Comparing the performance of metaheuristics applied to the Pickup and Delivery Problem

Daniel Mundell
Christian Dlamini
Kishan Jackpersad
Sipho Ntobela
Praveer Ramphul

School of Mathematics, Statistics, and Computer Science, University of Kwazulu-Natal, Private Bag Box X54001, Durban 4000, South Africa

220108104@stu.ukzn.ac.za
217036408@stu.ukzn.ac.za
217010318@stu.ukzn.ac.za
217067552@stu.ukzn.ac.za
217011174@stu.ukzn.ac.za

Abstract – The Pickup and Delivery Problem with Time Window constraints (PDPTW) is an NP-hard problem with no definitive solution. There exist many metaheuristic optimization algorithms that can be used to optimize it, such as the Tabu Search, the Cuckoo Search, and the Ant Colony Optimisation algorithm. In this paper, the first two aforementioned metaheuristic algorithms, as well as a hybridized version of the Max-Min Ant System, are applied to the PDPTW in order to compare their performance over a range of PDPTW task sizes, and determine the best algorithm in terms of lowest cost, best consistency, and lowest time to find valid solutions. This is done in an attempt to find the most efficient algorithms upon which future research may be done to improve time taken to find a near optimal solution to the PDPTW, possibly through a hybridization of two or more of the algorithms.

Keywords – Pickup and Delivery Problem with Time Windows, Max-Min Ant System, Cuckoo Search, Tabu Search, Metaheuristics, Optimisation, Hybridization

---

I. Introduction

The Pickup and Delivery problem (PDP) is a classic combinatorial optimization problem that revolves around the transportation and logistics industry. This problem is a variation of the Vehicle Routing Problem (VRP) and is an ongoing area of research that is aimed at minimizing the number of vehicles used, various expenses incurred and operational costs to perform all routes. These routing problems entail optimizing a service of either delivering goods or transporting people from one place to another. Optimization assists this industry in cutting costs for companies that are offering this service, as well as reducing the time taken for transportation.

Routes contain a sequence of pickup and delivery locations, where the pickup location is always reached before the delivery location, thus the introduction of a precedence rule. Each vehicle that is used on the route can only carry a certain number of goods or people and cannot be exceeded, thus the capacity rule is introduced. These two rules are the basis of PDPs. These problems are modelled as graphs that represent pickup and delivery locations as vertices, and the edge between them as the cost of traveling from one point to the next. Each route departs from the depot and ends at the final delivery vertex, while each vertex must be visited once and only once.

PDPs have many variations. The broadest one being static or dynamic PDP [1]. In a static PDP, all transport requests are known before the solving process begins. In a dynamic PDP, some of the transport requests are initially known, while others only become available in during the solving process. PDPs also vary by the number of vehicles used, that being Single-Vehicle PDP (SVPDP) or Multiple-Vehicle PDP (MVPDP). Lastly, not forgetting the PDP that takes time window constraints into consideration (PDPTW).

This study focuses on the PDPTW that takes the precedence constraint, the capacity of the vehicle, and the time windows of locations into consideration while applying various optimization algorithms to solve the problem. Namely, the algorithms explored are; the Tabu Search algorithm (TS), the Max-Min Ant System (MMAS), and the Cuckoo Search algorithm (CS). These algorithms were then compared against each other to assess their performance, efficiency, and consistency in three criteria: Best and average cost of a route, best and average time required to find the first feasible/valid solution, and the number of constraints broken.

II. Pickup and Delivery Problem (PDPTW) Description

Suppose we have m transportation requests between a set of locations $N = \{n_0, n_1, n_2, \dots, n_{2m}\}$, where n_0 is the initial depot or starting location and each $n_i, i = 1,2, \dots ,2m$ is a pickup or delivery location. Odd indices are pickup locations and even indices are delivery locations, where each pickup location n_i corresponds to the delivery location n_{i+1}. Due to the precedence constraint, the pickup location n_i must be visited before the corresponding delivery location n_{i+1} can be visited. Exactly one visit must be made to each location. Each transportation request has a load q_i to be collected from the pickup location and dropped off at the delivery location. Therefore, for each pickup and delivery pair n_i and n_{i+1}, we have $q_i > 0$, $q_{i+1} < 0$ and $q_i = -q_{i+1}$. Due to the capacity constraint, the vehicle has a maximum capacity of Q, which should not be exceeded. There is a symmetric travel time $t_{ij} = t_{ji}$ between all locations n_i and n_j. Each location has time window $[a_i, b_i]$ during which time it can be serviced. The vehicle can arrive at any location before the corresponding lower time window a_i, provided that it then waits until the location is ready to be serviced. However, a vehicle cannot service a location after the upper time window b_i has passed. There is no time associated with loading or unloading the vehicle.

To build a route, the vehicle begins at the depot and visits every location exactly once, while satisfying the precedence constraint, the capacity constraint, and the time window constraints.

While the PDPTW can have various objective functions, such as minimizing the total traveling distance, the total route duration, or the total waiting time [2], our objective function is to minimize the total route duration, which includes the total traveling time and the total waiting time.

A real-world example of the PDPTW would be the route taken by a food delivery service driver. The driver would always start at the depot and visit each location based on requests. The delivery driver needs to visit the restaurant before visiting the receiver of the goods (customer). If there are additional restaurants that need to be visited on the way, it would be a logical option to visit those locations as well before making the remaining deliveries. The customer would expect their food delivered within a certain timeframe, and thus food that is ordered for lunch should be delivered at lunch time. According to this example, the number of locations is two times that of the requests, i.e. for one request, the driver needs to go to the restaurant and then to the customer's delivery address. The time window would be within a certain time of ordering. It may be possible for an order to have been placed and if the driver is early to the restaurant, he would have to wait. If the driver is late, a penalty may be given, such as a smaller tip. The driver should also take into consideration the maximum capacity of their vehicle, and not carry more than this load at any given time.

The route taken by the delivery driver would be a series of locations that is determined by the pickup location and delivery location of each request. The driver should ensure that for each request, he picks up the food from the restaurant before delivering it to the respective customer, all within the selected time windows, and while not exceeding the vehicle's max capacity.

III. Background

A big part of any country entails dealing with transportation, i.e. moving of people or goods from one location to another. This brought about the need for researchers to develop algorithms or ways to minimize costs of transportation. Thus, the VRP was formally introduced in 1959 as The Truck Dispatching Problem by G.B Dantzig and J.H Ramser [3], where the problem was applied to a petrol delivery service. In the VRP, a set of customers, as well as their locations, and a fleet of vehicles that will accomplish this service are used as input data to the problem. In return, customers will be assigned to the available vehicles and a route that the vehicle will travel,

where the total distance travelled is minimized, will be generated. The VRP is then based on the Travelling Salesman Problem (TSP). In this problem, a list of cities and the cost of travelling between cities is used to find the most optimum route to minimize cost. So essentially once customers have been assigned to vehicles, the VRP turns into a TSP. A constraint of time windows was then applied to the VRP, introducing a new variant, the Vehicle Routing Problem with Time Windows (VRPTW).

Initially, the Ant System (AS) was proposed as an optimization method to solve the TSP. It could not solve the problem for more than 80 cities and could not compete with other heuristics or match their performance. As a result, research led to the Ant Colony Optimization (ACO) algorithm being developed. This worked much better than the AS when it was applied to the TSP [3].

The Tabu Search is a metaheuristic that uses local search methods for optimization in combinatorial problems. It was proposed in 1986 by Fred W. Glover where he describes the algorithm as one that tries to avoid entrapment in cycles by penalizing or outright forbidding movement which would take the solution to locations that have already been visited. Thus, the algorithm got its name from the Tongan word "tabu", which itself indicates things that cannot be touched because they are sacred.

The Cuckoo Optimization was developed in 2009 by Yang and Deb and the Cuckoo Optimization Algorithm was developed by Rjabioun in 2011 [4]. The inspiration for this algorithm came from the natural behaviour of the Cuckoo bird when it lays its eggs. Based on a nest where another bird has laid its eggs, the Cuckoo bird would lay its eggs there. Each egg in the nest represents a solution and the Cuckoo egg represents a new solution.

IV. Literature Survey

Jih et. al [5] had written a journal article on the PDPTW in which the Genetic Algorithm [6] (GA) was compared to a Family Competition Genetic Algorithm. It is based on a single vehicle which implies that the scope of the problem is limited to working with one vehicle at a time. To model this problem effectively, a starting location, which is referred to as the depot, is where each route begins. When each transportation request is assigned, the weight is known. The distance between locations are predefined. The vehicle must pick up loads from pickup locations and deliver them to delivery locations. The precedence constraint forces the vehicle to visit a pickup location before visiting the corresponding delivery location. Each location also has an assigned time window. Hence, Jih et. al [5] had chosen to explore 2 variations of the GA to optimise the PDPTW. The first variation is the traditional approach.

The traditional GA is based on Darwin's theory of natural selection, which simulates natural selection and includes operations such as crossover and mutation that occur during the process of natural selection. It starts off by 'guessing' the solution and then combining the fittest solutions to create the next generation of solutions. This process is repeated for each generation. This approach is not ideal due to the restraints of the PDPTW. Hence, Jih et. al [5] had researched the GA with Family Competition (FCGA).

The FCGA is based on the traditional GA but also includes the concept of family competition. Essentially, it simulates natural selection more accurately as only the best solution from each family can survive. This take on the Genetic Algorithm provides an improved method for the PDPTW, as the constraints may render the GA rather purposeless.

There are other metaheuristics which may be more well suited to the PDPTW, and the Ant Colony Optimisation (ACO) algorithm [7] is an attractive option. The ACO algorithm was developed in 2004 by Dorigo and is based on the foraging behaviour of ants in nature. By depositing pheromones along the path that they follow, ants have the ability to find the shortest path between two points. The path with the highest level of pheromones is considered to be the most favourable path and is the most attractive path for an ant to follow [7]. The longer the path is, the more time there will be for the pheromone to evaporate [8]. For this reason, the shortest path is favoured - there are inherently less pheromones that have evaporated along the path which results in more traffic and a higher concentration of pheromones.

The ACO typically consists of 3 factors, including the ant-based solution construction, pheromone update and iteration [7]. Each ant within the ACO algorithm represents a solution. Initially, when there are no pheromones, the ants will wander randomly which is why the initial solution is usually random [9].

In order to find the next path to be taken, each ant will consider the length of each path that is available as well as the level of pheromones from its current position. The choice of the next location is based on how attractive the path is. Once each ant has moved onto the next location, the pheromone value for that path is updated. The path gets a better 'rating' if more ants follow it.

The Max-Min Ant System (MMAS), which was proposed by Stützle, T. and Hoos, H. to solve the TSP in 1999 [10], can improve the general ACO's performance with minimal extra resource usage. The MMAX enforces max and min limits on the pheromone values, allowing the algorithm to begin by randomly searching a large area of the search space, and then starts to focus on the areas which are providing good solutions, and search them more thoroughly. Because of this, the algorithm is good at avoiding becoming stuck in local minimums, and is able to find good solutions consistently.

There has been significant work and research done on the Cuckoo search algorithm since it was developed in 2009 by Yang and Deb [11]. Shehab et al. [12] compiled a survey paper on the use of the cuckoo search in various areas. The algorithm has been modified to work in multiple domains such as in the medical field [13][14], economic dispatching [15][16], engineering[17][18], image processing[19] and energy[20][21].

The cuckoo search has also seen increased use in combinatorial problems including the travelling salesman and its variants such as the vehicle routing problem. The discrete cuckoo search by Yang has been used to solve the TSP [22]. The results showed that the cuckoo search was superior in comparison with other metaheuristics using the standard library for testing the TSP called TSPLIB. The discrete cuckoo search was also used by Ouyang et al. [23] to solve the spherical TSP. They also used the TSPLIB and achieved faster solutions using the HA30 entry from the TSPLIB. The discrete cuckoo search has also been used to solve the graph colouring problem which is a discrete problem [24]. The results show high performance of the algorithm when compared to other known heuristics techniques for searching namely the ant-based algorithm and bee colony.

Essgaer et al. [25] used a modified cuckoo search algorithm for solving the capacitated vehicle routing problem. They used twelve neighbourhood structures for a better improvement in the quality of the solution. The results obtained were comparable with those of other algorithms by finding near-optimal solutions. The findings also showed that these solutions were obtained at reasonable time and thus the cuckoo search could be used for the capacitated vehicle problem. The hybridization of the cuckoo search with other metaheuristics techniques is an ongoing research. The work of Kanagaraj et al. [26] showed that the cuckoo search and GA provided a very good balance between exploration and exploitation. The use of genetic operators which are mutation and crossover greatly reduced convergence which may lead to local optimum trap. This made the cuckoo search much more efficient at finding the optimal solutions.

Mati et. al [4] solved the single-vehicle PDPTW using the Tabu Search algorithm and a probabilistic Tabu Search algorithm. They found that their implementation produced optimal solutions in a short amount of time. Initially, the authors sorted the set of transportation requests in ascending order according to the lower bounds of the time windows. This was done as investigation into the PDPTW resulted in the time constraint being a major problem. The construction of the initial route also took the precedence and vehicle capacity constraint into consideration. Tabu Search and a probabilistic Tabu Search was then applied to the initial solution to improve it. To generate neighbours within a neighbourhood, the authors used a modified 2-exchange operator which considered the precedence rule and an insertion operator as well. The Tabu restriction did not allow a move to be executed again until a certain number of iterations had passed. This was stored in the Tabu List. The Tabu List construction was created to store tabu moves. The size of the Tabu List was not set to a fixed size, but was set to a random value at times during the search process. They found that if the size of the Tabu List was too big then the process became computationally expensive and if the Tabu List was set to a value that is too small, then the search got stuck in a cycle.

The Tabu Search algorithm and the probabilistic Tabu Search was tested on the Travelling Salesman Problem by Ascheuer, which had hard time windows (TSPTW) hence they did not take the capacity and precedence rules into consideration. Since optimal solutions were known, it was a good way to first test the algorithms. They then generated their own single-vehicle PDPTW instances to further test the performance of their algorithms. The algorithms were all run for ten executions. In terms of the TSPTW, they managed to obtain optimal solutions in the early stages. They found that the probabilistic Tabu Search had a lower frequency than the Tabu Search. They concluded that the cause for this was the probability of choosing a certain move would introduce infeasible solutions and thus the Tabu Search was more efficient than the probabilistic Tabu Search.

In terms of the instances that were generated by the authors for PDPTW, they stuck with a size of ten to forty customers. Both Tabu Search algorithms found a best-known solution in at least one execution. The Tabu Search managed to find feasible solutions more frequently and faster than the probabilistic Tabu Search. This study showed that the Tabu Search is an efficient algorithm for finding feasible solutions in a short amount of time, however the time window constraints were a major hurdle for the operators.

V. Choice of Techniques

We investigated various metaheuristic algorithms and identified three which we believed would work well for the PDPTW. Here we outline the reasons for implementing these three techniques:

Hybrid Max-Min Ant System with 2-OPT Local Search

- Constructive nature makes it efficient for problems similar to TSP.
- Population of ants can be compared to parallel processing.
- Convergence is guaranteed.
- Positive feedback causes fast improvement rates.

Cuckoo Search

- Population-based thus can search a huge number of solutions.
- Superior performance in continuous problems compared to the GA and Particle Swarm Optimization in multi-modal or multi-objective optimization problems.
- Has few parameters n and p_a thus much simpler to work with.
- Ease of hybridization in combination with other optimization techniques.

Tabu Search

- Enhances performance.
- Efficient.
- Applications are diverse.
- Finds quality solutions in short running times [27].

VI. Modelling and Program Design

Hybrid Max-Min Ant System with 2-OPT Local Search

The basic Ant Colony Optimization (ACO) is a stochastic nature-inspired metaheuristic algorithm which is used to solve hard combinatorial optimization problems by imitating the foraging behaviour of ants in the real world. Ants deposit trails of an odorous chemical substance, called pheromone, on the floor as a method of communication with other ants. After an ant has found a food source, it will return to its nest while depositing pheromone in quantities directly proportional to the quality and size of the food source. These pheromone trails guide other ants in their search for food, thus leading to an increased quantity of pheromone as more and more ants travel the same path [28].

An example of the foraging behaviour of ants, and their use of pheromone for communication is given in Figure 1. Consider a population of 20 ants along with Figure 1a, where A is the ant nest and there are two paths, through B and through C, to the only available food source D. The path through B has two sections of length 1 for a total length of 2, while the path through C has two sections of length 2 for a total length of 4. In Figure 1b, when the 20 ants begin their search for food, there are no existing pheromones, and so the ants split evenly between the two paths. Since the path through C is double the length of the path through B, the ants traveling through B will reach the food source and return to their nest by the time the ants traveling through C reach the food source. Therefore,

after one complete journey, there will be double the amount of pheromone deposited on the path through B. Figure 1c shows that when the 20 ants return to the food source for a second time, the higher pheromone level on the path through B will cause a larger proportion of the ants to take this path. In this way, the ants are collectively able to find the most direct path to the food source.

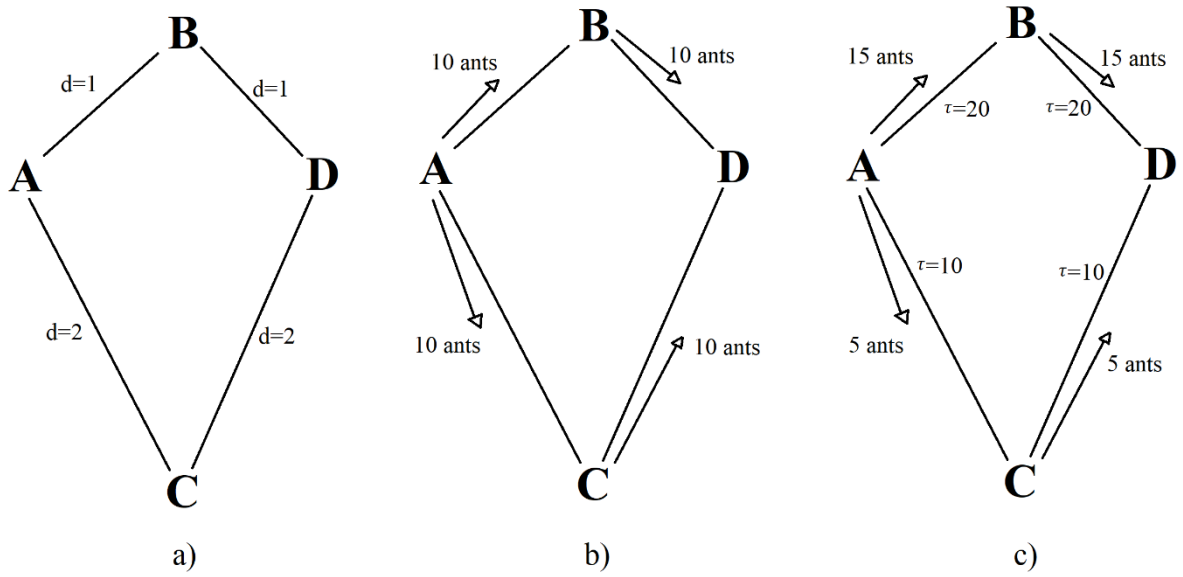

Figure 1: A simple ant pheromone example.

Since the ACO was initially not providing the performance we expected, we investigated ways in which we could improve its results. We discovered the MMAS [10], mentioned in the literature review, as well as the 2-OPT local search technique.

MMAS was able to drastically improve the performance of the ACO with three simple changes: Firstly, instead of all ants updating the pheromone trails based on their route's cost after each iteration, only the single best ant now does so. Secondly, there is an upper and lower bound applied to the pheromone value for any trail, i.e. a max and a min value. And lastly, there is a specific way in which the pheromone trail values are initialized. Since there was not a large increase in resource usage for the MMAS, we decided to implement it instead of the basic ACO. Applying the 2-OPT local search technique allowed us to hybridize the MMAS for even better performance.

Limitations:

- Time till convergence is uncertain.
- Difficult to analyse because of the inherent randomness.
- Probability distribution changes frequently.
- Limited benefit from 2-OPT because of the large number of constraints in PDPTW.

MMAS 2-OPT with respect to the PDPTW:

Algorithm 1 shows a pseudocode overview of applying the MMAS 2-OPT technique to the PDPTW. All the pheromone trails are initialized to a constant value $c > 0$ in the initializePheromones() function, as this is a requirement of the MMAS. However, because all the pheromone trails are initialized to the same value, the ants still search randomly before any pheromone updates take place. The rest of our implementation runs a set number of iterations, although it could be set to run until a difference criterion is met, such as reaching a specific cost or error allowance. When the generateSolution() function runs for each ant in the population, routes are constructively built using Algorithm 2. Each location is checked to see if it is feasible, i.e. it does not break the precedence or capacity constraints and has not already been visited. The time window constraints are dealt with separately in the heuristic information of the transition probabilities shown in Eqn. 1.

$$P_i^j = \frac{\left[\tau_i^j\right]^\alpha \times \left[\eta_i^j\right]^\beta}{\sum_{k \epsilon N}\left(\left[\tau_i^k\right]^\alpha \times \left[\eta_i^k\right]^\beta\right)} \quad (1)$$

Where τ_i^j and η_i^j are the pheromone and heuristic information from location *i* to location *j*.

If our implementation was being applied to the TSP, a simple heuristic could be used, such as the inverse of the distance between two locations, as shown in Eqn. 2. However, since the PDPTW has many more constraints, we created the heuristic shown in Eqn. 3, to help our implementation find feasible solutions.

$$\eta_i^j = \frac{1}{d_i^j} \quad (2)$$

$$\eta_i^j = \frac{1}{\left[d_i^j\right] + a_j\left[b_j\right]^2} \quad (3)$$

Where d_i^j is the distance between the location *i* and the location *j*, and a_j and b_j are the lower and upper time windows of the location *j*, respectively.

Taking into account the lower and upper time windows allows the implementation to optimize over these variables, thereby moving towards a lower cost feasible solution each iteration. As the solution route is constructed, we keep track of the cost of the route, which is a combination of the distance travelled, along with any waiting time or penalty costs for arriving at a location early or late, respectively. Once all the ants have generated solution routes, the route costs are compared in the compareSolution() function, where a lower cost is preferred. The local search 2-OPT is then applied to the route of the best ant according to Algorithm 3, to see if there are any solutions in its neighbourhood which are feasible and have a lower cost. The global pheromones are then updated, first according to Eqn. 5 for the evaporation, and then the pheromone deposits according to Eqn. 5, which is based on only the single best ant, as this is a requirement of the MMAS. The final requirement of the MMAS is that the pheromone values are confined to the range $[\tau_{min}, \tau_{max}]$, according the Eqn. 6 and Eqn. 7. During the next iterations, these pheromone values will guide the ants towards the best solution route.

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} \quad (4)$$

$$\tau_{ij} = \tau_{ij} + \begin{cases} \frac{ck}{L} & \text{if edge}(i,j) \text{ is in the best route} \\ 0 & otherwise \end{cases} \quad (5)$$

$$\tau_{max} = \frac{ck}{L} \quad (6)$$

$$\tau_{min} = \frac{\tau_{max}}{2N} \quad (7)$$

Where ρ is the evaporation, *c* is a constant, *k* is the number of ants, *L* is the cost of the solution route of the best ant, and *N* is the number of locations.

In all our testing, we used an alpha and beta value of 1, a constant *c* for pheromone calculations of 1000, an initial pheromone value of 4, an evaporation value of 0.7, and an ant population size off 100.

Algorithm 1: Pseudocode overview of MMAS 2-OPT initializePheromones():

```
initializePheromones()
for i = 0 to max iterations do
        for k = 0 to number of ants do
                generateSolution()
        end for
        compareSolutions()
        apply2OPT(best_ant)
        updateGlobalPheromones(best_ant)
end for
```

Algorithm 2: Pseudocode to constructively generate a solution:

```
route[0] ← 0
for i = 1 to number of locations do
        for j = 1 to number of locations do
                if locations[j] is a feasible location then
                        probability[j] ← transitionProbability()
                else
                        probability[j]  ← 0
                end if
        end for
        route[i] ← use roulette wheel selection from probability[]
        Keep track of current time, current load, and which locations have been visited
end for
```

Algorithm 3: Pseudocode for 2-OPT local search:

```
while improvement is made do
        start_marker:
        best_cost ← getRouteCost(existing_route)
        for i = 1 to number of locations to swap – 1 do
                for k = (i + 1) to number of locations to swap do
                        new_route ← exisiting_route[0] to existing_route[i – 1]
                                + reverse(existing_route[i] to existing_route[k])
                                + existing_route[k+1] to end
                        new_cost ← getRouteCost(new_route)
                        if new_cost < best_cost then
                                existing_route ← new_route
                                best_cost ← new_cost
                                goto start_marker
                        end if
                end for
        end for
end while
```

Tabu Search Algorithm

The Tabu Search algorithm was created by Fred W. Glover. It is a metaheuristic that uses local search methods for optimization in combinatorial problems. The purpose of the algorithm is to move from an original solution to an improved solution. This improved solution is within the neighbourhood of the original solution.

The word "tabu" indicates things that cannot be touched because they are sacred, hence the introduction of a tabu list. Since local search methods are often trapped in a local optimum, the tabu list is used. This list helps stop the algorithm from visiting previously visited solutions and becoming trapped in a cycle. The way the list works is that it is used as a set of rules or banned solutions that restrict certain solutions from being entered into the neighbourhood, thus introducing other potential solutions that may have never been explored.

The tabu search algorithm can be applied to scheduling, sequencing, telecommunications, resource allocation, and many other areas. The algorithm is also based on concepts that enable it to be used in artificial intelligence [27].

Limitations:

- Tabu list construction is unique for every problem
- No guarantee of global optimal solutions [29]

Tabu Search with respect to the routing problem

In terms of the PDPTW, an initial solution was represented as a Route, which contained an array of integers, where each integer represented a drop-off or pick-up point, and the first value was 0, which represents the depot. The Tabu List was then initiated which contained all the "sacred" Routes. A neighborhood of Routes was used. This stored all the candidate solutions that were generated by the PD-RearrangeOperator [30]. The fitness function was determined based on the total distance of the Route and the time the vehicle spent waiting at a location to open. Only feasible solutions were allowed into the Tabu List and the neighborhood. A feasible solution is one where the capacity of the vehicle is not exceeded, and the vehicle does not arrive at any location after the upper-time-window. Once a new solution has been deemed the best overall solution, it is then assigned as such, and the previous best overall solution is added to the Tabu List. A better solution is deemed when its fitness value is lower than the fitness value of the solution it is being compared to.

Operator: PD-RearrangeOperator

The neighbourhood of candidates is populated using the PD-RearrangeOperator [27]. The PD-RearrangeOperator obtains a pick-up and drop-off pair and inserts them into a new feasible location. This is all done within the same Route. An illustration is depicted in Figure 2.

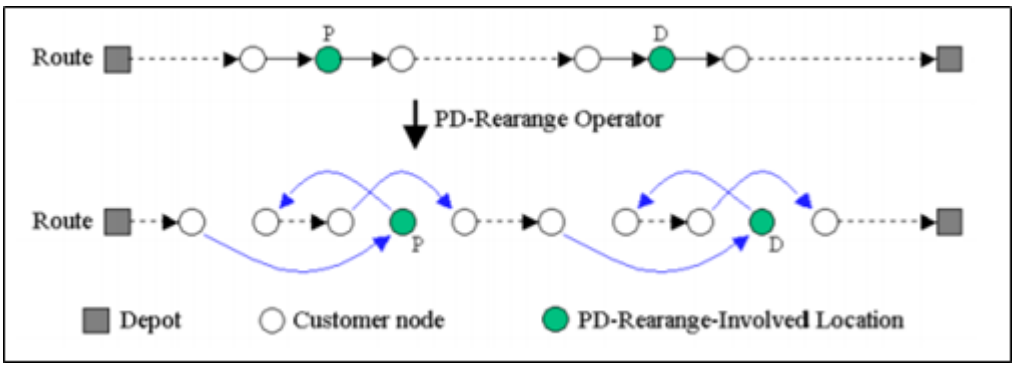

Figure 2: PD-Rearrange Operator [27]

Our implementation

Representation: A Route was represented by an array of integers, where each integer is either a pick-up of drop-of location, and the initial value of the Route is reserved for the depot. The length of the Route is equal to the amount of locations (depot + drop-off locations + pick-up locations). Each pick-up location had a value that represented the weight of the goods to be picked up. This was then added onto the vehicle's capacity upon pick-up. All locations had a distance between them, which was calculated using the Euclidean Distance formula. Each location also stored its lower-time window and upper-time window. This was used to ensure the vehicle arrives before the upper-time window for each location.

Fitness Function: The total distance travelled by the vehicle between each drop-off and pick-up location, as well as the time that the vehicle had to wait until the lower-time window of the location had arrived, was used to calculate the fitness value.

Algorithm 4: Pseudocode for the implemented Tabu Search:

Generate a feasible solution: x^i
bestSolution = x^i
bestCandidate = $bestSolution$
timesImprovedSolutionNotFound
Initialize Tabu List: tabuList ←[ ]

while(¬stoppingCondition()) **do**
generate neighborhood of candidates using PDRearrangeOperator from bestCandidate
bestCandidate = bestCandidateFromNeighborhood()
if (bestCandidate.isFeasible() and bestCandidate.fitnessValue < bestSolution.fitnessValue)**then**
if (¬tabuList.contains(bestCandidate)**then**
bestSolution = bestCandidate
end if
end if
if(timesImprovedSolutionNotObtainedReached)**then**
bestCandidate = obtainSolutionLargerThanCurrentSolutionFromNeighbourhood()
end if
updateTabuList()
end while
return bestSolution

Cuckoo Search Algorithm

The cuckoo search is a metaheuristic technique developed by XS Yang based on the behaviour of cuckoo birds during reproduction. The cuckoo bird (female) partakes in what is known as brood parasitism, where it will lay an egg in another host bird's nest resulting in that egg being fed and raised by the host. The host bird may be able to recognize the egg amongst its own thus will reject it by either throwing it off or abandoning the nest. The cuckoo can randomly search for potential nest amongst a population via Levy Flights

The algorithm is modelled after this behaviour. In the cuckoo search, the egg in a nest represents the solution of an individual amongst a population. The cuckoo's egg is a potential solution to be included or inserted into the population. The nest is the individual amongst the population of nests. The nest in the population can be removed by replacing it with another nest with the old nest representing an abandoned one.

Limitations:

- There is currently no theoretical analysis of how the cuckoo works, unlike GA [31] and PSO [32], and this is very problematic for debugging.
- Might converge too quickly to a solution which is not feasible.
- Tends to get stuck in a local optimal.
- It requires a better exploration strategy for finding optimal and feasible solutions.

The egg

The egg represents the potential solution in a nest. The egg is thus a route from the depot to the last delivery location. The cuckoo will try to lay an egg that will not be discovered by the host bird in the nest. Figure 3 shows the egg representation in the cuckoo search algorithm. The vehicle will start at depot 0 and traverse the locations until location 10, where it will stop. The pickup locations are the odd numbers 1, 3, 5, 6, 7, and 9. The delivery locations are the even numbers 2, 4, 6, 8, and 10.

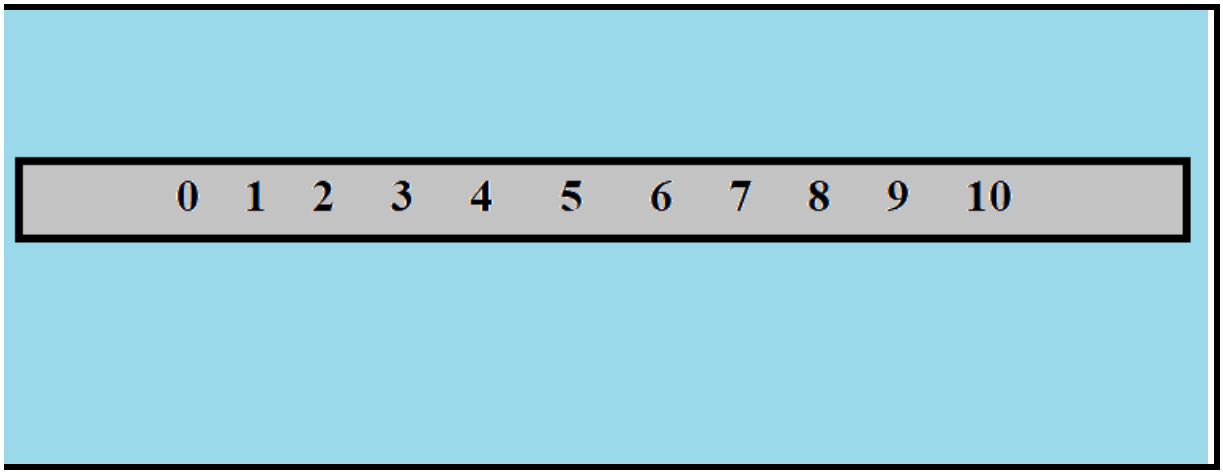

Figure 3: The representation of an egg in the cuckoo search.

Typical eggs will be permutations of the representation while maintaining the precedence constraint. At no point during the algorithm execution should the precedence constraint be violated.

Fitness Function

The egg has an associated fitness value associated with it. The fitness measures the quality of the egg; thus, the lower the fitness value, the more likely the egg will remain in the population. The cuckoo will attempt to insert its egg by producing an egg with an improved fitness such that the host will not discover it. The fitness in the cuckoo search is the sum of the total time travelled from the depot to the final location, the penalty incurred by overloading the vehicle capacity, and the penalty for arriving late. The mathematical formulation of the fitness function $f(E)$ is:

$$f(E) = Z_{travel\ cost}(E) + z_{penalty}(E) + c_{validity} \tag{8}$$

Where $Z_{travel\ cost}(E)$ is the total travelled time by the vehicle in route E, $z_{penalty}(E)$ is the penalty incurred for violating the capacity constraint (overload) and the upper time window constraint (late arrival).

The nest

The nest may have one or more eggs contained inside it. The cuckoo will then replace one or more of those eggs. The bird will reject the eggs (by ejecting them) or abandon it completely and instead build a new nest. The nest in the cuckoo search implementation contains only one egg; thus, the host will either accept the egg or reject it and leave for another nest.

The population

The population consists of n nests which have their respective fitness value. The population in the cuckoo nest is a double array ($M \times N$) with M rows, each representing a nest with an egg, and N columns, each representing the location in the nest. The cuckoo will fly around and pick a nest then generate an egg via Levy flights.

The Levy Flight and Neighborhood structure

The cuckoo flies to a random nest and lays an egg via Levy Flights. The nest was picked randomly then a Levy walk was applied to the nest's egg. The Levy walk operated on the egg using one of the neighborhood operations. Three neighborhood structures were applied based on the value obtained from the Levy function. The Levy walk was performed using Mantegna's algorithm;

$$LevyWalk = \begin{cases} 2 - PSwap, & if \ \frac{u}{v^{-\lambda}} < 0.45 \\ 4 - PSwap & if \ 0.45 \leq \frac{u}{v^{-\lambda}} < 0.95 \\ 6 - PSwap & if \ 0.95 \leq \frac{u}{v^{-\lambda}} \end{cases} \quad (9)$$

where $u \sim N(0, \sigma^2)$ is a random number drawn between values, σ^2 is the variance of value 1.44 obtained when λ=1.5, and $v \sim N(0,1)$.

The 2-point Swap Local Search

The 2-point swap is a local search method that swaps two adjacent points in a route. All the possible swaps are searched to find the best solution. Figure 4 shows a typical swap in a 2-point search and Algorithm 5 shows the algorithm.

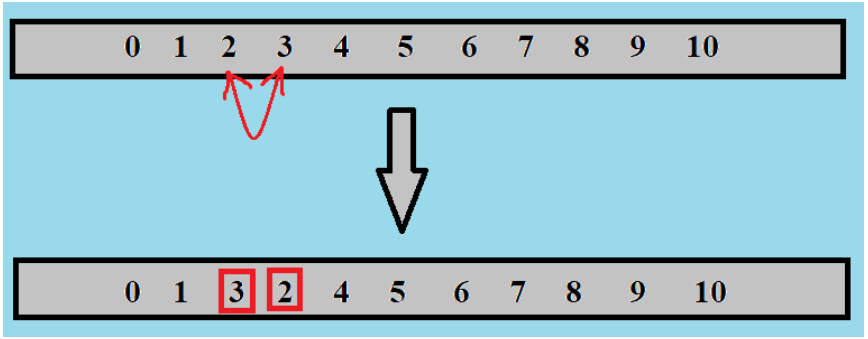

Figure 4: A swap in the 2-point swap search.

Algorithm 5: 2-Point Swap Search

Start search for Route:
bestRoute, fitNessOfBestRoute
loop:
for: i=2 in Route
    if precedence coinstraint is not violated **then**
    newRoute = swap(Route, i, i-1)
    fitnessOfNewRoute = fitnessFunction(newRoute)
    if(fitnessOfNewRoute<=fitNessOfBestRoute)
        bestRoute = newRoute
        fitNessOfBestRoute = fitnessOfNewRoute
    i++
end after loop termination

The 4-point Swap Local Search

The 4-point swap is a local search method that swaps four adjacent points in a route. All the possible swaps are searched to find the best solution. Figure 5 shows a typical swap in a 4-point search and Algorithm 6 shows the algorithm. Two different pickup locations are selected and swapped, and their respective delivery locations are also obtained and swapped.

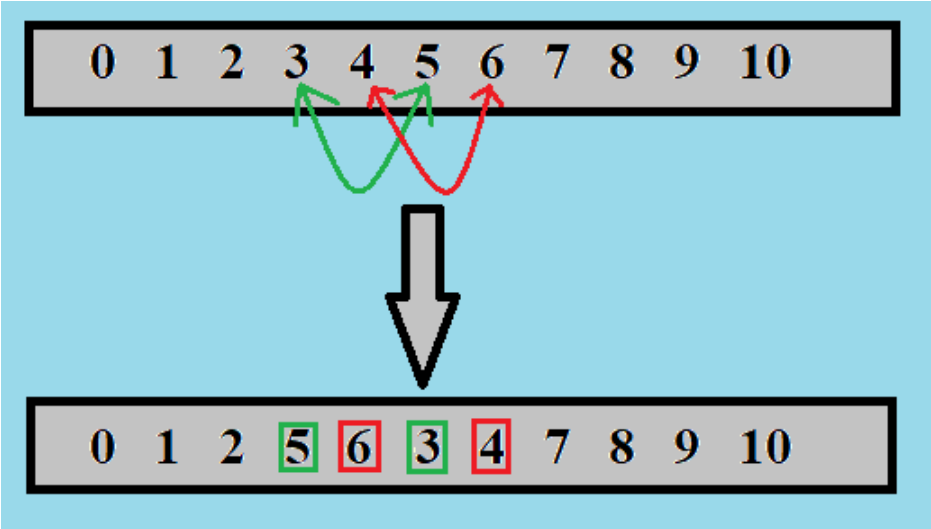

Figure 5: A swap in the 4-point swap search.

Algorithm 6: 4-Point Swap Search

```
Start search for Route:
bestRoute, fitNessOfBestRoute
loop:
for: i=1 in PickUpSet
        routePickUpIndex1 = getIndexInRouteOf(PickUpSet[i])
        routeDeliveryIndex1 = findDeliveryIndex(Route, routePickUpIndex1)
        for j = i+1 in PickUpSet
                routePickUpIndex2 = getIndexInRouteOf(PickUpSet[i])
                routeDeliveryIndex2 = findDeliveryIndex(Route, routePickUpIndex2)
                newRoute = swap(Route, routePickUpIndex1, routePickUpIndex2)
                newRoute = swap(Route, routeDeliveryIndex1, routeDeliveryIndex2)
                fitnessOfNewRoute = fitnessFunction(newRoute)
                if(fitnessOfNewRoute<=fitNessOfBestRoute)
                        bestRoute = newRoute
                        fitNessOfBestRoute = fitnessOfNewRoute
                j++
        i++
end after loop termination
```

4.6.3 The 6-point Swap Local Search

The 6-point swap is a local search method that swaps six adjacent points in a route and is similar to the 4-point swap search, albeit with an extra loop. All the possible swaps are searched to find the best solution. Figure 6 shows a typical swap in a 6-point search and Algorithm 7 shows the algorithm. Three different pickup locations are selected and swapped, and their respective delivery locations are also obtained and exchanged.

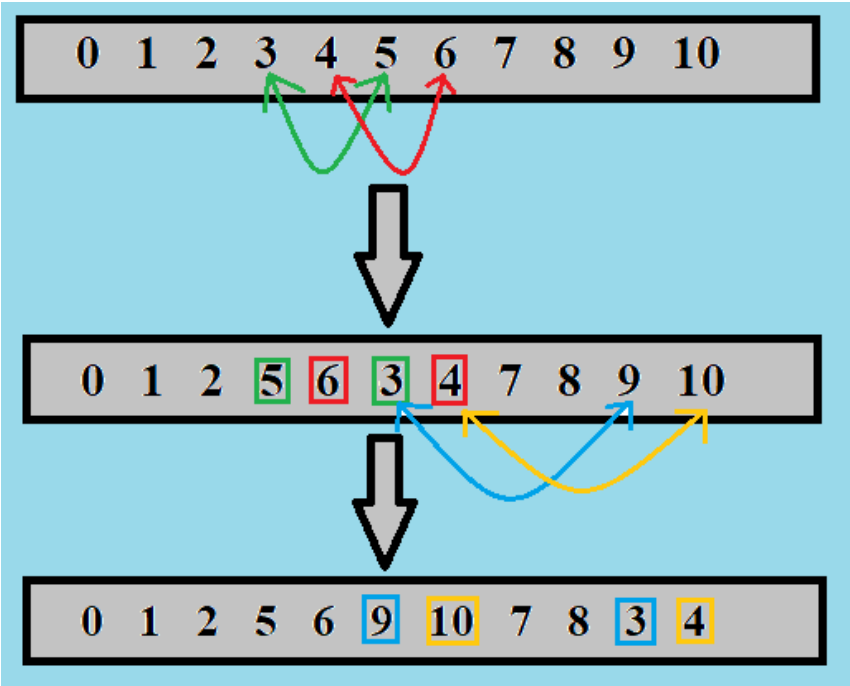

Figure 6: A swap in the 6-point swap search.

Algorithm 7: 4-Point Swap Search

```
Start search for Route:
bestRoute, fitNessOfBestRoute
loop:
for: i=1 in PickUpSet
        routePickUpIndex1 = getIndexInRouteOf(PickUpSet[i])
        routeDeliveryIndex1 = findDeliveryIndex(Route, routePickUpIndex1)
        for j = i+1 in PickUpSet
                routePickUpIndex2 = getIndexInRouteOf(PickUpSet[j])
                routeDeliveryIndex2 = findDeliveryIndex(Route, routePickUpIndex2)
                newRoute = swap(Route, routePickUpIndex1, routePickUpIndex2)
                newRoute = swap(Route, routeDeliveryIndex1, routeDeliveryIndex2))
                fitnessOfNewRoute = fitnessFunction(newRoute)
                for k = j+1 in PickUpSet
                        routePickUpIndex3 = getIndexInRouteOf(PickUpSet[k])
                        routeDeliveryIndex3 = findDeliveryIndex(Route, routePickUpIndex3)
                        newRoute = swap(Route, routePickUpIndex2, routePickUpIndex3)
                        newRoute = swap(Route, routeDeliveryIndex2, routeDeliveryIndex3))
                        fitnessOfNewRoute = fitnessFunction(newRoute)
                        if(fitnessOfNewRoute<=fitNessOfBestRoute)
                                bestRoute = newRoute
                                fitNessOfBestRoute = fitnessOfNewRoute
                j++
        i++
end after loop termination
```

4.6.4 The Cuckoo Search Pseudocode

The pseudocode for the cuckoo search is shown in Algorithm 8. The algorithm begins by initializing the nests randomly. Then the cuckoo uses Levy walks to construct a new solution from a random nest. A random nest is chosen from the nest population and its egg's fitness is compared with the fitness of the cuckoo's egg. If the

cuckoo's fitness is better than that of the nest, the nest's egg is replaced with the cuckoo's egg. All the worst-performing nests are replaced with new ones. The nests are ranked by their fitness, and the current best solution found.

Algorithm 8: Cuckoo Search

Start search for Route:
bestRoute, fitNessOfBestRoute
loop:
while (iter < MaxGeneration or Terminating Criterion)
    Create a cuckoo randomly using Levy Flight
    Calculate the cuckoo fitness f_i
    Pick a random nest among n and evaluate its fitness f_j
    if ($f_j > f_i$) **then**
        replace nest j with cuckoo i
    remove p_a fraction of nests from population
    Generate new nests via Levy flights
    Rank the solutions to determine the current best solution

end after loop termination

VI. Simulation Experiment Details

The implementations were developed and tested with Java in IntelliJ IDE Community Edition 2020 on Windows 10.

The following hardware was used to perform the testing of each metaheuristic algorithm:

- Intel® Core™ i7-7700K CPU @ 4.20GHz with 16GB of RAM

Dataset

While there are many benchmark problems for TSP and even VRPTW, unfortunately these do not suit the PDPTW because they have fewer constraints. Therefore, the dataset used was randomly generated according to Alg. 9, which is the same way in which our chosen paper generated their dataset [5].

Algorithm 9: Pseudocode for single vehicle PDPTW dataset generation.

for i to number of requests **do**
    Generate a pickup and a delivery location
    Generate a load q_i, where pickup load = q_i and delivery load = $-q_i$
end for
Calculate distance d_r^s between every location *r* and *s*
Randomly generate a route which satisfies the precedence constraint
Let t_i be the arrival time of location i
Let $AverageTime$ be the average traveling time between locations
for i to number of locations **do**
    $a_i = t_i - (random(width) \times AverageTime$
    $b_i = t_i + (random(width) \times AverageTime$
end for
$[a_i, b_i]$ is the time window for location i

VII. Results and Discussion

Since our computer specifications were not the same as the other articles we have referenced, we decided to compare our results between themselves, as they were all run under the same conditions on the same computer. This way, other researchers would be able to gauge the comparative performance of these three metaheuristic algorithms on the PDPTW, and from that make a decision on whether they want to further research on one of the algorithms.

We had a total of 10 test sets, which were randomly generated using the aforementioned dataset generator. Each test was run 10 times for each algorithm, allowing us to extract average performances. Because of the different approaches of each algorithm, we could not compare them based on their performance per iteration. Therefore, we ran each test for a maximum of one minute, or until the algorithm had converged, whichever came first. This allowed us to compare the results based on four criteria: The number of valid solutions found, the best and average cost per valid solution, the best and average time to find the first valid solution, and the average number of constraints broken for invalid solutions.

The PDPTW can be difficult to solve, or even just to find initial feasible solutions, because of its large number of constraints, which is why we report on the number of constraints broken for the test cases which did not find a valid solution.

Looking at Table 1 and Figure 7, we see that the MMAS 2-OPT was able to find valid solutions in all 10 tests of all 10 task sizes. The Tabu Search was close behind, finding valid solutions in all but one of the tests for the largest task size of 50. The Cuckoo Search was able to find valid solutions every time for the smaller half of the task sizes, however, its ability to find valid solutions declined from a task size of 30 onwards. We will make a note here that the Cuckoo Search was unable to find valid solutions for the task size of 35, but this will be discussed later.

Since we had no way to calculate the optimal cost values, we can only compare the performance between our own algorithms. We see that the MMAS had the best average cost for all task sizes, while the Tabu Search and Cuckoo search had equal average costs for 4 and 1 of the test cases, respectively. These were notably the smaller test cases. This shows that the MMAS was more consistently finding lower cost values. The Tabu Search and the Cuckoo Search both had 5 best costs equal to that of the MMAS, and one best cost which exceeded the performance of the MMAS, while the MMAS had 9 out of 10 best costs. The Tabu Search and Cuckoo Search outperformed the MMAS in terms of the best cost for the task size of 35, which is a notably large size. This shows that while the MMAS seemed to be performing more consistently throughout the tests, it was still possible to beat it. Overall, the difference between best and average costs of the algorithms was very small, especially when shown is a graph such as in Figure 8 and Figure 9.

However, when investigating Table 2, along with Figure 10 and Figure 11, we see that the MMAS outperformed the other algorithms by far in terms of the time to find a first valid solution, especially in the larger task sizes. While the consistency of the Cuckoo Search started to worsen with the large task sizes, it still managed to outperform the Tabu Search in terms of time to find the first valid solution. This shows that the Cuckoo search may be a good second choice if we are able to adapt it to favour exploration slightly more. The Tabu Search was able to find instant valid solutions for the smaller task sizes, with a more consistent speed than the other two algorithms, however the time it required to find a valid solution rose quickly as the task size increased. This shows that it may not perform as well as the other two algorithms for much larger applications. While it may at first seem odd that the Cuckoo Search was unable to find valid solutions for a task size of 35, but then managed to find them for larger task sizes, we see that the Tabu Search and MMAS also had difficulty with this task size, with higher than average times in this task size, indicating that the specific test case which was generated for the task size of 35 may have been more complex than usual. We expect that if a different test case of the same size was used, the Cuckoo search would have probably found between 3 and 6 valid solutions, based on the amount of valid solutions found for task sizes of 40 and 40, respectively.

Table 3 shows that while the Cuckoo Search was obtaining fewer valid solutions as the task size increased, it was not far off a valid solution. The number of broken constraints remained small, with almost all below 10. When comparing these values in Figure 12, we see that only a very small fraction of the total constraints was being broken by the Cuckoo Search. Figure 13 shows that the number of broken constraints for the Cuckoo Search was rising steadily as the task size got larger. The Tabu Search broke only 2 constraints in the largest task size, and the MMAS did not break any.

The total number of constraints was calculated according to the following formula:

Total constraints = 5L/2 - 2, where L is the number of locations.

This was based on the fact that, when we have L locations, we can break a total of L/2 precedence constraints, L-2 capacity constraints, and L time window constraints. This does not take into account the constraint of only visiting each location once.

Best and average cost of routes:

Task Size	Cuckoo Search			Tabu Search			MMAS 2OPT		
	Valid	Best	Avg	Valid	Best	Avg	Valid	Best	Avg
5	10/10	436	436	10/10	436	436	10/10	436	436
10	10/10	923	925.7	10/10	923	923	10/10	923	923
15	10/10	1376	1440.7	10/10	1376	1412.2	10/10	1376	1387.2
20	10/10	1681	1685.6	10/10	1681	1681	10/10	1681	1681
25	10/10	2152	2230.5	10/10	2152	2238.8	10/10	2116	2116
30	6/10	2613	2614.2	10/10	2613	2613	10/10	2613	2613
35	0/10	-	-	10/10	3130	3201.4	10/10	2800	2861.9
40	3/10	3906	3935.7	10/10	3906	3974.4	10/10	3923	3945.7
45	2/10	4197	4207	10/10	4004	4055.4	10/10	3961	3995.1
50	0/10	-	-	9/10	4951	5031.2	10/10	4783	4884

Table 1: Best and average cost of routes

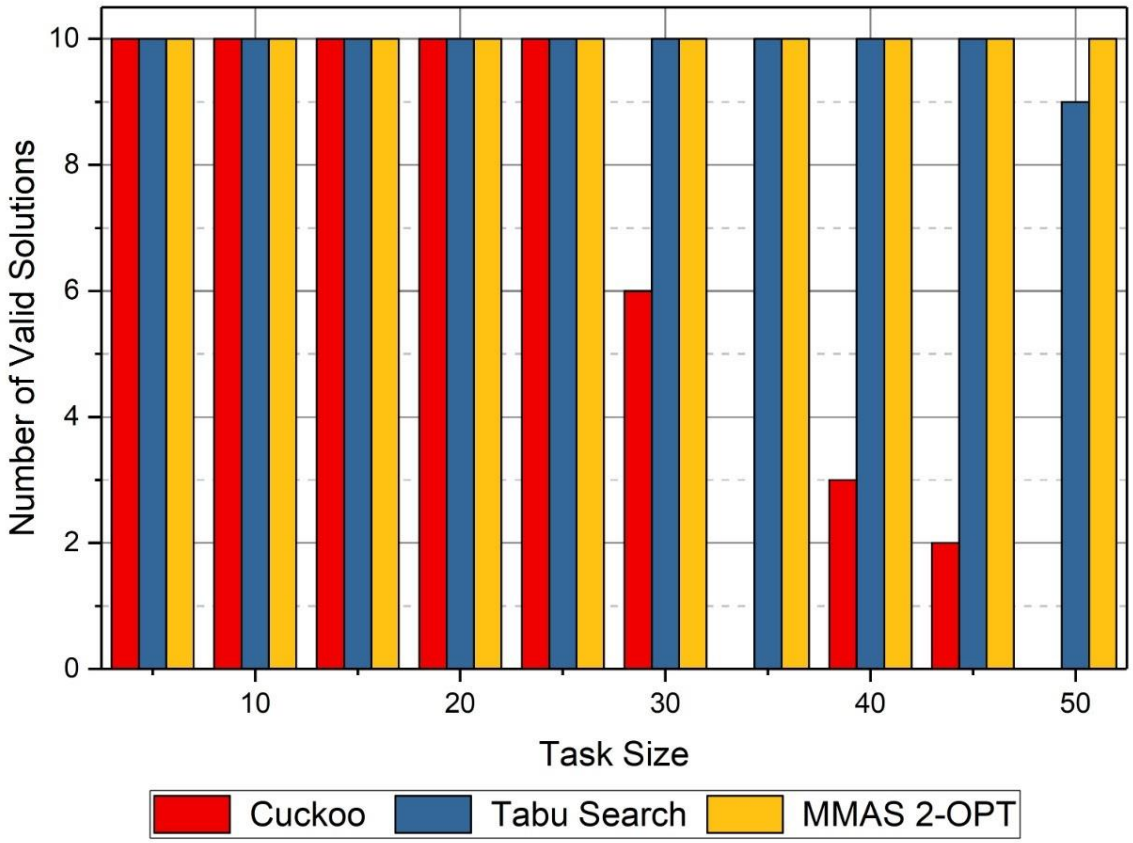

Figure 7: Number of valid solutions per task size.

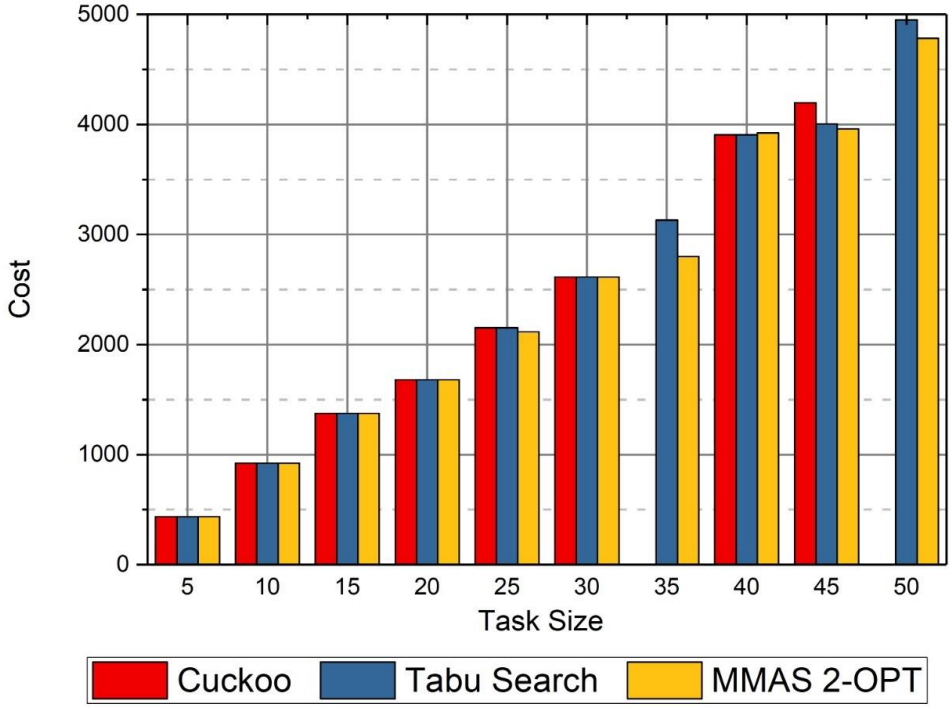

Figure 8: Best cost per task size.

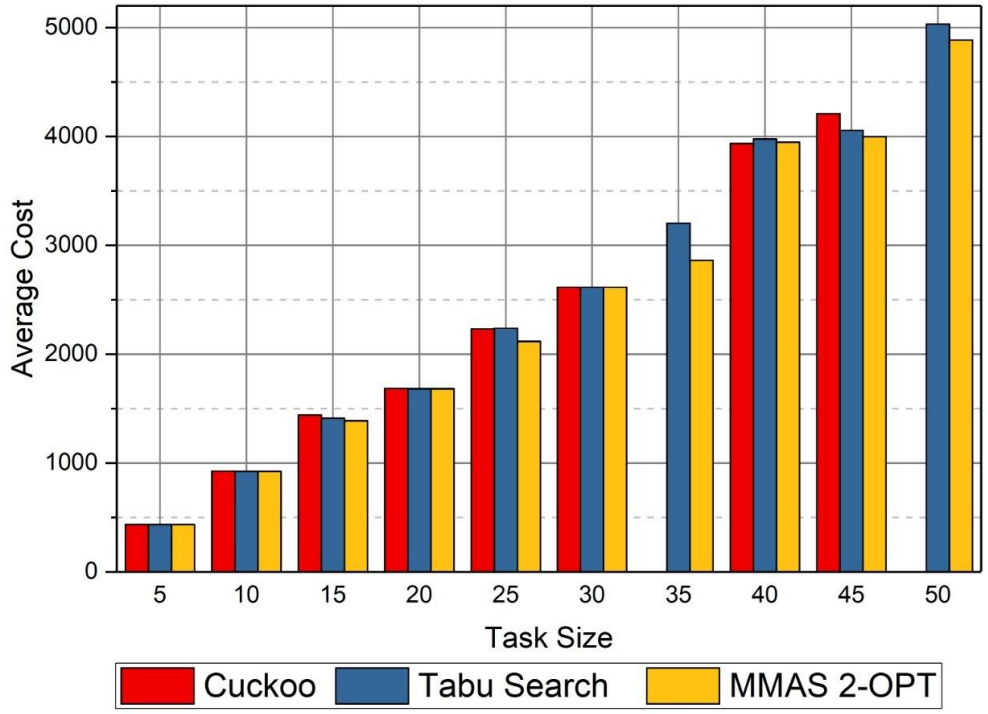

Figure 9: Average cost per task size.

Time taken to find first valid solution:

Task Size	Cuckoo Search			Tabu Search			MMAS 2OPT		
	Valid	**Best**	**Avg**	**Valid**	**Best**	**Avg**	**Valid**	**Best**	**Avg**
5	10/10	0	0	10/10	0	0	10/10	0	0
10	10/10	0	0	10/10	0	0	10/10	0	1
15	10/10	0	3	10/10	0	0	10/10	0	1
20	10/10	5	32	10/10	1040	1227	10/10	1	2

25	10/10	22	86	10/10	428	477	10/10	1	2
30	6/10	294	969	10/10	820	848	10/10	4	8
35	0/10	-	-	10/10	19829	20186	10/10	66	378
40	3/10	1735	5776	10/10	6957	10581	10/10	15	134
45	2/10	2717	3602	10/10	23001	23201	10/10	28	468
50	0/10	-	-	9/10	36128	43921	10/10	112	962

Table 2: Time taken to find the first valid solution.

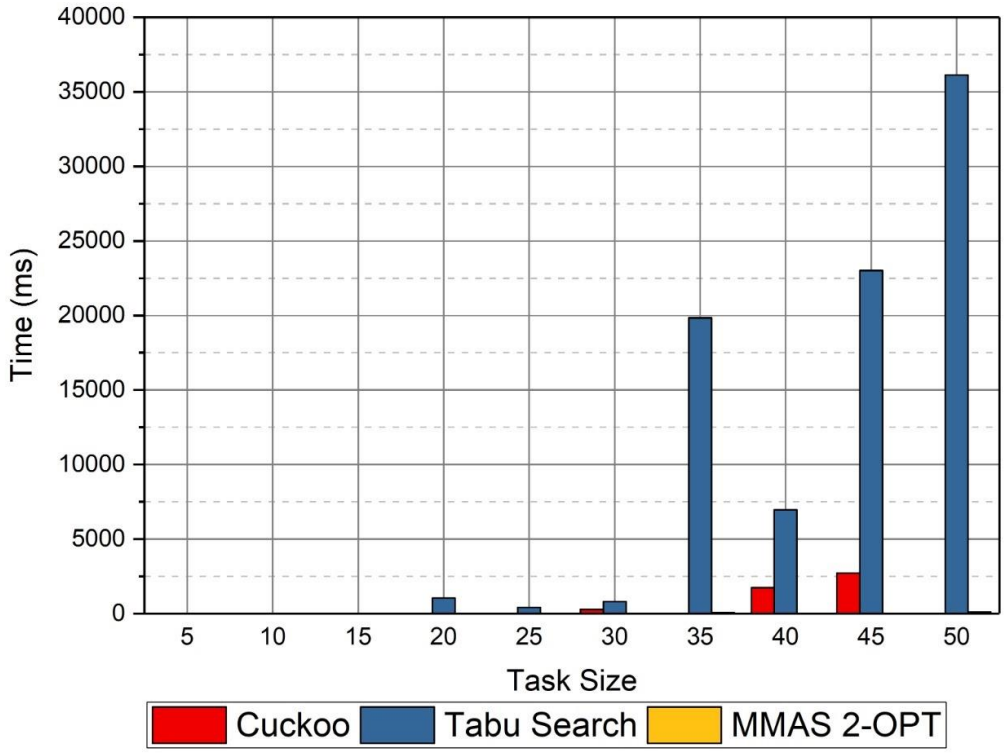

Figure 10: Best time to find first valid solution per task size.

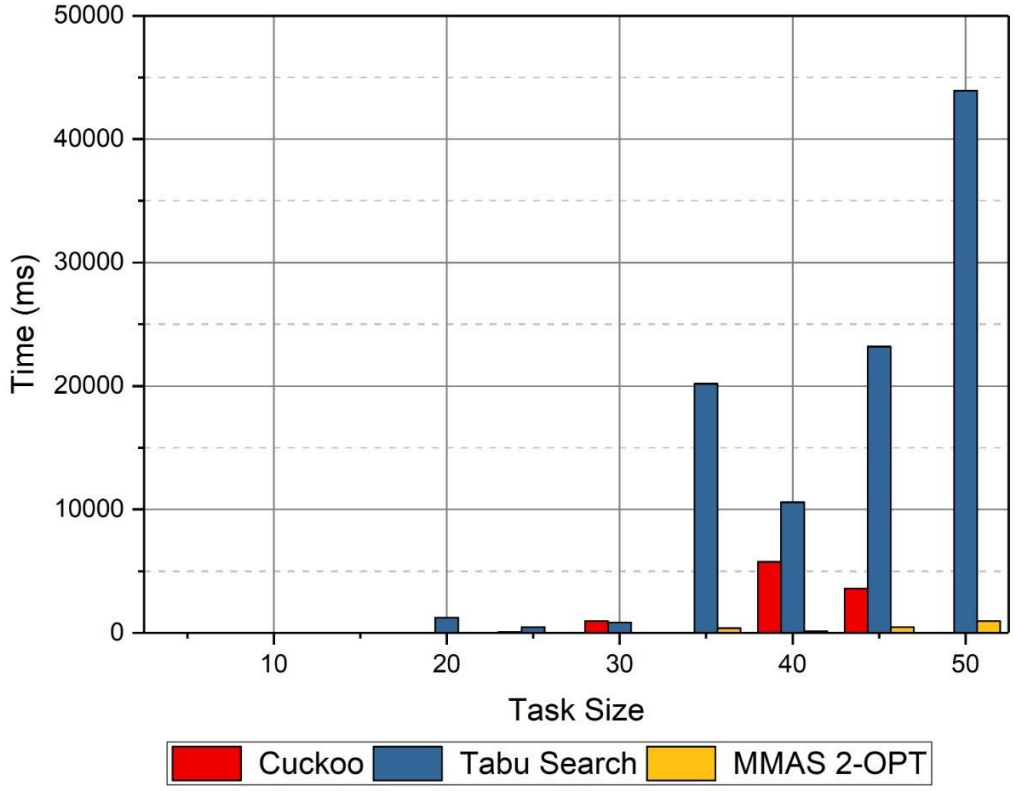

Figure 11: Average time to find first valid solution per task size.

Constraints broken:

*MMAS was not included in this table because it did not break any constraints.

Task Size	Total Constraints	Cuckoo Search			Tabu Search		
		Valid	Best	Avg	Valid	Best	Avg
5	23	10/10	0	0	10/10	0	0
10	48	10/10	0	0	10/10	0	0
15	73	10/10	0	0	10/10	0	0

20	98	10/10	0	0	10/10	0	0
25	123	10/10	0	0	10/10	0	0
30	148	6/10	0	3.8	10/10	0	0
35	173	0/10	6	7.2	10/10	0	0
40	198	3/10	0	8.7	10/10	0	0
45	223	2/10	0	8.9	10/10	0	0
50	248	0/10	7	11.4	9/10	0	2

Table 3: Number of constraints broken

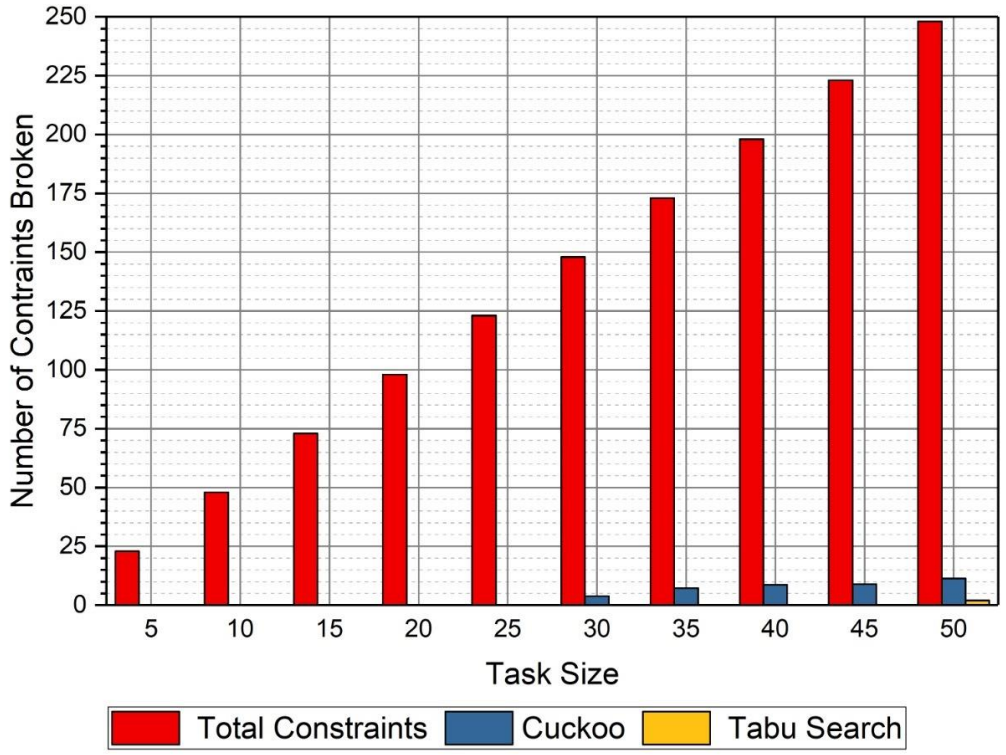

Figure 12: Average number of constraints broken per task size, including the total constraints.

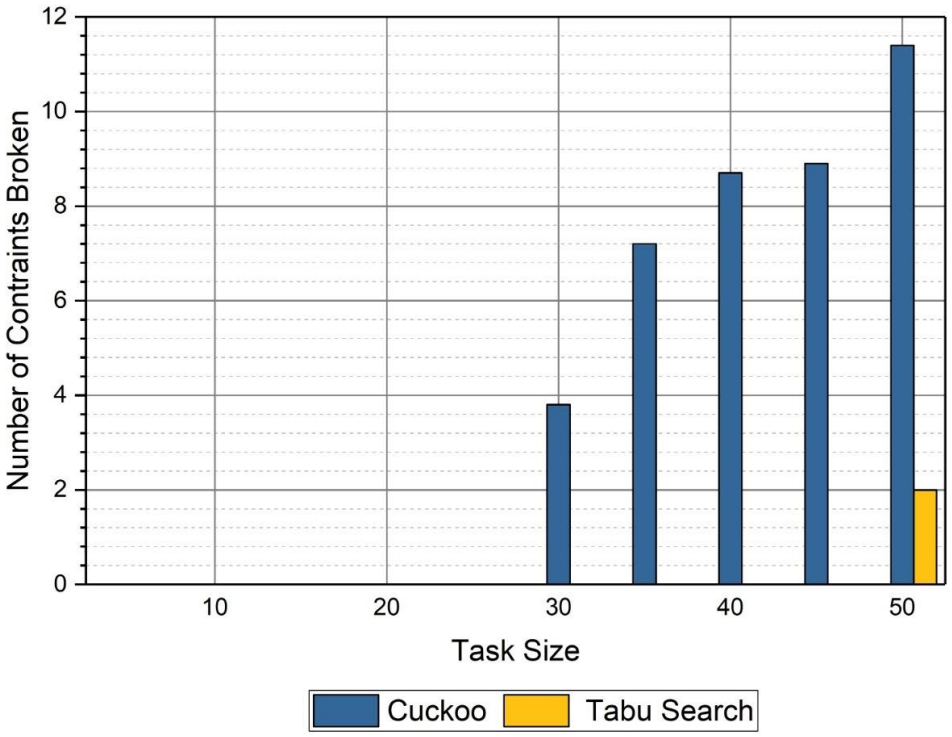

Figure 13: Average number of constraints broken per task size, not including the total number of constraints.

VIII. Conclusion

Overall, the hybrid MMAS 2-OPT was the best performer, in terms of best cost, consistency, and time to find the first valid solution. This is likely due to the Max-Min adaption of the Ant System, as it first favours exploration to find a valid solution quickly, and then swaps to favouring exploitation to focus on a smaller neighbourhood of solutions and optimize the cost sufficiently. The Tabu Search was a close second in terms of the best cost, and was able to find valid solutions for almost every test, however it took much longer than the other algorithms to find valid solutions. This shows it would be suitable for small task sizes, but may not be efficient enough for large ones in its current state. If the efficiency could be improved, it would rival the MMAS. In the future, we would like to look into other operators for the Tabu Search, which would generate neighbourhoods to increase the search space. The Cuckoo Search was the worst performer in terms of finding valid solutions, however when it did find valid solutions, they were of comparable performance to the other algorithms, and found in a timely manner. From the testing, it seemed like the Cuckoo Search was converging to a solution too quickly, and for the larger task sizes this meant not finding a valid solution. Once the algorithm became trapped in a local optimum, it was not able to continue finding better solutions, showing that it favoured exploitation too highly over exploration. With some further research of the parameters, and testing of different such values, the Cuckoo Search may produce better results. In the future, we would like to try combining two or more of our algorithms to make a new hybrid algorithm, to see if we can improve on the best performance in this report. It would also be interesting to increase the task size of our datasets, and investigate up to what size our algorithms continue to find valid solutions. If we could find benchmark problems for PDPTW or a similar problem, which have known optimal solutions, we would like to test our algorithms on these, and see how our performance compares to state of the art algorithms.

IX. References

[1] Savelsbergh, M. W., and Sol, M. (1995). The General Pickup and Delivery Problem. *Transportation Science*, *29*(1), 17-29. doi:10.1287/trsc.29.1.17

[2] Hosny, M.I, and Mumford, C.L. (2008). The single vehicle pickup and delivery problem with time windows: intelligent operators for heuristic and metaheuristic algorithms. *Journal of Heuristics*, 16(3), 417-439. doi:10.1007/s10732-008-9083-1

[3] Dantzig, G. B., and Ramser, J. H. (1959). The Truck Dispatching Problem. *Management Science*, *6*(1), 80-91. doi:10.1287/mnsc.6.1.80

[4] Landrieu, A., Mati, Y., and Binder, Z. (2001). *Journal of Intelligent Manufacturing*, *12*(5/6), 497-508. doi:10.1023/a:1012204504849

[5] Jih, W., Kao, C., & Hsu, J. Y. (2002). Using family competition genetic algorithm in pickup and delivery problem with time window constraints. *Proceedings of the IEEE Internatinal Symposium on Intelligent Control*. doi:10.1109/isic.2002.1157813

[6] Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). Genetic Programming – An Introduction. San Francisco, CA: Morgan Kaufmann. ISBN: 978-1558605107.

[7] Dorigo, M. and Stützle, T. (2003). Ant Colony Optimization. MIT Press, Boston, MA. ISBN: 9780262042192

[8] Dorigo, M. and Stützle, T. (2004). Ant Colony Optimization, p.12.

[9] Fladerer, J., and Kurzmann, E. (2019). WISDOM OF THE MANY: how to create self -organisation and how to use collective... intelligence in companies and in society from mana. ISBN 9783750422421

[10] Stützle, T. and Hoos, H.H. (1999). Max-Min Ant System. *Future Generation Computer Systems*, 16(8), 889-914.

[11] Yang, X., and Deb, S. (2009). Cuckoo Search via Lévy flights. *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, 210–214.

[12] Shehab, M., Khader, A.T., and Al-Betar, M.A. (2017). A survey on applications and variants of the cuckoo search algorithm. *Applied Soft Computing*, *61*, 1041–1059. doi:10.1016/j.asoc.2017.02.034

[13] Giveki, D., Salimi, H., Bahmanyar, G., and Khademian, Y. (2012). Automatic Detection of Diabetes Diagnosis using Feature Weighted Support Vector Machines based on Mutual Information and Modified Cuckoo Search. *CoRR, abs/1201.2173*. http://arxiv.org/abs/1201.2173

[14] Liu, X., and Fu, H. (2014). PSO-based support vector machine with cuckoo search technique for clinical disease diagnoses. *TheScientificWorldJournal*, *2014*, 548483–548483. PubMed. doi:10.1155/2014/548483

[15] Tran, C.D., Dao, T.T., and Vo, V.S. (2020). Economic Load Dispatch With Multiple Fuel Options and Valve Point Effect Using Cuckoo Search Algorithm With Different Distributions. *International Journal of Energy Optimization and Engineering (IJEOE).*

[16] Basu, M., and Chowdhury, A. (2013). Cuckoo search algorithm for economic dispatch. *Energy*, *60*, 99–108. doi:10.1016/j.energy.2013.07.011

[17] Gandomi, A.H., Talatahari, S., Yang, X.-S., and Deb, S. (2013). Design optimization of truss structures using cuckoo search algorithm. *The Structural Design of Tall and Special Buildings*, *22*(17), 1330–1349. doi:10.1002/tal.1033

[18] Ahmed, J., and Salam, Z. (2014). A Maximum Power Point Tracking (MPPT) for PV system using Cuckoo Search with partial shading capability. *Applied Energy*, *119*, 118–130. doi:10.1016/j.apenergy.2013.12.062

[19] Pare, S., Kumar, A., Bajaj, V., and Singh, G. K. (2016). A multilevel color image segmentation technique based on cuckoo search algorithm and energy curve. *Applied Soft Computing*, *47*, 76–102. doi:10.1016/j.asoc.2016.05.040

[20] Ahmed, J., and Salam, Z. (2013). A soft computing MPPT for PV system based on cuckoo search algorithm. *Fourth International Conference on Power Engineering, Energy and Electrical Drives*, 558–562. doi:10.1109/PowerEng.2013.6635669

[21] Buaklee, W., and Hongesombut, K. (2013). Optimal DG allocation in a smart distribution grid using Cuckoo Search algorithm. *2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, 1–6.

[22] Ouaarab, A., Ahiod, B., and Yang, X.-S. (2014). Discrete cuckoo search algorithm for the travelling salesman problem. *Neural Computing and Applications*, *24*(7), 1659–1669. doi:10.1007/s00521-013-1402-2

[23] Ouyang, X., Zhou, Y.-Q., Luo, Q., and Chen, H. (2013). A Novel Discrete Cuckoo Search Algorithm for Spherical Traveling Salesman Problem. *Applied Mathematics & Information Sciences*, *7*, 777–784. doi:10.12785/amis/070248

[24] Mahmoudi, S., and Lotfi, S. (2015). Modified cuckoo optimization algorithm (MCOA) to solve graph coloring problem. *Applied Soft Computing*, *33*, 48–64. doi:10.1016/j.asoc.2015.04.020

[25] Essgaer, M., and Othman, Z.A. (2016). *Cuckoo search algorithm for capacitated vehicle routing problem*. *88*, 11–19.

[26] Kanagaraj, G., Ponnambalam, S.G., and Jawahar, N. (2013). A hybrid cuckoo search and genetic algorithm for reliability–redundancy allocation problems. *Computers & Industrial Engineering*, *66*(4), 1115–1124. doi:10.1016/j.cie.2013.08.003

[27] Pirim, H., Bayraktar, E., and Eksioglu, B. (2008). Tabu Search: A Comparative Study. *Tabu Search*. doi:10.5772/5637

[28] Dorigo, M., Maniezzo, V. and Alberto, C., (1996). The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1), pp. 1-13.

[29] Qu, D. (2016). Artificial Intelligence Methods. Retrieved 2020, from http://www.cs.nott.ac.uk/~pszrq/files/notesTS.pdf

[30] Li, H., and Lim, A. (n.d.). A metaheuristic for the pickup and delivery problem with time windows. *Proceedings 13th IEEE International Conference on Tools with Artificial Intelligence. ICTAI 2001*. doi:10.1109/ictai.2001.974461

[31] Wróblewski J. (1996). Theoretical foundations of order-based genetic algorithms. *Fundamenta Informaticae* 28 (3, 4) 423–430.

[32] Schmitt B.I. (2015). Convergence Analysis for Particle Swarm Optimization, FAU University Press.