UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Department of Computer Science

Computer Science Honours Final Paper 2016

Title:	Optimization of Corporate Fund Matching using Metaheuristics
Author:	Jacques Heunis
Project abbreviation:	FUNDMATCH
Supervisor:	Michelle Kuttel

Category	**Min**	**Max**	**Chosen**
Requirement Analysis and Design	0	20	0
Theoretical Analysis	0	25	0
Experiment Design and Execution	0	20	20
System Development and Implementation	0	15	10
Results, Findings and Conclusion	10	20	20
Aim Formulation and Background Work	10	15	10
Quality of Paper Writing and Presentation	10		10
Quality of Deliverables	10		10
Overall General Project Evaluation	0	10	0
Total marks			80

Optimization of Corporate Fund Matching using Metaheuristics

Jacques Heunis
HNSJAC003
University of Cape Town

CCS Concepts

•Theory of computation → Bio-inspired optimization; •Computing methodologies → Genetic algorithms;

ABSTRACT

Holding companies needs to manage the need and usage of money available to their subsidiaries. This is done by making loans between subsidiaries in order to satisfy those that require additional funds. The problem of fund matching optimization is to determine the set of loans which will satisfy all funding requirements at the lowest total cost. We investigate the use of Genetic Algorithms and Particle Swarm Optimization in addressing this problem. We evaluate both algorithms on a real-world instance of the problem as well as a variety of randomly generated instances. Particle Swarm Optimization performs poorly on all problem instances that we tested, but we find that the Genetic Algorithm performs better on average than existing manual and heuristic approaches.

Keywords

Optimization, Genetic Algorithm, Particle Swarm Optimization

1. INTRODUCTION

Large holding companies own many subsidiaries, with widely varying financial needs. Subsidiaries can be categorized by their profitability, either generating excess income (a funding source) or being in need of additional income (a funding requirement). Companies in need of additional funds would ordinarily need to acquire those funds with a loan from an external source, such as a bank. Loans from a bank carry a high interest rate. However, the holding company can instead allocate loans between its subsidiaries, satisfying funding requirements using funds from a source subsidiary at significantly lower interest rates. These allocations are currently determined manually by employees of the holding company. These employees follow a greedy heuristic to decide which sources loan to which requirements and to select a start date, duration and amount for each loan. Given the complexity of the problem and the large number of possible solutions, allocations created manually or greedily are likely to involve spending more money than is necessary paying interest on the specified loans.

An efficient allocation of funds would directly save money in holding companies by satisfying all funding requirements while paying less interest in total on the loans required to do so. An automatic process to output an efficient set of allocations would also reduce the time that employees spend evaluating or modifying allocations.

We investigate algorithmic allocation of funds, with the aim of finding a method that is significantly more efficient than the existing manual process. The input is a set of funding sources and requirements, each with a start date, duration, funding amount, tax class and (in the case of sources) an interest rate. The output is an allocation of sources to requirements such that the total cost of satisfying all requirements is minimized. We can formalize this as a constrained optimization problem in the following manner:
We are given a set of requirements R, a set of sources S, and a set of allocations A. For a source s we denote its start date as b_s, its duration as d_s, its interest rate as i_s, its tax class as τ_s and its available balance as v_s. For an allocation a, we denote the source being allocated from as s_a, the requirement being allocated to as r_a, the duration of the allocation as d_a, the start date of the allocation as b_a, and the amount being allocated as v_a. We define the set of allocations from a source s at time t as:

$$A_{s,t} = \{a \in A : s_a = s \text{ and } b_a \leq t < b_a + d_a\}$$

as well as the set of allocations to a requirement r at time t as:

$$A'_{r,t} = \{a \in A : r_a = r \text{ and } b_a \leq t < b_a + d_a\}$$

Finally, we define a constant i_E, representing the interest rate of the external source of funding used when no source can satisfy a requirement. The problem is then to minimize the cost of satisfying all requirements, given by

$$Cost = \left(\sum_{a \in A} i_{s_a} d_a v_a\right) + i_E \sum_{r \in R} \left(\int_{b_r}^{b_r+d_r} \left(v_r - \sum_{a \in A'_{r,x}} v_a\right) dx\right)$$

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

subject to the following constraints:

$$\tau_{s_a} = \tau_{r_a} \qquad \forall a \in A \quad (1)$$

$$b_{s_a} \leq b_a < b_{s_a} + d_{s_a} \qquad \forall a \in A \quad (2)$$

$$b_{s_a} \leq b_a + d_a < b_{s_a} + d_{s_a} \qquad \forall a \in A \quad (3)$$

$$\sum_{a \in A_{s,t}} v_a \leq v_s \qquad \forall t \in [b_s, b_s + d_s], \forall s \in S \quad (4)$$

Equation 1 ensures that a source can only be allocated to a requirement with the same tax class. Equations 2 and 3 ensure that an allocation overlaps in time with its source. Equation 4 ensures that, at all points in time, the total value allocated from a given source does not exceed the capacity of that source.

This is a constrained function minimization problem where the complexity of the cost function makes it infeasible to construct an optimal solution analytically. Instead we search for an approximate solution using metaheuristics. Metaheuristics are stochastic search algorithms in which we can calculate the "fitness" of a solution - a measure of how good it is - and the search is guided solely by the fitness of known solutions. In particular we investigate the use of Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO). These two algorithms have both been successfully applied to related problems[9, 14, 16, 17] and take very different approaches.

We specifically seek to address the following questions:

1. Can we find a meta-heuristic search algorithm solution that performs better than the existing manual method for the funding allocation problem?
2. Can we create a Genetic Algorithm that performs better than the existing manual method for the funding allocation problem?
3. Can we create a Particle Swarm Optimization that performs better than the existing manual method for the funding allocation problem?
4. What is the relative performance of these two methods on the funding allocation problem?

To do this, we implement Particle Swarm Optimization and a Genetic Algorithm and compare their performance on a number of randomly generated datasets, as well as one real-world dataset provided by Old Mutual.

2. BACKGROUND

Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO) are different metaheuristic approaches to optimization.

2.1 Fitness Function

The main requirement for applying metaheuristics to a problem is that we can objectively evaluate a candidate solution to determine how good it is relative to other solutions. To do this, we construct a function (called a "fitness function") which takes a candidate solution as input and outputs a real-valued number. The output value of the fitness function is unimportant. The only requirement is that given the output of the function for two solutions, we can determine which solution is better.

"Better" may correspond either to lower values (for example the monetary cost of a solution) or higher values(for example the efficacy of a solution).

2.2 Particle Swarm Optimization

First developed by Kennedy and Eberhart in an attempt to model social behaviour, Particle Swarm Optimization is an approach to function optimization inspired by the motion of groups of birds or insects[12]. The algorithm consists of a number of candidate solutions (called particles), each with a position, velocity and a set of "neighbour" particles. On every iteration, each particle has a weighted average of the best position it has seen (the "local best") and the best position seen by any of its neighbours (the "global best") added to its velocity. The velocity of every particle is then added to its position and the process is repeated until the algorithm converges, or a set number of iterations has been completed.

In its original form, PSO has the following parameters: the number of particles, the number of neighbours, the strategy for selecting neighbours, the maximum velocity of each particle, and the weights of the local and global best positions. In 1998, Shi and Eberhart presented a modification of PSO which performed significantly better but introduced an additional "inertia weight" parameter which scales the velocity of a particle before it is accelerated[18]. Clerc and Kennedy presented an algebraic analysis of the convergence of PSO[4]. They proposed a variation in which the local and global best weights are equivalent, so they are both parameterized by a single number. In addition, their variant gives a means of calculating inertia weight, thereby reducing the number of parameters by two. Finally, in Poli, Kennedy and Blackwell's overview of recent advances to PSO techniques, they give commonly used values for the inertia weight and local best weight[15]. This leaves the number of particles and the number of neighbours as the only remaining numeric parameters that need to be set.

2.3 Genetic Algorithm

Genetic algorithms are an optimization technique that was first described by Holland and models the processes of natural selection and sexual reproduction[11]. Genetic algorithms consist of a population of individuals (each representing a candidate solution) whose fitness is evaluated at each iteration and is used to select a set of individuals that act as "parents" for the next iteration. These "parent" individuals are then combined and randomly modified to produce a "child" population, to be used in the next iteration.

The general description of genetic algorithms leaves many details unspecified and a wide variety of modifications has been introduced and explored in the literature. When originally described by Holland, individuals were represented by fixed-length bit-strings. This representation makes crossover and mutation easy to define, but is not suitable to every problem. Other common representations still have a fixed length, but use arrays of integers or real values instead of bits[16][3][19].

Implementations can be broadly categorized as either "generational" (where an entirely new population is created from the parents at each iteration) or "steady-state" (where only a very small number of individuals are replaced at each iteration). The processes of parent combination and modification are left unspecified, requiring each implementation to define a "crossover operator" (which specifies how two individuals are recombined into two new ones) and a "mutation operator" (which specifies how single individuals are randomly modified). Which operators are selected depends on the specific problem being solved. For example in some representations, the order of the values is important and so operators that take this into account must be used. How parents are selected from the population is also left up to the implementation. Which parent selection scheme is used affects the behaviour of the optimization, potentially causing it to converge prematurely to a local optimum or to converge very slowly. Some examples include elitist selection (in which we select k parents by taking the best k members of the population), roulette-wheel selection (in which any individual can be selected with probability proportional to its fitness) and tournament selection (in which a few individuals are randomly selected and the best of those is used as a parent).

2.4 Applications

While literature on optimizing corporate funding allocations is scarce, both of the algorithms that we investigate have been successfully applied to a variety of problems in the past.

Roth and Levine applied a genetic algorithm to the problem of extracting geometric primitives from image or sensor data[17], representing a primitive as a minimal set of points which uniquely defines the primitive. This approach was effective but was not shown to be an improvement over existing deterministic methods. Mendes et al. presented a genetic algorithm for the constrained multi-project scheduling problem, trying three variants of their approach which make use of increasing amounts of problem-specific information to improve performance[9]. The variant which used the most problem-specific information performed the best, often getting very close to the optimal solution even on large problem instances. Oh et al. proposed the use of genetic algorithms to support portfolio selection for index fund investments[13]. Experimental results show that the GA performs significantly better than conventional mechanisms in all scenarios except when the financial markets do not fluctuate very much.

We give a few examples of the use of Particle Swarm Optimization in economics or finance, although an analysis of the applications of PSO by Poli showed that there is relatively little literature on the use of PSO in this area[14]. Some of the more common applications include designing resource distribution networks (such as for electricity), image analysis and data clustering or classification. Chang and Shi proposed using Particle Swarm Optimization to construct investment stock portfolios[2]. Their approach is a hybrid one which first uses conventional methods for selecting stocks which are likely to perform well and then uses PSO to find a good weighting of these stocks within the investment fund. Their experiments show that their method produces more profitable investment portfolios than conventional methods. Dye and Ouyang proposed the use of PSO to determine an optimal selling price, and stock replenishment schedule for retail products[7]. Their experimental results show that the PSO performs well enough to be used in practice, but they also give a number of extensions which could improve performance.

3. METHODS

The two main components of this project are the Particle Swarm Optimization and Genetic Algorithm implementations. Both of these algorithms require an encoding of candidate solutions into a form that the algorithm can work with, as well as a means of evaluating those solutions. This section covers the details of how solutions are evaluated, the encoding used, and the details of our PSO and GA implementations. A greedy heuristic approach has also been implemented as a baseline against which to compare.

3.1 Evaluating candidate solutions

The goal of the fund matching process is to reduce costs by making efficient allocations. We therefore evaluate solutions based directly on the total amount of interest charged for a set of allocations. Note that since lower values are better, we will use the more self-explanatory "cost function" instead of "fitness function".

We calculate cost using Algorithm 1, with the following notation:
Denote the interest rate of the external credit facility as i_E. If s is a source, then i_s denotes the interest rate for s. For x a requirement or allocation, t_x denotes its start date, d_x denotes its duration, and v_x denotes its value.

The cost of an allocation can be broken into two categories: the interest owed to a source by making an allocation and the interest that is implicitly owed to the external source of funding because a requirement is not fully satisfied for some period of time. Algorithm 1 achieves this by first computing the cost of satisfying every requirement from the external funding source. For every allocation, the algorithm then subtracts the amount that is saved by allocating from a source instead of from the higher-interest external source.

Algorithm 1 Fitness Function

```
 1: result ← 0
 2: for all requirements $r$ do
 3:     result ← result + $v_r d_r i_E$
 4: end for
 5: for all allocations $a_i$ do
 6:     $t_{start} = \min(t_a,\ t_{r_a})$
 7:     $t_{end} = \max(t_a + d_a,\ t_{r_a} + d_{r_a})$
 8:     result ← result − $v_a(t_{end} - t_{start})(i_E - i_{s_a})$
 9:     if $t_a < t_{r_a}$ then
10:         $t_{extra} \leftarrow t_{r_a} - t_a$
11:         result ← result + $v_a t_{extra}(i_E - i_{s_a})$
12:     end if
13:     if $t_a + d_a > t_{r_a} + d_{r_a}$ then
14:         $t_{extra} \leftarrow (t_a + d_a) - (t_{r_a} + d_{r_a})$
15:         result ← result + $v_a t_{extra}(i_E - i_{s_a})$
16:     end if
17: end for
18: return result
```

When comparing the cost of output solutions, we normalize the costs by dividing by the "worst-case" cost of satisfying all the requirements in a given data set. This is the cost of satisfying all requirements entirely from the external funding source, which has a higher interest rate than any of the input sources. Normalizing in this fashion allows for comparison of results across datasets.

3.2 Solution Encoding

In order to apply a GA or PSO to our problem we must first construct a means of encoding candidate solutions so that the optimization algorithms can work with them. We consider the funding allocation problem as a graph where each source and requirement is represented by a vertex. There is an edge from vertex s to vertex r if a loan can be made from s to r. Each edge then corresponds to a potential allocation and has its own starting date, duration and amount.

We encode a candidate solution as a string of numbers $s_1, d_1, v_1, s_2, d_2, v_2...$ where the i^{th} allocation has start date s_i, duration d_i and amount v_i. Allocations that have $d_i \leq 0$ or $v_i \leq 0$ are ignored.

The only difference between the encoding used for the PSO and the GA is that the numbers in the PSO can be any real value while those in the GA are kept to integer values. The GA uses integers because unlike the PSO, the GA can work effectively with integers and all numbers in the input and output data are integers. Using integers also reduces the size of the search space, making good solutions easier to find.

We encode dates to numbers by taking the number of months since the 1^{st} of January on the year 1. This encoding does not include the day of the month, so we assume that all dates correspond to the 1^{st} of that month. This greatly simplifies date calculations but does not lose too much detail since allocation duration is always an integer number of months as well.

3.3 Particle Swarm Optimization

Our PSO implementation is based on Clerc and Kennedy's version with a constriction coefficient[4]. This implementation has fewer parameters to set than earlier versions.

Pseudo-code for our implementation is given in Algorithm 2. We denote a particle p with position x_p, velocity v_p, and past location with the lowest cost x_p^*. We denote the constriction coefficient by χ, the weight of the best previous location by ϕ_1 and the weight of the best previous location of any neighbour by ϕ_2. The function $\vec{U}(x)$ returns a vector of real values from a uniform distribution in the range $[0, x]$. $\otimes$ is a coordinate-wise product.

Algorithm 2 Particle Swarm Optimization

```
 1: Initialize particles with random positions and velocities
 2: for $n$ iterations do
 3:     for all particles $p$ do
 4:         Find $p_n$, the neighbour with the best past success
 5:         $\vec{v_p} \leftarrow \chi(\vec{v_p}+\vec{U}(\phi_1)\otimes(\vec{x_p^*}-\vec{x_p})+\vec{U}(\phi_2)\otimes(\vec{x_{p_n}^*}-\vec{x_p}))$
 6:     end for
 7:     for all particles $p$ do
 8:         $\vec{x_p} \leftarrow \vec{x_p} + \vec{v_p}$
 9:         if $p$ represents a feasible solution then
10:             Compute $p$'s new cost
11:             if $p$'s new cost is the lowest seen so far then
12:                 $\vec{x_p^*} \leftarrow \vec{x_p}$
13:             end if
14:         end if
15:     end for
16: end for
```

We initialize particle velocities to random value in the range $[-0.1, 0.1]$. Swarms initialized in this way have been shown to converge faster than the conventional approach of assigning random values within the domain of each variable[8]. Particle positions are initialized by assigning a random value within the feasible range of each dimension. The exception to this is the (monetary) amount of each allocation. The amounts of allocations from the same source are inherently linked because if their sum exceeds the amount of the source then the solution violates one of the problem constraints. Allocation amounts are initialized randomly in the range $[0, v_s]$, where v_s denotes the total amount available from the source for that particular allocation. This initialization is done independently for each allocation.

This initialization procedure naturally results in a large number of infeasible solutions. To combat this, our initialization procedure attempts to re-initialize each particle until a feasible position is found, with a maximum of 5 re-initializations per particle. This alone is not sufficient to ensure that a feasible solution is found. After initialization, one of the particles is re-initialized with all of its allocation amounts set to 0. This ensures that at least one feasible solution is known after initialization.

Traditional PSO does not have a means of handling constraints, so we introduce one for our problem. We use an approach inspired by a constraint-handling technique proposed by Deb for Genetic Algorithms[5]. When determining which of two solutions is better, there are three possible cases to consider regarding the feasibility of the two solutions. If both solutions are feasible then the one with lower cost is selected. If one solution is feasible and the other is infeasible, then the feasible solution is selected. If both solutions are infeasible then we compute a measure of how badly each of them violate the problem constraints, selecting the solution that is closer to being feasible. This comparison is applied both when determining whether a particle's new position is better than its best previously seen position and when determining which of a particle's neighbours have the best previously seen position.

3.4 Genetic Algorithm

We implement the traditional Simple Genetic Algorithm (Algorithm 3), with the main difference being that our individuals are encoded as a string of integer values, rather than a string of bits. Both generational and steady-state GAs were implemented initially. The generational GA showed more promise than a steady-state one in early tests, so we examine only the generational GA in this report.

Algorithm 3 Simple Genetic Algorithm

```
1: Initialize the population with random values
2: for $n$ iterations do
3:     Select parent individuals for the next population
4:     Crossover pairs of parents
5:     Mutate all parents
6:     Evaluate the new population
7: end for
```

We implement Deb's constraint-handling approach[5], which specifies the use of tournament selection to select the parent population. Selection of individuals for the tournament is done with replacement, so an individual can be selected multiple times for the same tournament. This is to reduce the amount of computation needed when selecting individuals for the tournament.

Crossover and mutation operators were considered as tunable parameters and are detailed in Section 3.6.2

3.5 Greedy Heuristic

For comparison, we implement a heuristic that greedily assigns sources to requirements in a one-to-one fashion based on how closely their amounts and durations match. Pseudocode is given in Algorithm 4. When looking for the best matching source for a given requirement, we first look for sources that have large amounts and durations in an attempt to fully satisfy the requirement. Once a satisfying source has been found, we start looking instead for smaller sources which still satisfy the requirement. This is to prevent the largest sources being assigned to small requirements that appear early on in the list.

Algorithm 4 Greedy Heuristic Allocation

```
1: for all requirements $r$ do
2:     Find the unused source $s$ that best matches $r$'s amount and duration
3:     Allocate $s$ to $r$
4: end for
```

This heuristic is based on a description of the existing manual process, which involves an employee roughly following these steps to create a set of allocations.

3.6 Parameterization

Metaheuristic algorithms often have a number of parameters (usually real numbers) which need to be set according to the specific problem at hand. Parameterization refers to the identification of parameters that need to be set and the process of determining to what value they should be set.

3.6.1 *Particle Swarm Optimization*

Parameters that need to be set for PSO are the number of particles, the number of neighbours that each particle has, an inertia factor and the weights assigned to the best solution seen by a particle itself and by its neighbours.

Clerc and Kennedy proposed a replacement for inertia, called a "constriction coefficient"[4], which is multiplied by the current velocity as well as the acceleration vectors (whereas the inertia factor is only multiplied by the current velocity). If we denote the self-best weight by ϕ_1 and the neighbourhood-best weight by ϕ_2, then let $\phi = \phi_1 + \phi_2$ and for $\phi > 4$, the constriction coefficient can be calculated as

$$\chi = \frac{2}{\phi - 2 + \sqrt{\phi^2 - 4\phi}}$$

Modern PSO implementations typically use $\phi_1 = \phi_2$ and $\phi = 4.1$ giving $\chi \approx 0.73$[1], which is the value used in our implementation.

This just leaves us to set the size of the swarm (number of particles) and the number of neighbours that each particle has.

For the swarm size, we tested the values 10, 20, 50, 75, 100 and 200. Each time we did five runs on three data sets (with 20, 40 and 60 requirements respectively) and took the average normalized cost. The number of neighbours used was different each swarm size (Table 1). On the smaller swarm sizes this was in order to keep the neighbour count small relative to the size of the swarm. On larger swarm sizes the neighbour count was capped at 10 to prevent particles from considering too many neighbours, which could encourage them to converge prematurely.

Swarm Size	Neighbour Count	Average Normalized Cost
10	2	0.987
20	4	0.977
50	10	0.967
75	10	0.967
100	10	0.980
200	10	0.973

Table 1: Normalized solution cost for different swarm sizes. Averaged across runs and datasets.

Number of neighbours	Average Normalized Cost
1	1.000
2	1.000
3	0.983
4	0.983
5	0.977
6	0.983
7	0.970
8	0.973
9	0.976
10	0.973

Table 2: Normalized solution cost for different neighbour counts and a swarm size of 50. Averaged across runs and datasets.

The results showed that a swarm size of 50 produces slightly better results than the other values we tested (Table 1).

Using a swarm size of 50, we determined a number of neighbours by testing the values $1, 2, 3...10$. Again we did five runs on three data sets (with 20, 40 and 60 requirements) and took the average normalized cost. The results showed that a neighbourhood of 7 other particles produces slightly better results than the other values we tested (Table 2). We therefore adopted a neighbourhood size of 7.

3.6.2 *Genetic Algorithm*

Our GA has the following parameters to set: Population size, mutation rate P_m, mutation operator, crossover rate P_c and crossover operator. De Jong suggested a mutation rate of $P_m = 1/\text{Population Size}$ and a crossover rate of $P_c = 0.6$[6], so we used those values. A population size of 100 was used while testing other parameters. As with PSO, all tests involve five runs on three data sets (with 20, 40 and 60 requirements) for each parameter value.

We considered two different mutation operators. For the first operator (which we will call "Single value" mutation) we mutate each allocation with probability P_m, randomly selecting between starting date, duration and amount and assigning that variable a new random value within its domain. The other operator we considered (which we will call "Single requirement" mutation) follows the same process, except that we assign a new random value for all three variables each time instead of just one. The results showed that the first option (mutating only a single value each time) produces significantly better solutions (Table 3).

We considered four combination operators. The traditional one-point crossover randomly selects a single allocation and all allocations preceding that one are crossed over. Interval crossover randomly selects two allocations and

Operator	Average Normalized Cost
Single value	0.830
Single requirement	0.873

Table 3: Normalized solution cost for different mutation operators. Averaged across runs and datasets.

Operator	Average Normalized Cost
1-Point	0.827
Interval	0.830
N-point	0.823
Requirement	0.827

Table 4: Normalized solution cost for different combination operators. Averaged across runs and datasets.

crosses over all allocations between them. N-point crossover gives every allocation a 50% probability of being crossed over. "Requirement crossover" (as we have chosen to call it) randomly selects a requirement and crosses over all allocations to that requirement. The results showed that N-point crossover gives slightly lower-cost solutions on average than the other operators (Table 4).

For the population size, we tested the values 10, 20, 50, 100, 150 and 200. The results showed that a population size of 200 produces the best results (Table 5). Larger population sizes were not tested because of the increased computation time that they require.

3.7 Evaluation

We evaluated each algorithm on a series of five randomly generated datasets and one real-world dataset (Table 6). Random datasets were generated by fixing the number of sources and requirements, selecting a range for the amounts and selecting a period of time into which all sources and requirements should fall. Sources and requirements were then generated with random amounts and source interest rates. Start dates and durations were also randomly generated, but within a range that ensured that they always fit into the selected time period.

Given the stochastic nature of Genetic Algorithms and Particle Swarm Optimization, each algorithm was run twenty times with each dataset as input. We recorded the minimum, maximum and average cost of the final solution across all ten runs. While our approach does not optimize for the fewest number of allocations, fewer allocations is preferable because it is simpler to manage. We therefore also recorded the minimum, maximum and average number of allocations that are present in each solution.

The real-world dataset was anonymized and provided to us by Old Mutual, along with a solution that they calculated manually. For this dataset, we also compare our results to the manually determined solution.

Population Size	Average Normalized Cost
10	0.943
20	0.920
50	0.867
100	0.830
150	0.807
200	0.800

Table 5: Normalized solution cost for different population sizes. Averaged across runs and datasets.

Dataset	Sources	Requirements
RDS-1	22	20
RDS-2	42	40
RDS-3	62	60
RDS-4	82	80
RDS-5	102	100
RW-DS	87	108

Table 6: Sizes of the datasets that we generated for testing.

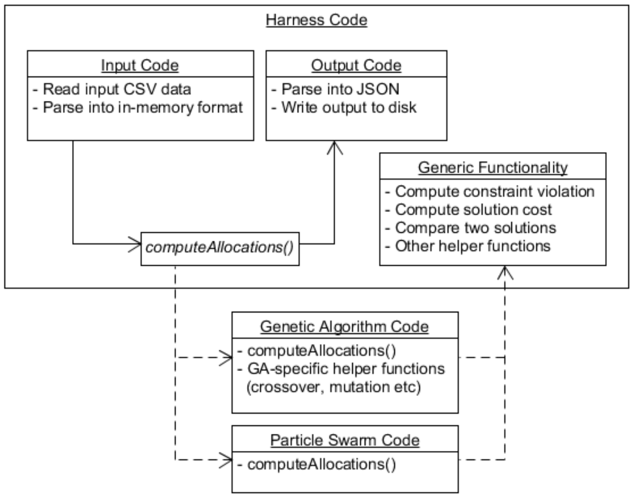


Figure 1: High-level layout of our implementation

4. IMPLEMENTATION

Implementation of Particle Swarm Optimization and a Genetic Algorithm is fairly simple - the complexity lies in creating an appropriate cost function and selecting values for parameters. We use C++ because it is fast in comparison to other popular languages such as Python or Java.

Our code was architected to make implementing new algorithms very easy, so the code has three main components: Genetic Algorithm code, Particle Swarm code and harness code (Figure 1). The scaffolding code handles data I/O, parsing and calling the appropriate optimization function. The input data is in comma-separated value (csv) format and the output is in JavaScript Object Notation (JSON) format. JSON was selected as the output format because it is a common data interchange format and is natively supported by JavaScript, so it can easily be used for visualization within a web browser. The parsing step allows the optimization to work with data in the most appropriate format regardless of the input or output formats.

In order to call the appropriate optimization routine, our harness code contains the forward-declaration of a `computeAllocations()` function that is not implemented. Each algorithm defines an implementation of `computeAllocations()` and the program is compiled with only one algorithm at a time, allowing the linker to make the connection.

We followed an agile approach to development[10]; at each point creating the simplest possible implementation of functionality and incrementally improving it. The harness code was written first to provide an environment in which any algorithm can run. The Particle Swarm implemention was next, followed by the Genetic Algorithm. PSO was implemented first because it is simpler, has fewer parameters and allowed us to get a better understanding of the problem structure before implementing the GA.

5. RESULTS AND DISCUSSION

While the goal of the optimization is to minimize the total cost of all allocations, we also include an analysis of the number of allocations present in each solution and the running time of each algorithm.

Note that Figures 2, 4 and 5 have error bars. These denote the full range of values that we observed. The columns represent the average of the values we observed for that dataset.

5.1 Total allocation cost

Particle Swarm Optimization did not manage to find good solutions. On every dataset that we tested, the best solution found by the PSO was significantly worse than the worst solution found by either the GA or the heuristic (Figure 2). On all except the smallest two datasets (RDS-1 and RDS-2), the best solution found by PSO was not better than the worst-case situation in which no allocations are made. Indeed the solutions that PSO found for RDS-4 and RDS-5 where exactly the solution with no allocations (Table 4). It is likely that this is a result of the interaction between the larger number of requirements and our initialization procedure. We initialize allocation amounts to random values between zero and the size of the source, independent of the other allocations from this source. The large number of requirements makes it highly likely that this initialization "over-allocates" from any given source, making it an infeasible solution. It is therefore highly likely that just after initialization, the only particle which represents a feasible solution is the one which has amounts all set to 0. Since the constraint-handling criteria prefer any feasible solution over any infeasible solution, all particles are accelerated towards this one feasible solution, which by definition has no allocations.

To check this justification, we counted the number of particles that represented feasible solutions immediately after initialization. We tested our six datasets five times each and every time, only one particle (out of fifty) was feasible.

The test runs show that while its progress is much slower than that of the GA, our PSO implementation is able to find better solutions after initialization (Figure 3a). On larger datasets however, the final solution is either the same or negligibly different from what was randomly selected during initialization (Figures 3b and 3c).

The GA performed better than the heuristic on three out of the six datasets that we tested, and performed equivalently well on one of them (Figure 2). Interesting to note is that while the size of RW-DS is between that of RDS-4 and RDS-5 (Table 6), the heuristic performed better on both RDS-4 and RDS-5, but worse on RW-DS. This might suggest that the heuristic is less effective on real-world instances of the problem than randomly-generated ones but with only one real-world dataset to test, we do not have enough evidence to draw a justifiable conclusion about this.

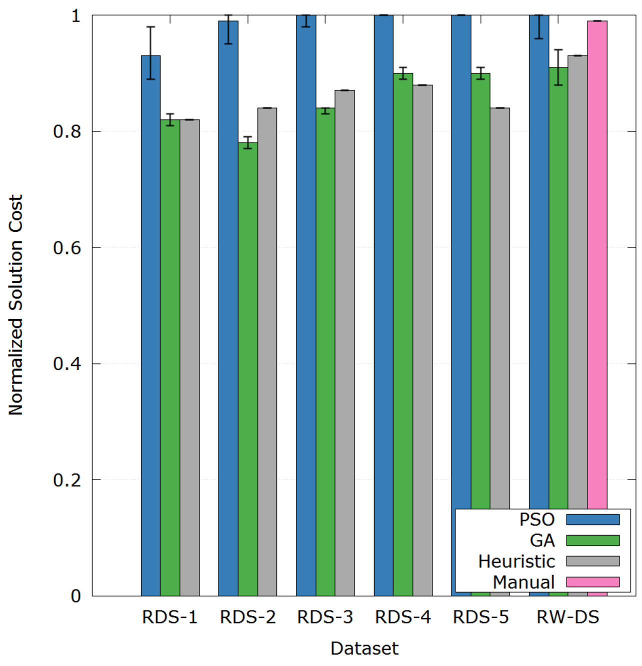


Figure 2: Normalized average solution cost for each algorithm across datasets

It is interesting to note that the cost of the manual solution is not significantly lower than that of the (essentially random) solutions found during initialization of the GA and PSO (Figure 3d).

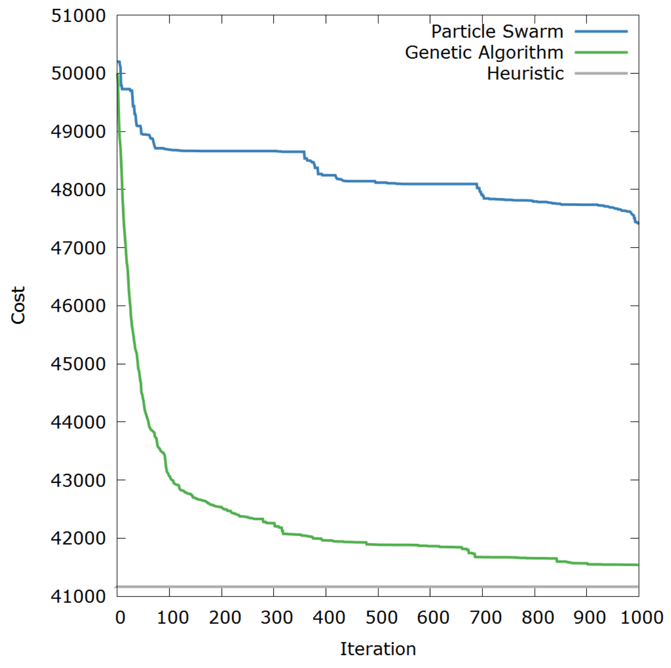
The difference between the cost of the heuristic solution and the manual one is of interest because the heuristic was developed from a description of the manual process. This difference could therefore indicate that the manual process either follows a different heuristic, does not follow this one very closely or that there are other factors which affect the desirability of a solution but which our cost function has not taken into account. As with any manual process, there is also the possibility of human error.

A closer inspection of the manual solution revealed exactly one allocation that runs over time - it runs for two months after the end date of the requirement to which it allocates. While it is possible that there is some external factor (such as wanting to align the end of the allocation with another event), the fact that this appeared only once suggests that human error is the more likely cause.

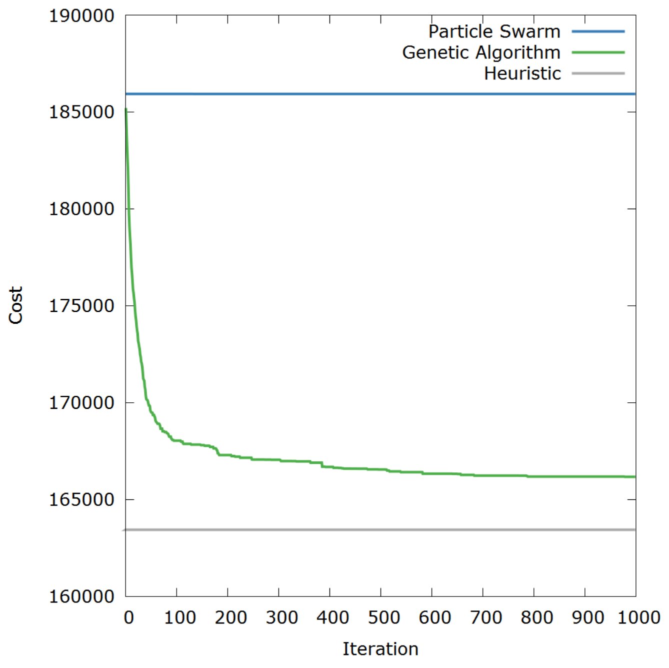
5.2 Number of allocations

The GA consistently produced solutions with significantly more allocations than the PSO (Figure 4). As explained in Section 5.1, this is likely because the PSO fails to find a good solution and therefore tends towards the worst-case zero-allocations solution.

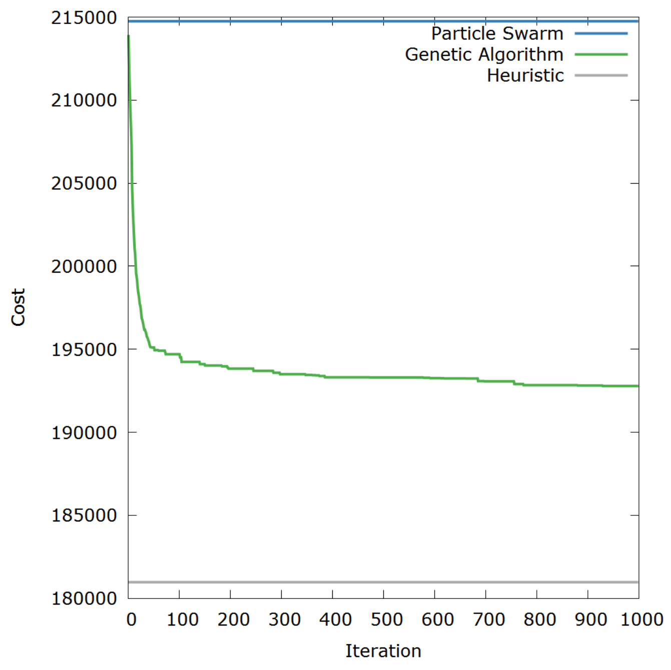
What is interesting about the PSO allocation counts in the smaller datasets RDS-1 and RDS-2, is that they cover a wide range of values. This wide range is also apparent in the normalized solution costs for these datasets (Figure 2) and indicates a significant level of inconsistency in the output of the optimization. This inconsistency is undesirable as it



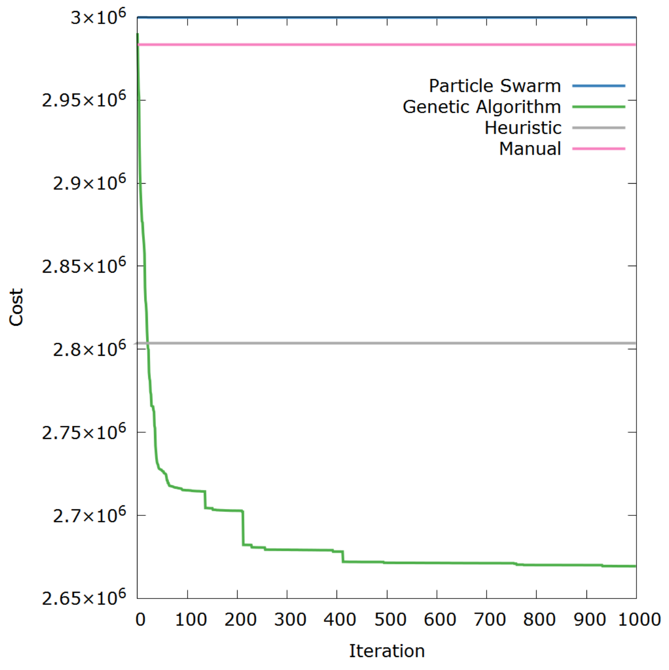
(a) RDS-1



(b) RDS-4



(c) RDS-5



(d) RW-DS

Figure 3: Normalized fitness within a single run on different datasets

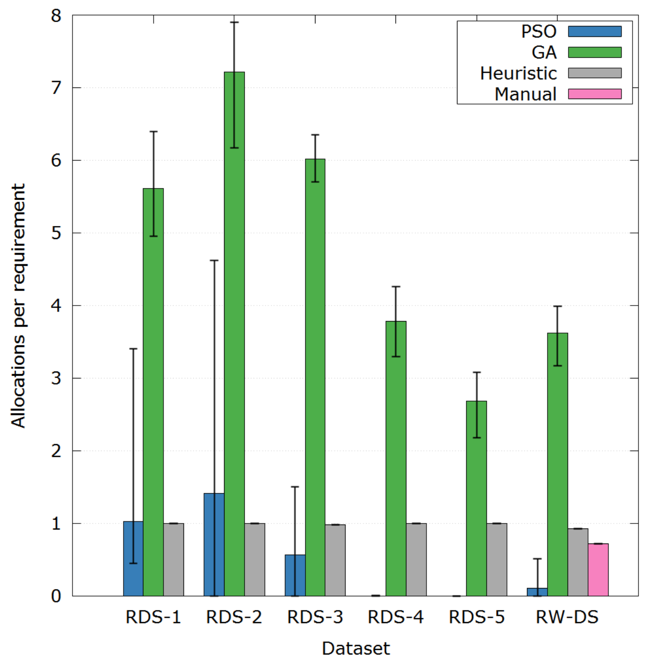


Figure 4: Number of allocations output by each method across all datasets

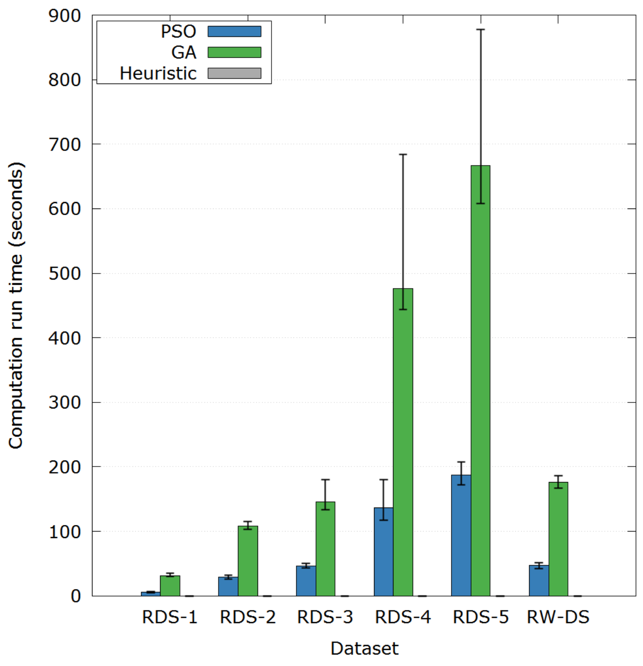


Figure 5: Runtime of each method across all datasets

Dataset	Allocations considered
RDS-1	247
RDS-2	1051
RDS-3	1610
RDS-4	3842
RDS-5	5048
RW-DS	1489

Table 7: Number of possible allocations considered by the optimization for each dataset.

detracts from the feasibility of applying PSO to the fund matching problem in the real world.

It is also worth noting that the GA produces solutions with a very high number of allocations. It could be that the optimal solution is one with many allocations, or it could be that the GA is simply not finding a good solution with fewer allocations. Without knowledge of a better solution, we cannot say which statement is true.

The number of allocations generated per requirement by the heuristic is not interesting because by design that number is one.

5.3 Run time

Despite running the same number of iterations each time, the GA takes much longer to run than the PSO (Figure 5). This can most likely be attributed to our GA having two hundred individuals in each population, while the PSO only has fifty particles in the swarm. However, it is also worth noting that the PSO is better suited to being optimized by the C++ compiler because it consists mostly of simple arithmetic operations and contains fewer branches than the GA.

Unlike the GA and PSO, the heuristic does not do any searching and instead it simply checks every source-requirement pair, selecting the best one. This is why the heuristic runs to completion in under a second, even on the largest dataset that we tested.

Another point of interest is the very high variability in the runtime of the GA, especially on RDS-4 and RDS-5.

While the real-world dataset (RW-DS) is larger than RDS-4, both algorithms run to completion in a significantly shorter amount of time on RW-DS than on RDS-4. One possible reason for this is that even though there are more sources and requirements, there is less overlap between them in terms of time. The optimization only considers allocations between sources and requirements which are active (IE their start date has passed but their end date has not) at the same time, so if the sources and requirements are very spread out, there would be fewer total allocations to consider than if they were clustered together. Analysis of each dataset reveals that this is indeed the case, RW-DS has far fewer allocations to consider than RDS-4 and RDS-5 (Table 7).

6. CONCLUSIONS AND FUTURE WORK

We applied Particle Swarm Optimization and a Genetic Algorithm to the problem of optimization of corporate funding allocation with the aim of improving on the existing manual method.

Of the three approaches we tested, PSO consistently had the worst performance. On a real-world dataset, it failed to find a solution with lower cost than the manually-computed one. On datasets whose size matches that of a typical real-world instance of the problem, PSO tended towards the worst-case solution of making no allocations and satisfying all requirements from the external source of funding.

The GA performed well overall, giving solutions which were better on average than that of the heuristic for half of the tested datasets. On the real-world dataset, the GA produced solutions which were significantly better than that of the heuristic and the manual solution. The cost of solutions given by the GA is fairly stable and varies within a relatively small range.

Solutions given by the GA tended to have a very high number of allocations when compared to those given by the heuristic or PSO. This could affect the practicality of using these solutions in practice.

The run time of the GA was far greater than that of PSO, but still well within the bounds of what could reasonably be used in practice.

Future work on this problem could investigate other approaches to initialization which avoid having to set all allocation amounts to zero, thereby improving the diversity of early populations. Implementation of automated or dynamic parameter control would be useful to eliminate reliance on experimentally determined parameter values. Reducing the number of allocations in solutions given by the GA would also make those solutions more practical for businesses to implement. This could be achieved by incorporating the number of allocations into the cost function or considering the number allocations as an objective and applying a multi-objective GA.

7. ACKNOWLEDGEMENTS

I would like to thank Graeme Smith and Ian Tunbridge from Old Mutual for their help in understanding the business aspects of the problem and for providing real-world data on which to test.

8. REFERENCES

[1] D. Bratton and J. Kennedy. Defining a standard for particle swarm optimization. In *IEEE Swarm Intelligence Symposium*, pages 120–127, 2007.

[2] J.-F. Chang and P. Shi. Using investment satisfaction capability index based particle swarm optimization to construct a stock portfolio. *Information Sciences*, 181(14):2989 – 2999, 2011.

[3] R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms-i. representation. *Computers & Industrial Engineering*, 30(4):983 – 997, 1996.

[4] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[5] K. Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2-4):311 – 338, 2000.

[6] K. A. DeJong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.

[7] C.-Y. Dye and L.-Y. Ouyang. A particle swarm optimization for solving joint pricing and lot-sizing problem with fluctuating demand and trade credit financing. *Computers & Industrial Engineering*, 60(1):127 – 137, 2011.

[8] A. Engelbrecht. Particle swarm optimization: Velocity initialization. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[9] J. Gonçalves, J. Mendes, and M. Resende. A genetic algorithm for the resource constrained multi-project scheduling problem. *European Journal of Operational Research*, 189(3):1171 – 1190, 2008.

[10] J. Highsmith and A. Cockburn. Agile software development: the business of innovation. *Computer*, 34(9):120–127, 2001.

[11] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[12] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings, IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[13] K. J. Oh, T. Y. Kim, and S. Min. Using genetic algorithm to support portfolio optimization for index fund management. *Expert Systems with Applications*, 28(2):371 – 379, 2005.

[14] R. Poli. Analysis of the publications on the applications of particle swarm optimisation. *J. Artif. Evol. App.*, 2008:3:1–3:10, 2008.

[15] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007.

[16] J.-Y. Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63(3):337–370, 1996.

[17] G. Roth and M. D. Levine. Geometric primitive extraction using a genetic algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(9):901–905, 1994.

[18] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proceedings, IEEE International Conference on Evolutionary Computation*, pages 69–73, 1998.

[19] M. Srinivas and L. M. Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, 1994.