet
store_zehrs_2	astro_yogurt
store_zehrs_3	post_cereal
store_zehrs_2	shredded_cheese
store_zehrs_2'	boneless_pork_chop
location_university_of_waterloo	

Appendix 4: ACS Solution

Table 22: ACS Solution for Real Data

Store	Item
location_university_of_waterloo	
store_valu_mart_1	pepsi
store_food_basics_2	entree
store_valu_mart_1	grape
store_zehrs_3	juice
store_valu_mart_1	cheese_bar
store_zehrs_3	post_cereal
store_zehrs_3	coffee
store_zehrs_3	pc_chicken_breast
store_food_basics_2	water
store_zehrs_3	astro_yogurt
store_zehrs_3	salad
store_sobeys_2	fish_fillet
store_zehrs_3	shredded_cheese
store_zehrs_3	boneless_pork_chop
store_zehrs_3	salsa
location_university_of_waterloo	

Appendix 5: PSO Solution

Table 23: PSO Solution for Real Data

Store	Item
location_university_of_waterloo	
store_zehrs_3	boneless_pork_chop
store_food_basics_2	grape
store_food_basics_2	entree
store_giant_tiger_1	pepsi
store_zehrs_3	post_cereal
store_giant_tiger_1	astro_yogurt
store_zehrs_3	pc_chicken_breast
store_giant_tiger_1	juice
store_food_basics_2	water
store_zehrs_3	salsa
store_sobeys_1	coffee
store_valu_mart_1	cheese_bar
store_sobeys_1	fish_fillet
store_zehrs_3	salad
store_zehrs_3	shredded_cheese
location_university_of_waterloo	

Appendix 6: ABC Solution

Table 24: ABC Solution for Real Data

Store	Item
location_university_of_waterloo	
store_food_basics_2	entree
store_food_basics_2	grape
store_giant_tiger_1	post_cereal
store_zehrs_2	boneless_pork_chop
store_sobeys_1	salsa
store_sobeys_1	coffee
store_valu_mart_1	cheese_bar
store_sobeys_4	fish_fillet
store_food_basics_2	water
store_zehrs_3	shredded_cheese
store_zehrs_3	astro_yogurt
store_zehrs_2	pepsi
store_zehrs_1	pc_chicken_breast
store_zehrs_1	salad
store_zehrs_1	juice
location_university_of_waterloo	