Dynamic Parameter Tuning Method With Presets For Metaheuristics

Author
Kaiyu Wei

University Supervisors
Dr.Ir. W.J.A. van Heeswijk
Dr. E. Topan

Company Supervisor
Tomasz Górski

October 20, 2022

Management summary

Research background

DELMIA, a subsidiary of Dassault Systèmes, is a company focusing on various optimization problems in many fields, such as supply chain planning, sales and operation planning, logistics planning, workforce scheduling and workforce scheduling. Their platform DELMIA Quintiq 6.0 (in the year of 2022) is the latest version of software for solving optimization problems. There are some built-in metaheuristics in the software for addressing the problems. When using a metaheuristic, it is necessary to determine the value of parameters before running the model, such as the number of neighborhoods, the length of running time, or the times of iterations. For some metaheuristics, there are multiple parameters for which different values and their combinations will affect the performance of these metaheuristics.

The easiest way to determine the parameter values is just to set the values according to the user's experience. The user can set the parameter values he/she thinks to be sensible and effective for the metaheuristic. But this does not always work since a conjecture based on experience is not always reliable. Another way of setting parameter values is to do some pilot tests. For determining the parameter values, the optimization process is divided into two stages. First, we run some pilot tests for the metaheuristic with different parameter value settings, and then we compare the results from different settings and select the one that gives the metaheuristic the best performance. Then in the second stage, which is the formal training and optimization process, we use the metaheuristic with the selected value setting. However, the pilot-test method may take a long time before we can use the metaheuristic with the selected parameter values, so it may not be feasible for the situation that a solution is needed within a limited optimization time. Therefore, DELMIA needs a method to readily determine the value of metaheuristic parameters, without pilot tests in advance, whereby the appropriate parameter values can be selected while the optimization process is running. Such a method selects the effective parameter values while the metaheuristic is running without wasting time on pilot tests, and has better effectiveness than conjecture by experience.

In this paper, we design an online learning method to select the proper combination of parameters to give the running metaheuristic a good performance without pilot tests. The method runs simultaneously with the metaheuristic and learns about the attributes of different parameters and their values, named as *Dynamic parameter value tuning method* (DPTP). As mentioned above, the advantage of this method is that it needs no pilot test. This cuts down on running time for users and gives a good selection of parameter values. Existing parameter-control methods mostly handle the situation where there is only one parameter, while DPTP can manage multiple parameters at the same time. We provide "presets", which are alternative combinations of parameter values, for DPTP so that it can dynamically select different value settings and apply the selected values to the metaheuristic.

DPTP method

Before using DPTP, users need to provide it with sets of parameter values, i.e. presets, as mentioned above, to be selected by DPTP. These presets can be generated randomly from their value ranges.

DPTP gives each preset a selection probability. DPTP selects one preset in each epoch based on the selection probabilities of these alternative presets, and gives some feedback about the improvements of the benchmark function value obtained by using the selected preset, compared to the value before it is used, which can be seen as the effectiveness of the selected preset. The feedback therefore determines the update of the selection probabilities. In summary, better performance can give a higher probability to a preset, so that it is more likely to be selected in the future. By this means, DPTP selects different parameter values from the provided presets at different stages based on the feedback when the metaheuristic is running, instead of using fixed parameter values. Currently, many of the parameter-control methods use the similar process of collecting feedback, and the feedback determines the effectiveness assessment of the parameter values.

Research approach

In order to use a metaheuristic in an optimization problem with one or more parameters, users can generate parameter values based on their previous experience or making conjectures, and then the fixed set of values is used throughout the entire run. Thus, a bad choice of values affects the entire optimization process and gives the metaheuristic a relatively poor performance. DPTP has a better impact on the performance of metaheuristics than the average performance of using a fixed set of values, where "better than average" means that DPTP performs better than or equal to at least half of the presets it uses. Otherwise, it makes no sense to use DPTP to optimize the parameter values, for we are more likely to get better performance by randomly selecting parameter values rather than by using DPTP.

To validate DPTP, we compare the results of DPTP with those of fixed values. Both of these adjust the parameter values of a metaheuristic, *coral reef optimization* (CRO) in our research. It is a genetic metaheuristic starting with an initial solution and evolving through biologically inspired operators such as mutation, crossover and selection. We use this method because it has 6 parameters, a relatively large number, ensuring that the value assignment and the interactions between different parameters are sufficiently complex. The effectiveness of DPTP can be tested in a convincing way using such a metaheuristic method with so many parameters.

In the validation experiments, we first set up some presets and then run CRO multiple times for each preset, to check which presets are "good" presets that give CRO good performance, and which are "bad" presets otherwise. Next, we run CRO by using DPTP and compare the results with the previous results using fixed presets to check if DPTP's performance is better than at least half of the fixed presets.

For generality, we measure the performance of different experiment settings using seven benchmark functions that have already been widely used to test the metaheuristics. Thus, the complexity and the reliability of these functions are guaranteed. In addition, all of these benchmark functions are for minimization problems.

Experiment results and conclusion

After the experiments and the data analysis, Table.I presents the overview of the comparison between DPTP and the fixed presets measured by the 7 different benchmark functions. The numbers of fixed-

presets that DPTP outperforms in the context of each benchmark function are listed in the table.

BM function	C23	C24	C25	C26	C27	C28	C29
No. of winning	8	4	1	5	8	8	10

Table I: The number of presets that DPTP is better than or equal to, as measured by different benchmark functions

In the experiments, we provide 10 presets to DPTP. From the results, we can see that for 5 out of 7 benchmark functions, DPTP outperforms at least half of the fixed presets. Thus, the research indicates that DPTP has an above-average performance compared to the fixed presets for most of the benchmark functions. It implies that we can use DPTP on metaheuristics as an option for adjusting the parameter values.

Since DPTP gains above-average performance in most of the cases for the basis algorithm in this paper, it is validated to be a promising direction for parameter control of metaheuristic algorithms. However, it still has some limitations. DELMIA continues the research on DPTP after this paper is completed, focusing on why DPTP fails in some cases, and how to improve it to make it work better in these cases. Additionally, they will build the algorithm into their QUINTIQ platform to apply it to more types of real optimization problems in their business, so that the generality of DPTP can be further validated.

Acknowledgements

The completion of this paper marks an end to my learning process in the master project of Industrial Engineering and Management in the University of Twente. In the two-year study, not only have I learned a lot of knowledge and improved myself, but also found myself new possibilities for my life. I will never forget how I felt when I decided to quit my previous job and started study in the Netherlands, and I think that was really the smartest decision I've ever made. It was not that easy. Looking back on the last two years, I've experienced the unprecedented pandemic, loneliness from living on myself, and some anxiety about my age and my future. But finally I overcame all of them, and that is really something I should be proud of.

I had a really unforgettable time in the internship provided by Dassault Systèmes company. I would like to thank my supervisor Tomasz Górski in the company for giving me the opportunity to start the research topic, and his help in the research. Especially, there was a time when I was in depression, and he showed great patience to help me go through that time, and I am so grateful for the warmth from him.

I would also like to thank my first supervisor Wouter van Heeswijk, and my second supervisor Engin Topan for the generous help from them. Wouter always showed me inspiring directions when I felt I was lost sometimes, and gave me elaborate feedback in all the previous versions of the paper. Engin also gave me some practical feedback in the final stage and helped me improve more in the writing work.

Furthermore, I would like to express great gratitude to my parents. Without their support, I would not be able to finish the study of the master project. It was not easy for them to accept that I chose to study abroad after working for 5 yeas, but they provided me with their greatest understanding and support.

Kaiyu Wei,
October 20, 2022

Contents

1 Introduction

1.1 Company introduction

In 1981, a subsidiary of Dassault Aviation 3DS was formed to develop and sell its surface design software, CATIA. 3DS was the precursor of Dassault Systèmes, which later became known not only for the software for aerospace industry, but also for a wider range of more complex industrial software, e.g., urban planning, energy transition, and healthcare, etc. (*What we are*, 2021). In addition to its 3D design software such as CATIA and SolidWorks, the company has launched more products for different areas: ENOVIA for product data management and collaboration, DELMIA for manufacturing and global operations, and Simulia for real world simulation (*Dassault Systèmes - Wikipedia*, 2016). Some products and service that 3DS provides are presented in the Figure.0.

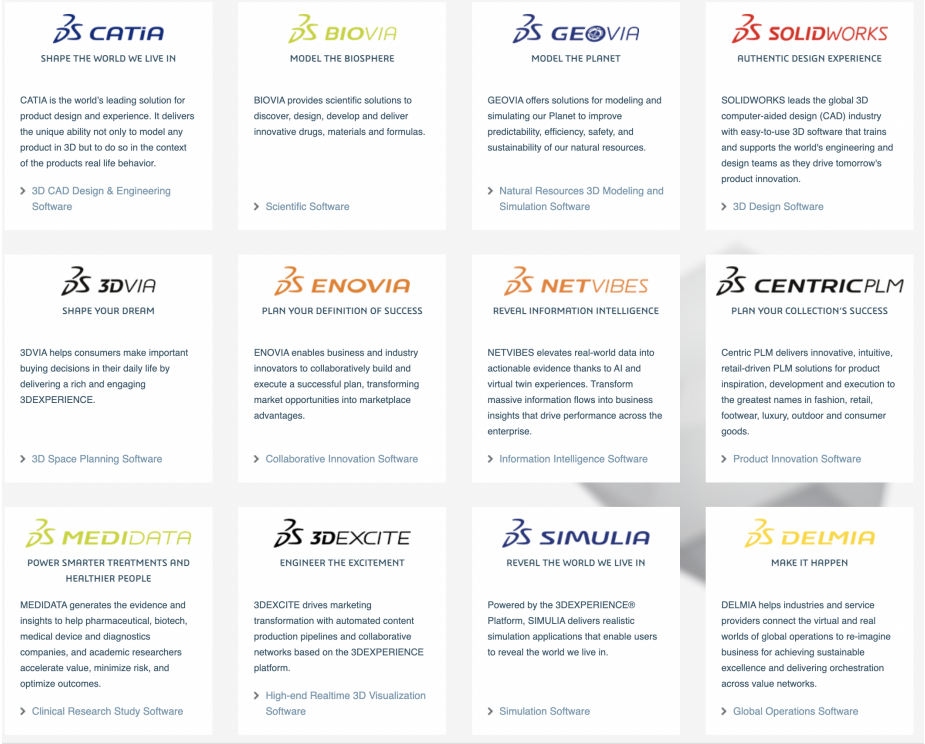


Figure 0: Some products and services 3DS provides
Source: Dassault systèmes website,
`https://www.3ds.com/products-services/`

As a subsidiary of Dassault systèmes, DELMIA, which the research topic of this paper is from, provides customers with solutions for modeling and optimizing their business activities. Its solutions are widely used in logistics, manufacturing, and inventory management processes. As its website notes, *"the solutions are used to plan and optimize complex production value networks, optimize intricate logistics operations, and plan and schedule large, geographically diverse workforces. Key capabilities include predictive and prescriptive data analytics, forecasting, what-if scenario planning, collaborative decision-making, disruption handling and production scheduling"* (*Delmia Quintiq – Dassault systèmes*, 2022).

1.2 Research background

For addressing modeling, simulation, and optimization of real-world problems powered by its own platform named DELMIA Quintiq, many built-in heuristics and metaheuristics are used. By using the metaheuristics, we often need to set up the parameter values, such as the initial temperature and the cooling factor of *simulated annealing* (SA), the number of neighborhoods in many methods, or the mutation factor of genetic algorithms, etc. The problems in the real world are so complex that even trivial changes in the parameters of these solutions can make a large difference. At present, most parameters of metaheuristic algorithms are set manually. In such a condition, the setting of parameter values needs to be carefully considered for ensuring that the optimization algorithms perform well. This is why a tuning method is needed to select and apply appropriate parameter values for these algorithms, and it is not supposed to take a long time to do that before or during the optimization algorithm run when the computational budget and time are limited. Thus, we come up with an online learning method that selects the provided parameter value combinations to give the running metaheuristic good performance. The method runs simultaneously with the metaheuristic and learn about the attributes of different parameters and their values, which is named *Dynamic parameter value tuning method* (DPTP). It is an adaptive parameter control method that adjust the parameter values according to the running feedback as you can see in Section.3.4.

The rest of the paper is organized as follows. In Section 2, we describe the problem in more details and give a definition of the problem. In Section 3, we review some previous work and papers that could help us understand the problem better, and inspire us about how the tuning algorithm should be designed. Section 4 presents the methodology of DPTP. Section 5 introduces how the experiments are conducted and how the result data are organized and analyzed. Section 6 presents the experiment results and analyzes the result data by both visualization and mathematical hypothesis test. Finally, In Section 7, we draw the conclusions from the data and the analysis and have a discussion about the work that can be done in the future to improve DPTP.

2 Research problem

In this section, we introduce the problem we try to solve in this paper, specifying why we need DPTP for solving such problems. In Section.2.1, a general description of how DPTP is supposed to solve parameter-tuning problems is presented, and we make the distinction between the two important terminologies "basis algorithm" and "tuning algorithm" used in this paper. In Section.2.2, a definition of the parameter-tuning problem and the research questions are given. The purpose of the research in this paper is looking for a method to solve the problem and answering the research questions.

2.1 General description

For optimization problems, the solving methods can be simply classified into two categories: exact methods, and approximate methods. The former always give the global optimal solution if there is one, maximizing or minimizing the objected function, but they can take long time for solving a problem. Typically, we use big O notation to measure the time complexity of an algorithm, formatted as $O(n)$, $O(n \log n)$, $O(2^n) \ldots$, etc. The big O notation means the upper bound for an algorithm's

running time in the size of the input n for the algorithm. If the running time of an algorithm is upper bounded by a polynomial expression, i.e., $T(n) = O(n^k)$, it is said to have a polynomial running time (Wikipedia contributors, 2022b). Differently, the approximate methods give "good enough" solution in limited time, but the solution may not be the global optimality. Exact methods are quite intuitive: build models, create objective functions and constraints, and solve them. However, in the real world, most of the optimization problems are NP-hard problems (Talbi, 2009) that cannot be solved in polynomial time by exact methods. Therefore, heuristic methods are raised to solve such problems, given that the solving time is limited in many situations. They can give good solutions in polynomial time, but the solutions are not guaranteed to be global optimal solutions.

Heuristics and metaheuristics are two types of approximate methods. A heuristic method generally starts from an initial solution, and improves the solution through iterations. The metaheuristics, on the other hand, are less likely to fall into local optimum compared to heuristics. Some parameters are involved in the running of heuristics or metaheuristics, for example, the cooling factor in simulated annealing, the mutation rate in genetic algorithms, or number of neighbors in a large neighborhood search (LNS). These parameters greatly affect the performance of the meta-heuristics.

Usually, the values of these parameters can be set manually. The values can be determined by pilot tests, in which we try different parameter values and observe which values give the algorithm good performance before we formally run the algorithm, and we use the good values in the latter formal runs. Also, we can also set parameter values by leveraging experience of some experts of the problem or the algorithm, since they may know which values are suitable for the algorithm. However, pilot tests can take a long time to get useful information about the impact and extent of parameters on the algorithm performance before we can use it to solve problems, and experts' experience is not always that reliable or accurate. Moreover, it is even more difficult to tune them manually when the number of parameters is large, as they could be interactive.

As mentioned above, the tuning method DPTP in this paper optimizes the parameter values of heuristic or meta-heuristic algorithms. In an optimization process, there are basically two kinds of algorithms involved: one is used for tuning the parameter values, which is itself a meta-heuristic presented in this paper, and its parameter values are tuned to get better performance for a specific optimization problem. For the purpose of distinguishing, we name the former algorithm as "**tuning algorithm**" or "**tuning method**", and the latter one as "**basis algorithm**" or "**basis method**" hereafter.

2.2 Problem definition and research questions

In this section, we give the problem definition and the research questions. Define $\boldsymbol{P}$ as an NP-hard problem, and there is a meta-heuristic method (the basis method) $\mathfrak{M}$ for solving it. Define $\mathcal{S}$ as the set of parameters for $\mathfrak{M}$, and $p_i \in \mathcal{S}$ is a parameter of $\mathfrak{M}$, where $p_i \in \mathbb{R}$ and $i \in \mathbb{Z}^+$. We are looking for a method $\mathfrak{M}'$ (the tuning method) to select appropriate parameter values for $\mathfrak{M}$ from among some sets of provided values, a matrix $\mathscr{C}$:

$$\begin{Bmatrix} \lambda_{11}, & \dots, & \lambda_{1i} \\ \vdots & \ddots & \vdots \\ \lambda_{j1}, & \dots, & \lambda_{ji} \end{Bmatrix},$$

where $j \in \mathbb{Z}^+$ is the number of sets of alternative values provided for $\mathfrak{M}'$. We name an alternative set of parameter values $\mathcal{V} = \{\lambda_1, \lambda_2, \ldots, \lambda_i\}$ as a **preset** hereafter in this paper, where λ_i is the provided value in the preset for parameter p_i. In the above matrix $\mathscr{C}$, λ_{ji} is the value for parameter p_i of $\mathfrak{M}$ in preset j.

Assume that the result of a solution for $\boldsymbol{P}$ is evaluated by minimizing a fitness value f, which is the value of a benchmark function used for measuring the algorithm $\mathfrak{M}$. Among all the available presets, $\mathfrak{M}$ can get the smallest fitness value f^* on average when values from preset $\mathcal{V}^*$ are applied to it in the same unit running time. In other words, $\mathcal{V}^*$ is the best choice among all the provided presets. The goal of the tuning method $\mathfrak{M}'$ is then to select presets and apply them to $\mathfrak{M}$ so that the average fitness value of the resulting solutions is as close to f^* as possible, given that users do not know which preset results in good or bad performance for $\mathfrak{M}$.

Since the research problem is clarified, we accordingly need to address the following research questions in the research:

RQ How to design an algorithm to select parameter values for a metaheuristic when it is running, so that the metaheuristic can gain above-average performance, compared to only applying a fixed preset to the metaheuristic?

In **RQ**, the word "dynamically" means that we do not always use the same parameter values for the whole run. Sometimes the tuning algorithm can change the parameter values, as long as it believes that such a change can improve the performance of the metaheuristic it is working on. Moreover, "above-average" means the method proposed in this paper, DPTP, is expected to gain performance that is not worse than at least half of the presets it is using, compared to the results by fixedly using each of its preset. We want the results by using DPTP to converge as much as possible to these best presets. For example, if we provide 10 presets to DPTP, then we can get the results from using DPTP, and 10 groups of results from using only one fixed preset from each of the provided preset. The results of using DPTP should not be worse than at least 5 groups of fixed-preset results if we assess DPTP as an effective method. Otherwise, it is meaningless to use DPTP to optimize the parameter values, for arbitrary selecting of any preset is expected to be better than using DPTP. Section.4 introduces the processes and steps of using DPTP.

During the experiment, we use 7 widely used benchmark functions to measuring the performance of using either DPTP or fixed parameter. More about the benchmark functions can be found in Section.5.1 The result data (benchmark function values) are exported for analysis in Section.6.

In order to answer the research question, we decompose it into a list of sub-questions:

1. Since we tend to let the tuning algorithm, DPTP, to determine the parameter values without any pilot test or pre-knowledge about the metaheuristic, based on what criteria should it make decisions about the parameter values?
2. Once we determine the criteria and method based on which DPTP makes decisions about selecting presets, how to collect the data about the performance of DPTP as well as fixed values, and how should we use these data to improve DPTP's selection making?
3. How to design and execute experiments and data analysis to validate that the metaheuristic using the tuning method gains above-average performance, compared to only using fixed

parameter values during the running?

4. How does DPTP affect the performance of the metaheuristics compared to using only fixed presets, according to the experiment data?
5. If DPTP has a satisfactory above-average effectiveness, what is its behavior like with regarding to the selection of presets?

2.3 Section summary

In this section, we give a general description about the problem we are trying to solve for building an intuitive concept of the problem. After that we give a definition of the problem and the research question. The general description helps us understand what the problem is and why it needs to be solved. The research question is the pivot of this paper, as all the research and the designing are organized based on it. For a better understanding about the research question, it is decomposed into some sub-questions. The main research question can be answered by separately solving the sub-questions. This section gives an insight of the guide for the research, for all the research topics in this paper are based on the research question.

3 Literature review

To solve the problem and answer the **RQ** and its sub-questions, first, we need some knowledge about the optimization problems and methods that are widely used to solve such problems to see if there is any existing method, or if we can get any inspiration from the previous work. Since the problem we work on is an optimization problem, so first Section.3.1 explains the conception of optimization problems, and why approximate methods are acceptable for solving such problems. Section.3.2, 3.3 review some literatures about two types of frequently used approximate methods, heuristics and metaheuristics respectively. We can get some basic conceptions about these methods, and understand how different parameter values influence the performance of them. What's more, we need to design a method of adjusting parameter values, so Section.3.4 reviews literatures about parameter tuning and parameter control methods, which are two types of parameter adjustment method. Since we do not use any pilot test before the basis algorithm is formally run, and we adjust the parameter values during the running time, we are sure that the method we use is a parameter control method, for which the definition can be found in the last subsection. Therefore, we care more about the knowledge of parameter control methods, and this section also gives a summary of the steps of parameter control methods, that can be used in our method design. Section.3.5 reviews some literatures about the online and offline learning methods, and introduces how these methods can help us solve problems. We adjust the parameter values while the algorithm is running, and we use the feedback to improve the decision-making, so the method we design is an online learning method according to literatures reviewed in this section, and we can also learn how to design and use it in this section.

3.1 Optimization problem

Optimization problems are from a wide range of fields involving decision-making, whether in science, engineering or economics. Optimization can be evaluated by objective functions or performance indices (Chong & Zak, 2004). Generally, the globally optimal solution or alternative is expected by users for an optimization process, and some exact methods e.g. linear programming can be used to gain the exact optimal solution. However, most optimization problems in the real world are NP-hard problems that take non-polynomial time to solve (Talbi, 2009), so in practice, most optimization problems can accept good but not guaranteed to be the optimal solutions. In such cases, the approximate methods such as heuristics and metaheuristics can be used to obtain "good enough" solutions for many optimization problems.

3.2 Heuristics

Different from the exact methods, heuristic methods are able to obtain a good solution (not necessarily optimal) for a given optimization problem while the time for obtaining such a solution is acceptable compared to the exact methods (Reeves, 1993). The basic steps of a heuristic method include identifying certain characteristics of a problem, creating an initial solution, and searching locally or globally through specific methods (Salcedo-Sanz, 2016).

In general, heuristics can be divided into two categories: the constructive algorithm and the local search algorithms (Michiels et al., 2007). As the name implies, constructive algorithms create a solution from scratch by taking some designed steps, and in each of these steps the solution is extended from the previous step (Ahuja et al., 2002). Such extensions from the previous step can be performed by adding basic components of the solution, the values of the solution, or in the reverse order, reducing components, etc.

One of the simplest constructive algorithms is the greedy algorithm. A greedy algorithm is an optimization method in which the most promising alternative solution is chosen in each step. Choosing only the most promising alternative in each step leads to a local optimality in most of the cases (Michalewicz & Fogel, 2013). Take the travel salesman problem (TSP) as an example: given a set of cities, and the travel cost (distance) between each two cities, a salesman needs to travel each city precisely once, and go back to the starting city finally at a total cost as low as possible. Intuitively, there is a "brute solution" of the TSP exact solution that enumerates all the possible routing permutations, with a complexity of $O(n!)$ (Abdulkarim & Alshammari, 2015) given the input size n, which is even worse than polynomial time complexity. However, according to the greedy algorithm, starting from the starting city, the unvisited city to which has the lowest cost from the current city is always chosen to be the next city to travel. This greedy algorithm has a complexity of $O(n^2)$. It is dramatically reduced compared with the one of the exact method, with the cost of losing the guarantee of global optimality.

Greedy algorithms are often used as a method to build an initial solution for an optimization problem, but they can be very far from the global optimality. In order to get closer to optimality, the local search algorithms are adopted to improve it. In a local search, a "neighborhood" for the current solution is expected to be specified. The neighborhood here is a set of solutions that can be "reached" by the current solution. In other words, it is "close" to the current solution (Johnson

et al., 1988). Also, for the example of TSP, a frequently used neighborhood is a solution that has only two edges different from the current solution. With such neighborhoods, the current solution is repeatedly replaced by its neighborhood solutions, if the neighborhood solutions lead to a better objective function until some point, e.g. long enough running time, or low enough cost, etc., is reached. This method of looking for neighborhoods in a local search is called "two-opt" swap.

Some popular heuristic methods include local search, divide and conquer, branch and bound, dynamic programming, and cut and plane, etc. (Desale et al., 2015). Aarts et al. (2003) presents a lot of applications of heuristic methods in optimization problems such as traveling salesman problem, vehicle routing problem, and machine scheduling problem, etc. Heuristics are widely used in solving many classical optimization problems.

3.3 Metaheuristics

Although heuristic methods can effectively solve NP-hard problems and give good solutions, it is easy to fall into some local optimality. One of the advantages of meta-heuristic algorithms is that they are more flexible than heuristics and can jump out of local optimality since they are able to cover a larger range of alternative solutions during the same running time compared to other methods (Hansen et al., 2010). It is used in numerous areas such as engineering, machine learning and data mining, systematic modeling, and planning problems etc. (Talbi, 2009). So far, there is no commonly accepted definition for metaheuristics. According to Osman & Laporte (1996), a metaheuristic is an "iterative generation process" that allows its subclass heuristics to exploit the solution space through a number of learning processes in order to find more promising directions in the search for a good solution. Stützle (1999) defines metaheuristic as an advanced strategy that guides the underlying heuristics to improve its performance. During the execution of metaheuristics, a worse movement can be allowed, or a new starting solution can be generated in order to escape from a local optima. Furthermore, metaheuristics do not use randomness blindly compared to traditional heuristics.

Metaheuristic algorithms include ant colony optimization, evolutionary algorithm, genetic algorithm, scatter search, simulated annealing, tabu search, guided local search, hill climbing, iterated local search, stochastic algorithm, etc.(Desale et al., 2015; Pirlot, 1996). All of these methods improve the heuristic methods by either reducing the computational time or improving the solution quality (making the final solution closer to local or global optimality in the same running time).

3.4 Parameter tuning v.s. parameter control

Most metaheuristics are inspired by natural laws such as physics and biology and usually have parameters that users can set. Parameter values have a very strong effect on metaheuristics because they are in charge of the process of heuristics (Huang et al., 2019). Different parameter values bring different behavior to the metaheuristic algorithm, such as the speed of convergence to local optimality and the probability of accepting inferior solutions. Therefore, it is very important to set the parameters of the metaheuristic algorithm carefully in order to improve the performance of the algorithm. Parameter setting problem of metaheuristic algorithm can be divided into parameter tuning and parameter control (Huang et al., 2019; Skakov & Malysh, 2018).

The parameter tuning is also called offline tuning. In this case, the parameter values are set before running the algorithm and remain the same for the duration of the run. The tuning result outputs the best set of parameter values as a result. The main drawback of parameter tuning is that it is very time-consuming. To find the best value, many pilot tests are needed to fine-tune the parameters before running the metaheuristic algorithm (Skakov & Malysh, 2018). Then, after the running of the metaheuristic, if the results are not good enough, the tuning parameters are run until a satisfactory solution is found.

Parameter controls are also called online tuning, in which parameter values can be changed during metaheuristic processes. It requires appropriate initial values and good control strategies during the run. So in such a case, the tuning algorithm is executed in the meantime with the basis algorithm, getting feedback from the performance of the basis algorithm and taking corresponding control strategies (Aleti & Moser, 2016; Hoos, 2011).

Since parameter tuning methods lack the flexibility and ability to update parameter values and are time-consuming (Sun & Lu, 2019), as noted above in the process of metaheuristics, the focus of this paper is parameter control methods. The research for parameter control methods mainly focuses on evolutionary algorithms, which are a major metaheuristic class (Maier et al., 2019), and the basis algorithm used in this paper is also a genetic algorithm. Parameter control methods can be divided into three categories: deterministic parameter control, adaptive parameter control, and self-adaptive parameter control (Aleti & Moser, 2016; Hinterding et al., 1997):

- **Deterministic parameter control** Deterministic parameter control methods update parameter values through some deterministic rules, such as a fixed schedule or a fixed formula with the number of iterations as variables (Sun & Lu, 2019). The feedback from the execution process of the parameter-control algorithm does not matter to the deterministic control process.
- **Adaptive parameter control**

  Adaptive parameter control uses the feedback of the adaptive algorithm to determine the direction or magnitude of the parameter updating strategy or to determine the optimization direction by changing the coefficient of the target function (Hinterding et al., 1997). In the adaptive parameter control method, feedback collection, effect assessment, quality attribution, and parameter update strategies should be considered and applied in these four different steps (Aleti & Moser, 2016):

  - **Feedback collection**

    Feedback collection is a distinguishing feature of adaptive control methods. The feedback from running is collected and then affects the parameter value correspondingly. However, for deterministic parameter control methods, the feedback is not considered; for self-adaptive parameter control methods, the feedback is implicit and does not need to be collected.

  - **Effect assessment**

    With the output of optimization with different values of parameters, we can assess the effect of different values on the performance of an optimizing algorithm.

- **Quality attribution**

  Quality indicators are defined for all alternative parameter values to determine which values are the most successful ones.

- **Parameter update**

  Parameter updates should strike a balance between using high-quality parameters and exploring new values in order to escape the trap of local optimization.

- **Self-adaptive parameter control**

  Self-adaptive parameter control is a combination of finding optimal parameters and finding optimal solutions. In this method, parameter values are usually encoded with the solution genotype, that is, they evolve in parallel with the solution (Aleti & Moser, 2016). Adaptive parameter control method is especially suitable for evolutionary problems with continuous parameter values. When a genetic algorithm is applied for discrete optimization, its performance is not as good as the adaptive parameter control method (Thierens, 2005).

As for specific algorithms for parameter tuning, there has already been some work done about parameter control and tuning. Bartz-Beielstein (2010) and Bartz-Beielstein et al. (2010) presented an R package for automatically tuning of parameters of optimization algorithms using SPOT algorithm. Arcuri & Fraser (2011) demonstrated search-based software engineering (SBSE) technique can be effective for parameter tuning problems of genetic algorithms.

DPTP, the method we design in this paper, turns out to be a parameter control method, since it adjust parameter values while the basis algorithm is running. Hereafter in this paper, both the words "parameter control" and "parameter tuning" mean the "parameter control" in this section.

3.5 Online & offline optimization

Optimization problems, like the parametric problems that we are trying to solve in this article, are usually solved online or offline, or by combining the two methods. In the above Section.3.4, we learned about parameter tuning (offline) and parameter control (online). Offline approaches assume that all information about the optimization problem is known and provide a solution before any uncertainty about the optimization problem is identified. Therefore, offline optimization usually requires a lot of computation cost (De Filippo et al., 2021). As for online approaches, information or data about the problem is often not available a priori (Van Hentenryck et al., 2010). Since online approaches can take feedback from periodic results of the optimization problem and adjust the solution according to them, uncertainty in the optimization can be handled dynamically. However, online methods tend to have strict time constraints, as the results of optimization problems often need to be obtained in a short period of time when implementing online methods. For example, when it comes to work shift scheduling, workers can be arranged to some time slots days or even weeks in advance by offline planning, and when the schedule is needed, it is already there. Therefore, offline approaches usually have plenty of time to plan everything. However, the uncertainty can arise once any worker on the schedule takes unexpected sick leave, making it challenging for online approaches to create a feasible schedule in a short time.

In many real-world problems, online and offline approaches actually combine to solve problems (De Filippo et al., 2021). In many cases, information about uncertainty in a problem is available before it is revealed. As in the previous example, we can use the ienssegeenformation we have offline to plan a shift schedule plan and dynamically replan if any uncertainties are exposed. Combining online and offline can improve each other's performance. In addition, Bemporad et al. (2002) and Pannocchia et al. (2007) demonstrate that offline methods can be used to determine the status and control laws of online optimization. Ravey et al. (2011) uses offline optimization to develop control rules for online optimization and then uses online optimization for real-time energy management. In summary, the use of online or offline methods can be quite flexible, and the boundary between them can be very vague. The combination use is considered to be a good choice in many cases.

3.6 Section summary

From the literature review in this section, an introduction to optimization problems is made, as well as the method about how to solve them with heuristic and metaheuristic algorithms. We also learn how parameter values affect the performance of metaheuristic algorithms. In addition, the literature review in this section provides some ideas about how to adjust the parameter values before or while a metaheuristic is running. This section gives us an insight about the guideline of designing the method: since we do not want any pilot tests before the running, the algorithm we design in this paper should be an online parameter control method; moreover, the literature review in this section also gives the steps of online parameter control, based on which we can design our own online-learning method.

4 Methodology

This section introduces the *dynamic parameter tuning method with presets* (DPTP) from aspects of preset setting, tuning method setting and validation setting. From a review of the previous work in the Section.3, we have not found any existing methods that can solve our problem properly. However, it gives us some good inspirations and insights about how to use online-learning ideas to get feedback and improve the process. Considering that we do not want any pilot tests before the formal run of the basis algorithm, we need to adjust the parameter values during the run, which is obviously an online-learning method. It takes feedback from the running process of the basis algorithm, and adjust parameter values of the basis algorithm accordingly while the basis algorithm is running. On the other hand, DPTP also has some features of the offline method. Some optional sets of values are needed before the basis algorithm is run. By using DPTP, users do not have to worry about how to set parameter values, for the algorithm does it work in a more sensible way based on the feedback of different parameter values.

This section is organized as follows: Section.4.1 introduces an important conception for DPTP in this paper, the preset. It illustrates the data structure of a preset, and how it is generated and selected when it is used in DPTP. Section.4.2 specifies how the probabilities of different presets are updated according to the feedback from the running process. This is the most important part for DPTP method. To make it clear, the pseudocode of DPTP and its relative algorithms are given in this section. For consistency, the same notations from the Section.2.2 are used hereafter.

4.1 Preset setting

As mentioned above, DPTP is an online method with some offline settings before running. We need to provide some so-called presets as alternatives to DPTP so that DPTP can set parameter values from them. This section illustrates the use of presets in DPTP. Section.4.1.1 describes the data structure of presets, i.e. how they are composed. Section.4.1.2 specifies how we determine the number of presets as the input of DPTP, and how it affects the result of DPTP. Next, Section.4.1.3 describes how to generate parameter values for a preset and gives the pseudocode of the algorithm. Finally, Section.4.1.4 introduces how to select a preset by probability.

4.1.1 Preset data structure

In DPTP method, a preset is a set, or combination of parameter values, as described in Section.2.2. It can be seen as an alternative that users provide for DPTP. For tuning a basis algorithm $\mathfrak{M}$ that is used for optimizing problem $\boldsymbol{P}$, the DPTP, which we noted as $\mathfrak{M}'$, helps $\mathfrak{M}$ select appropriate parameter values with a bunch of predefined presets $\mathscr{C}$ as input.

In order to define a preset $\mathcal{V}_j$, where $j \in \mathbb{Z}^+$ is the number of presets we provide to DPTP as the input, values for parameters of $\mathfrak{M}$ are needed first. Users should determine which parameters of $\mathfrak{M}$ they want to tune with DPTP. When a parameter p_i is determined to be tuned, there should be a corresponding value of it in $\mathcal{V}_j$.

Theoretically, all the parameters that are used for the basis algorithm can be added to a preset, because in most cases we do not know in advance how these parameters affect $\mathfrak{M}$'s performance without pilot tests. This is one of the advantages of DPTP that users do not have to know much about parameter attributes, influences on $\mathfrak{M}$'s performance and even interactions among multiple parameters.

When DPTP is running with the basis algorithm, it always selects different presets periodically and assign the values from the selected preset to basis algorithm $\mathfrak{M}$'s parameters. For the matter of selection, each preset should have a probability attribute ψ_j, by which DPTP selector selects the preset applied to $\mathfrak{M}$, so that ψ_j meets:

$$\sum_{n=1}^{j} \psi_n = 1, \quad j \in \mathbb{Z}^+$$

where j is the number of provided presets to DPTP. How presets are selected by DPTP is presented in Section.4.1.4.

Finally, each preset should have a name or index for recording feedback and giving rewards or punishments. All the provided presets constitute an input set $\mathscr{C}$ for DPTP, and the selection of the preset whose value is used will be based on the input set. Figure.0 shows the data structure of a DPTP input.

4.1.2 Determine the number of presets

For using DPTP, we need to provide pre-defined presets. If we give DPTP more presets, then it has more options, and it is more likely that presets that can strongly boost the basis algorithm is included in these presets. However, more presets mean DPTP need longer running time to get

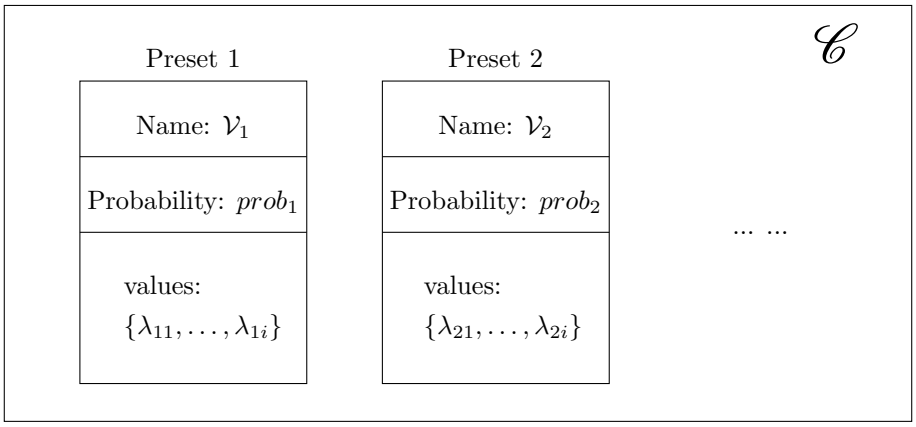


Figure 0: Data structure of the presets input of DPTP

enough feedback. When making decisions about the number of presets, these pros and cons should be carefully considered. Since $\mathfrak{M}$ itself is a metaheuristic, it keeps looking for new neighborhood iteration by iteration. DPTP selects the preset to be applied in each iteration, and updates the probabilities of presets in every r iterations based on how many improvements are obtained per unit of time by using a particular preset, in which we name r the "**cycle length**". Therefore, between any two probability updates, each preset should first have the opportunity to be selected, and then have a long enough run time for DPTP to collect enough data to be able to reliably assess the effect of the preset to $\mathfrak{M}$'s performance while minimizing the randomness as much as possible.

For the above reasons, when determining the number of presets j and the value of the cycle length r, the larger r is, the greater j should be, and vice versa. They should be proportionally associated, i.e. $r \propto j$.

Besides, we should see that more presets provide more options for DPTP. If we had unlimited time and computing capacity, we would try as many presets as we can and set a very long cycle length. However, in the real world, larger j means fewer updates of preset probability of a preset during the same cycle length r. To maintain the balance between the depth and width, we should choose appropriate values of j and r.

4.1.3 Values of parameters in presets

DPTP assumes that users know little or nothing about the effects of parameters on the basis algorithm $\mathfrak{M}$. This is also an advantage of DPTP: you do not have to spend a lot of time studying parameter values or completing time-consuming pilot tests. The generating of parameter values can be quite random: we generate values for parameters by random numbers in some ranges for these parameters, and all we need to provide are these different ranges, and the values in presets can be randomly generated from each parameter's range. This is easier than precisely determining a particular value for each of them.

A good idea for setting the parameter range is that users can always turn to experienced experts with knowledge of the problem for a reliable range. Though we assume that users have little or even no knowledge about the basis method $\mathfrak{M}$ and the parameters' influence on it, some suggestions

from experts who know it well can effectively narrow the scope, and give more accurate values for parameters. Otherwise, the range should be as wide as possible, and wider range means that we should have more presets provided, so larger j as well.

After the range for a parameter is provided, we can simply use a random number generated from the range as the value of the parameter. Algorithm.1 presents the pseudocode of generating j presets and implies that the probability for each preset is initialized by $\frac{1}{j}$.

Algorithm 1: Generate presets

Input:
number of presets j
set $\mathcal{S}$ composed of all parameters p_j to be tuned
map $\mathscr{R}$ of range for all the parameters in $\mathcal{S}$
Output: set $\mathscr{C}$ composed of j presets
$\mathscr{C} \leftarrow \{\}$;
for $\_ \leftarrow 1$ **to** j **do**
    $\mathcal{V} \leftarrow \{\}$; `/* empty preset */`
    foreach $p \in \mathcal{S}$ **do**
        $p.value \leftarrow$ a random value from range $\mathscr{R}[p]$;
        $\mathcal{V}.values[p] \leftarrow p.value$;
    end
    $\mathcal{V}.prob \leftarrow \frac{1}{j}$;
    $\mathcal{V}.name \leftarrow$name defined by the user;
    $\mathscr{C}.append(\mathcal{V})$;
end
return $\mathscr{C}$;

4.1.4 Preset selection

The selection of presets depends on the probability of presets. In other words, presets with larger probabilities are more likely to be selected, and vice versa. DPTP dynamically updates preset probabilities for basis algorithm $\mathfrak{M}$ based on the feedback, or performance from running iterations. Algorithm.2 presets the process of selecting a preset by probability.

4.2 Updating probabilities

DPTP updates preset probabilities dynamically, and the preset probabilities are not fixed at the initialization value. Obviously, we want the more promising presets (i.e., more improvements per unit of time, or less time spent using them to get the same improvements) to have a better chance of being selected and used. We update the probabilities for every r iterations to collect enough data about the improvements obtained by using the provided presets.

In order to update the probability, we need to provide data on improvements and the time or duration of the improvements. Since a preset can be selected multiple times, we collect this data in each preset list or array. All the lists are combined in a map $\mathscr{M}$ as the input for updating the

Algorithm 2: Presets selection

Input: set of presets $\mathscr{C}$

Output: selected preset $\mathcal{V}_{selected}$

$cumProb \leftarrow \{\}$; `/* an empty map for storing cumulative probabilities */`

$tempSum \leftarrow 0.0$;

$rand \leftarrow$ a random value from range (0, 1);

$lastProb \leftarrow 0.0$; `/* the cumulative probability of last preset */`

foreach $\mathcal{V} \in \mathscr{C}$ **do**

  $tempSum \mathrel{+}= \mathcal{V}.prob$;

  $cumProb[\mathcal{V}.name] \leftarrow tempSum$;

  if $lastProb < rand <= cumProb[\mathcal{V}.name]$ **then**

    $\mathcal{V}_{selected} \leftarrow \mathcal{V}$;

    return $\mathcal{V}_{selected}$

  end

  $lastProb \leftarrow tempSum$;

end

error: *NoPresetSelected*

probabilities. The key of $\mathscr{M}$ is the name of a preset $\mathcal{V}$ and the corresponding value is a list of improvements obtained by using $\mathcal{V}$ every time when $\mathcal{V}$ is selected. If preset $\mathcal{V}$ is selected and used for τ times, it is easy to know that the length of the list $\mathscr{M}[\mathcal{V}.name]$ is equal to the τ since we add the feedback data into $\mathscr{M}$ for τ times, i.e., $\tau = length(\mathscr{M}[\mathcal{V}.name])$.

The data structure of map $\mathscr{M}$ is presented in Figure.0. In addition, of $\mathscr{M}$, we also need a map $\mathscr{D}$ for collecting the duration costed for getting these improvements with the same data structure as $\mathscr{M}$, but the value of $\mathscr{D}$ is a list of duration values instead.

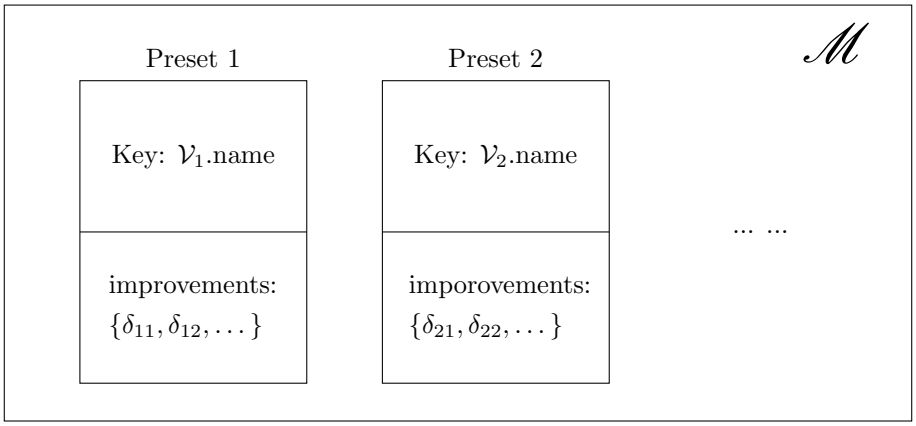


Figure 0: Data structure of $\mathscr{M}$ for updating preset probabilities

The pseudocode for updating the probabilities is given in Algorithm.3. It is executed for each r iterations.

As a result, $\mathfrak{M}$ uses different parameter values in each of its iteration from different presets selected

Algorithm 3: Update preset probabilities

Input:

map $\mathscr{M}$ of improvements

map $\mathscr{D}$ of durations

cycle length r

effect factor β

Output: `void`, the algorithm changes preset probabilities in place

$W \leftarrow 0$; `/* Total weight, for calculating weight of each probability */`

foreach *preset* $\mathcal{V}_j$ **do**

  $\Delta_j \leftarrow 0$; `/* Total improvements` $\mathcal{V}_j$ `gets in last` r `iterations */`

  $T_j \leftarrow 0$; `/* Total time` $\mathcal{V}_j$ `spends in last` r `iterations */`

  for $k := 1$ **to** τ_j **do** `/*` τ_j `is the length of` $\mathscr{M}[\mathcal{V}_j]$ `and` $\mathscr{D}[\lambda_j]$ `*/`

    Δ_j += $\mathscr{M}[\mathcal{V}_j][k]$;

    T_j += $\mathscr{D}[\mathcal{V}_j][k]$;

  end

  W += $\frac{\Delta_j}{T_j}$;

end

if W != 0 **then** `/* If no improvements was made in last` r `iterations, i.e.` $W == 0$`, the updating is skipped */`

  foreach *preset* $\mathcal{V}_j$ **do**

    $\omega_j \leftarrow 0$; `/* Weight of preset` $\mathcal{V}_j$ `*/`

    if $\mathcal{V}_j$ *is used in last* r *iterations* **then**

      $\omega_j \leftarrow \frac{\Delta_j}{T_j}$;

      $prob_j \leftarrow \beta * prob_j + (1 - \beta)\frac{\omega_j}{W}$;

    end

  end

  normalize all $prob_j$ so that $\sum_j prob_j = 1$;

end

by DPTP $\mathfrak{M}'$, according to the probabilities of these presets. For each r iterations in $\mathfrak{M}$, the preset probabilities are updated.

From Algorithm.3, we can also see that for those presets that were not used in the previous r iteration, we are simply giving them a 0 weight in the current update. However, this may not be the situation we want. Imagine a promising preset that was not selected due to randomness in the last r iteration, so it weighs 0. However, if it had been chosen, it could have produced good results. More specifically speaking, we should only give a low weight to a preset just for the poor performance of it, rather than any other factors. Therefore, we want to keep the probability unchanged until next time we get any feedback of using the preset. By this way, we improve the Algorithm.3 by adding some mechanism that protects the unselected presets in the previous r iterations. We can simply keep their probabilities unchanged, and only update the probabilities of the selected and used presets, see Algorithm.4.

Algorithm.4 ensures that the probabilities of unselected presets remain the same, while the summation of all $prob_j$ is still 1. Algorithm.5 gives the pseudocode of DPTP, and the flowchart of DPTP is given in figure.0.

4.3 Section summary

This section illustrates the DPTP method in detail. To use DPTP, we first need to prepare some presets, which are combinations of the parameter values, as the input to DPTP. These presets can be generated by random values within the ranges of the parameters. If the range of a parameter is not known, we can just set a range for it by experience and then generate values from the range. Though this is still like making conjesture, but making a conjesture about a range is easier than making one about an exact value. After the presets are generated and provided to DPTP, DPTP selects a preset based on the selection probabilities, and applies the values from the selected preset to the basis algorithm until the next time when another preset is selected. Through such a selection-and-application process, the basis algorithm uses dynamically selected parameter values by DPTP rather than fixed values. The probability of a preset is determined by the improvements the basis algorithm gains in the unit time when the preset is used by the basis algorithm. In other words, the presets that help the basis algorithm achieve more improvements per unit of time get higher probabilities allocated by DPTP, so that they have higher probabilities to be selected.

5 Experiments and evaluation

In this section, we specify the system for evaluating the effectiveness of DPTP, and how we design experiments for the evaluation. Before we start designing the experiments, the first thing we need is the criteria to measure the optimization results, with which we can compare the effectiveness of different parameter settings (using DPTP or using fixed values). Section.5.1 introduces some benchmark functions for measuring the optimization results, which are used in CEC, an authoritative evolutionary-computation conference held every year. These functions are used for evaluating the optimization algorithms submitted by experts who take part in the conference. Thus, we can trust the reliability of these functions for evaluating DPTP as well. Next, we also need a metaheuristic algorithm on which DPTP can work, so that we can see how the effectiveness of DPTP is when it

Algorithm 4: Update preset probabilities with protection for unselected

Input:

cycle length r

effect factor β

map $\mathscr{M}$ of improvements

map $\mathscr{D}$ of durations

set $\mathscr{U}$ of unselected presets in last r iterations

set $\mathscr{S}$ of selected presets in last r iterations

Output: `void`, the algorithm changes preset probabilities in place

$W \leftarrow 0$; `/* Total weight, for calculating weight of each probability */`

foreach *preset* $\mathcal{V}_j$ **do**

  $\Delta_j \leftarrow 0$; `/* Total improvements` $\mathcal{V}_j$ `gets in last` r `iterations */`

  $T_j \leftarrow 0$; `/* Total time` $\mathcal{V}_j$ `spends in last` r `iterations */`

  for $k := 1$ **to** τ_j **do** `/*` τ_j `is the length of` $\mathscr{M}[\mathcal{V}_j]$ `and` $\mathscr{D}[\lambda_j]$ `*/`

    Δ_j += $\mathscr{M}[\mathcal{V}_j][k]$;

    T_j += $\mathscr{D}[\mathcal{V}_j][k]$;

  end

  W += $\frac{\Delta_j}{T_j}$;

end

if W != 0 **then** `/* If no improvements was made in last` r `iterations, i.e.` $W == 0$`, the updating is skipped */`

  $R \leftarrow 1 - \sum_j prob_j,\ j \in \mathscr{U}$;

  foreach *preset* $\mathcal{V}_j,\ j \in \mathscr{S}$ **do**

    $\omega_j \leftarrow 0$; `/* Weight of preset` $\mathcal{V}_j$ `*/`

    if $\mathcal{V}_j$ *is used in last* r *iterations* **then**

      $\omega_j \leftarrow \frac{\Delta_j}{T_j}$;

      $prob_j \leftarrow \beta * prob_j + (1 - \beta)\frac{\omega_j}{W}$;

    end

  end

  normalize all $prob_j,\ j \in \mathscr{S}$ so that $\sum_{j \in \mathscr{S}} prob_j = 1$;

  foreach *preset* $\mathcal{V}_j,\ j \in \mathscr{S}$ **do**

    $prob_j \leftarrow R * prob_j$

  end

end

Algorithm 5: DPTP

Input:

cycle length r

map $\mathscr{M}$ of improvements

Output: `void`, the algorithm changes preset probabilities in place

$\mathscr{C} \leftarrow$ *Algorithm.*1 *Generate presets* ;

assign values from $\mathscr{C}$ to parameters of the basis algorithm $\mathfrak{M}$;

epoch $\leftarrow 0$;

while *stop criteria not met* **do**

  if *epoch* $\%\, r == 0$ ***and*** *epoch* $!= 0$ **then** `/* Skip the first epoch */`

    *Algorithm.*4 *update probabilities* ; `/* The update is executed in place */`

  end

  $\mathcal{V}_{selected} \leftarrow$ *Algorithm.*2 ; `/* select a preset for current epoch */`

  assign values from $\mathcal{V}_{selected}$ to parameters of $\mathfrak{M}$;

  //...basis algorithm runs and collects feedback data...//;

  epoch += 1;

end

works on the algorithm. We name such an algorithm as "basis algorithm". Section.5.2 describes the basis algorithm we use in this paper, the *Coral Reef Optimzaition algorithm* (CRO). It is a genetic algorithm with 6 parameters to be tuned. Section.5.3 specifies the setup and steps of experiments. Finally, 5.4 illustrates how we process and analyze the result data obtained from the experiments.

5.1 Performance measure

For exact algorithms, the only focus is the time efficiency when they are evaluated, since they always give the optimal solution if there is one, and we want to obtain the best solution with a time cost as short as possible. As for heuristics, it's a different story: consider not only time efficiency, but also the quality of the solution. How fast we get the solution and how close it is to optimality are both important for evaluating the heuristic (Rardin & Uzsoy, 2001).

To measure the performance of DPTP, we use the benchmark function from IEEE congress on evolutionary computation (CEC). It is a world-class conference for researchers and experts of evolutionary computation from all around the world. Liang et al. (2013) summarized and explained all the benchmark functions for testing the algorithms submitted to CEC in the year of 2014. All the functions are for minimization problems, i.e. the smaller benchmark function an experiment can obtain during the same running time, the better the experiment result is thought to be. In addition, these benchmark functions also provide us with the emulated situation of solving real problems, since for real problems, we first convert them into all kinds of objective functions. With different benchmark functions as the objective functions of the optimization, the optimization method can be deemed as if it is solving many kinds of real problems. In this way, the generality of the validation is ensured.

According to Liang et al. (2013), the benchmark functions presented by them are classified to three

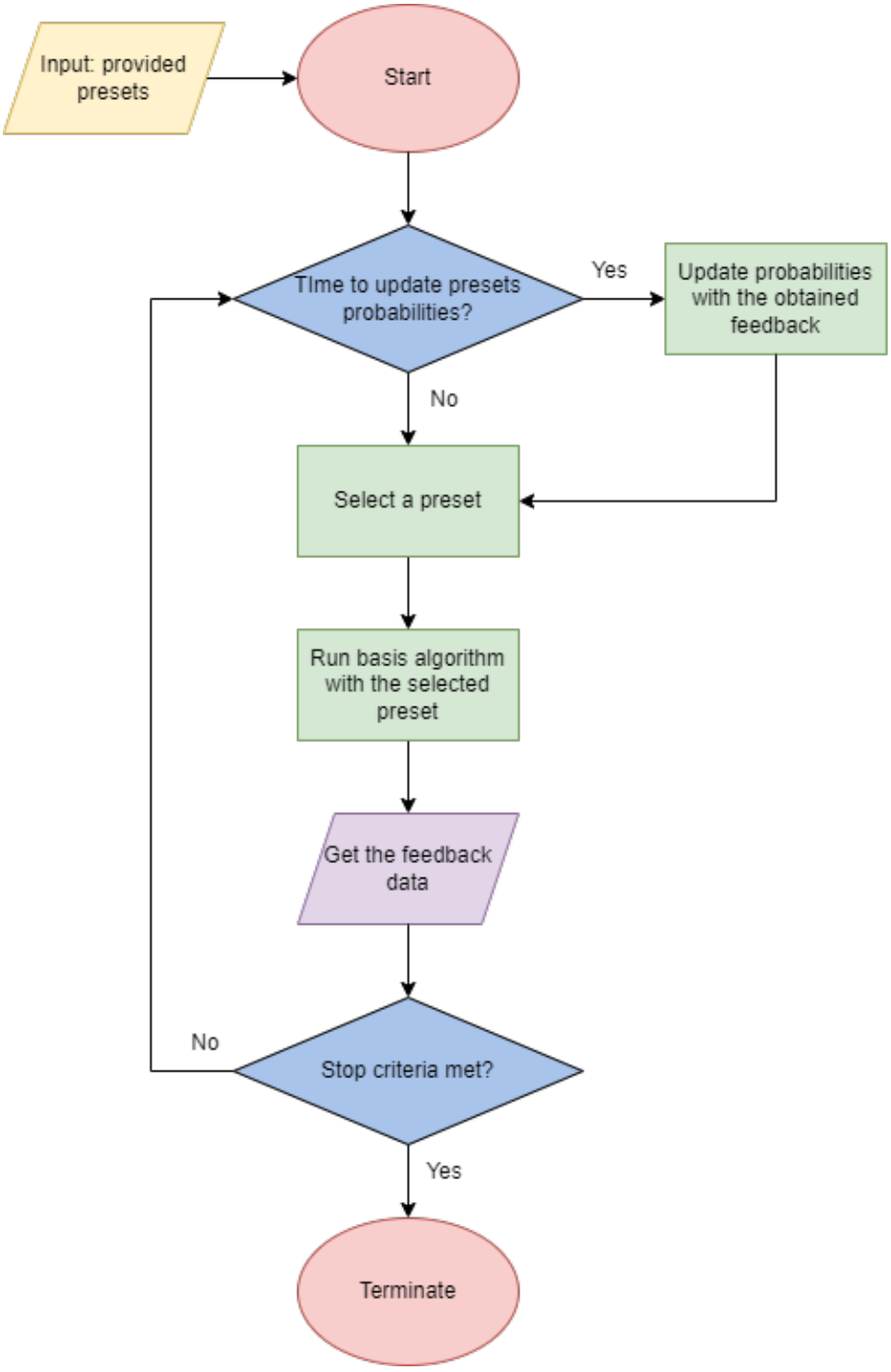


Figure 0: The flowchart of DPTP

types: basic functions, hybrid functions, and composition functions, in which hybrid functions are composed of basic functions:

$$F(\mathbf{x}) = g_1(\mathbf{M}_1\mathbf{z}_1) + g_2(\mathbf{M}_2\mathbf{z}_2) + \cdots + g_N(\mathbf{M}_N\mathbf{z}_N) + F^*(\mathbf{x})$$

, where $F(\mathbf{x})$ is the hybrid function, $g_i(\mathbf{x})$ is the i^{th} basic function used for constructing the hybrid

function, N is the total number of basic functions.

Similarly, composition functions are composed of basic functions or hybrid functions:

$$F(\mathbf{x}) = \sum_{i=1}^{N} \{\omega_i^* \left[\lambda_i g_i(\mathbf{x}) + bias_i\right]\} + F^*$$

, where λ_i is used for controlling the height of each $g_i(\mathbf{x})$, $bias_i$ defines which local optimum is the global optimum (the local optimum with the smallest bias is the global optimum), and ω_i is the normalized weight of $g_i(\mathbf{x})$.

The composition functions are more complicated so that they create challenging test conditions for algorithms, and they also maintain the continuity around the local or global optimum. Thus, in this paper, We prefer using composition functions as the benchmark functions to measure the experiment results. On the other hand, among all the composition functions provided by Liang et al. (2013), functions C23 to C28 have more appropriate function value ranges, according to the results of some pilot test, while the value range of function C30 is too wide (around $10^{-4} - 10^{7}$) for both presenting in a plot and analyzing mathematically. Therefore, we choose benchmark functions C23 to C29 to measure the experiment results, so that the benchmark function values measures the effectiveness of different optimization settings. Figure.0 presents 3D and contour maps of the benchmark functions C23 to C28 (The maps for function C29 is not given in the source paper, but it is still used as a benchmark function in this paper). We can see that many of them have a huge amount of local optimalities that ensure the complexity of the validation.

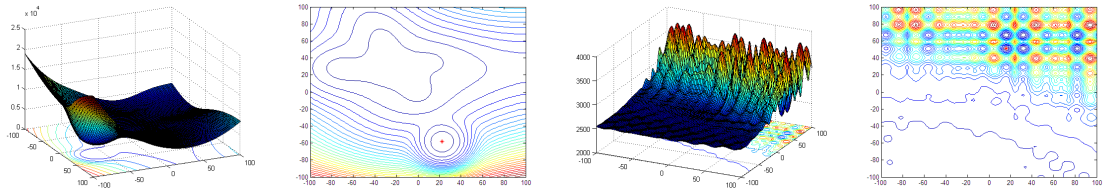
5.2 Basis algorithm

Optimization algorithms with many parameters are more difficult to deal with than algorithms with fewer parameters, because more parameters mean more combinations and more possible interactions among different parameters. Therefore, we prefer to test DPTP with basis algorithm with more parameters to make the results more convincing.

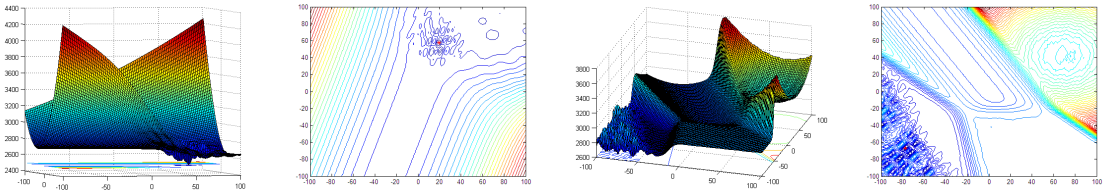
Salcedo-Sanz et al. (2014) introduces a genetic metaheuristic algorithm, the *coral reefs optimization algorithm* (CRO). Like other genetic algorithms, it involves crossover and mutation. It has 6 parameters, so the number of parameters is not so few. Considering the interactions, assuming there is interaction between each 2 parameters, there are at most $\binom{6}{2} = 15$ pairs of interactions among these parameters to be considered. It is sufficiently complicated if we want to set proper parameter values manually. So CRO is a good choice as the basis algorithm for testing DPTP. Since CRO algorithm is not the focus of this paper and we only care about its parameters, no more knowledge about CRO is presented here in this section. More detailed descriptions about CRO can be found in Appendix.A. Table.1 below lists the parameters that need to be adjusted in CRO while it is running. We make presets for these values and then provide the generated presets to DPTP.

The code for all the basis algorithms and the benchmark functions used in this paper is from a GitHub repository *metaheuristics* created by Nguyen et al. (2018, 2019).

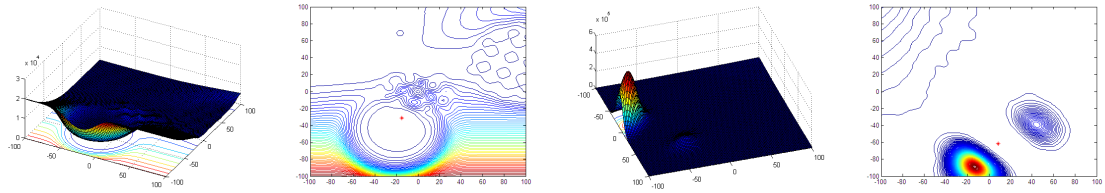
For generality, it is always good that DPTP can be used in more basis algorithms. But because of the time limitation, we only use one basis algorithm in this paper, with 7 benchmark functions.



(a) C23 3D map (b) C23 contour map (c) C24 3D map (d) C24 contour map



(e) C25 3D map (f) C25 contour map (g) C26 3D map (h) C26 contour map



(i) C27 3D map (j) C27 contour map (k) C28 3D map (l) C28 contour map

Figure 0: 3D and contour map of benchmark functions C23 - C28

Source: Problem Definitions and Evaluation Criteria for the CEC 2014 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization, Liang et al. (2013)

Parameter	**Meaning**	**Range**
ρ_0	Initialized rate of free to occupied squares	[0.0, 1.0]
F_b	Rate of broadcast spawner to existing corals	[0.0, 1.0]
F_a	Fraction of corals that duplicate themselves to those who settle in a different part of the reef	[0.0, 1.0]
F_d	Fraction of the worse health corals on the reef that will be deprecated	[0.0, 1.0]
P_d	Probability of depredation	[0.0, 1.0]
k	Number of attempts for a larva to set on the reef	$\mathbb{Z}$

Table 1: Parameters of CRO algorithm

5.3 Experiment setup

This section illustrates how we make preparation before the experiments start, and the experiment steps we should follow. Section.5.3.1 introduces why and how we control the randomness for all the experiment groups. Section.5.3.2 specifies the steps of our experiments. As described by

Section.5.3.2, we first run the experiment by using DPTP, and then use the presets generated by Algorithm.5 for the following experiments groups. Section.5.3.3 presents these presets generated and used in the experiments.

5.3.1 Randomness control

All the experiments use the same random seed, also known as the common random numbers (CRNs). It gives all setting systems the same experimental condition so that a fairer comparison environment (Goldsman et al., 1998). Besides, using the same random seed in the experiments of using DPTP ensures that each time DPTP generates the same presets. Additionally, in order to keep a consistent running experiment environment for all experiments, all the experiments are conducted on a Macbook Air with Apple M1 chip and 16GB RAM.

5.3.2 Experiment steps

For evaluation of DPTP, the result data, which are the benchmark function values, from the basis algorithm obtained by using DPTP is compared to the data obtained by using fixed parameter values in the basis algorithm. For the ease of description, we name the former data the "**DPTP results**", and the latter the "**fixed-parameter results**". From the comparison between them, we can see if one of them is significantly larger or smaller than the other. Since all the benchmark functions are for minimal problems, if the DPTP results are significantly smaller than the fixed-parameter results, we say that the DPTP results is better than the fixed-parameter results, so DPTP performs better than the fixed values used in the basis algorithm, and vice versa. On the other hand, if they are not significantly different from each other, it indicates that DPTP has the same effectiveness with the fixed parameter values used in the basis algorithm. In the experiments, after we get the experiment data, we repeat the above comparison comparing DPTP results with different fixed-parameter results of using preset 1, 2, ..., etc., to see if DPTP can compete with any of these fixed values.

In total, we provide 10 randomly-generated presets for DPTP, so there are 11 setups of experiments: one with DPTP and others with 10 fixed presets, for adjusting the parameter values of the basis algorithm. As for the cycle length r for DPTP, we set it to a value of 30, i.e. preset probabilities are updated for every 30 iterations. Since these 10 presets are also randomly generated, the results obtained from using different fixed presets are compared as well to see which presets are good for the basis algorithm and which are poorer. We want the results of DPTP to be as close to good presets as possible.

Following the above illustration, we perform the experiments of using DPTP first, since it randomly generates presets that are used in the following experiments of using these fixed presets. As for ranges of generating presets, we use the same range from Table.1, except that the parameter k will use a range of $k \in [1, 15]$, and k is integer. Thus, for other parameters, we use the maximum range of values that they can have to make the experimental settings as random as possible. Of course, as mentioned above, if a user can have a narrower range when generating presets, that could be better.

In order to reduce randomness in the experiments, each group of experiment is run 5 times, and each run lasts for 10 minutes. The results of the 5 runs are averaged for subsequent analysis. In

each run, the basis algorithm runs iteration by iteration, just like other metaheuristics do. In each iteration, the algorithm looks for a new neighbor by an operator, and generates some data about the improvements or the performance of the newly used neighbor. As for logging data, for all runs, the fitness values of the benchmark functions are logged in each iteration. In different runs, the number of observations may not be identical, because the number of iterations of a run can be affected by many factors, such as the quality of the initial solution and neighborhoods. We'll give a way of processing the data to deal with the different numbers of observations in different runs, for the ease of analysis in the Section.5.4.1.

In summary, the steps of the experiments are as follows:

1. Determine ranges for parameter values as input for DPTP.
2. Run the basis algorithm using DPTP for 5 times, and 10 minutes for each run. The reason why we set 10 minutes as the running time is that 10 minutes are enough for the experiment results to converge from some pilot runs. This is the first experiment setup we run. While the basis algorithm is running, log the selected presets for each iteration, the fitness values after each iteration and the time iteration takes.
3. Run the following 10 experiments with using 10 fixed presets in the basis algorithm. For each preset, run 5 times and 10 minutes for each run, just like the above setup using DPTP. Similarly, while the basis algorithm is running, log the selected presets for each iteration, the fitness values after each iteration and the time iteration takes.
4. The data we get from the above setups are logged iteration by iteration. As we illustrated above, in different runs, the number of observations may not be identical, and this is not convenient for data analysis. Thus, we should convert results from being logged by iterations to the data logged by seconds (600 seconds in each run), and then make the average of 5 runs for each setup of experiments as the output data to be analyzed.

5.3.3 Presets for experiments

As mentioned above, the presets are generated by the first group of experiments using DPTP, and these presets are used for the next 10 groups of experiments using fixed presets. Table.2 lists the presets generated from the first group of experiments, and these presets are used in the whole process of experiments in this paper.

5.4 Processing of data

This section gives an illustration about how to process and analyze the data from the experiment results. Though we set a run time of 10 minutes following the setting from Section.5.3.2, different groups can have different numbers of observations (i.e., benchmark function values we get at different moments) from experiments. For the convenience of analysis, Section.5.4.1 introduces how to convert data with different number of observations into that with the same number of observations. Next, we want to examine whether data from DPTP groups are significantly different from the data from fixed-preset groups, so that we know whether DPTP is effective for tuning the parameter values. For this purpose, we use Hypothesis test. Section.5.4.2 simply introduces the steps of hypothesis

Preset	ρ_0	F_b	F_a	P_d	k
0	0.6669	0.1029	0.4241	0.4764	8
1	0.1626	0.7618	0.7678	0.3473	6
2	0.6326	0.2232	0.1210	0.9659	10
3	0.1964	0.2798	0.2790	0.5339	1
4	0.2875	0.8130	0.4783	0.6252	15
5	0.0291	0.4424	0.8045	0.5007	8
6	0.7230	0.5721	0.8537	0.4813	1
7	0.0425	0.9829	0.3522	0.2382	14
8	0.7245	0.6713	0.3028	0.6728	13
9	0.9035	0.8572	0.7358	0.3787	1

Table 2: Presets used in experiments

test, and how we set the null hypothesis (H_0) and the alternative hypothesis (H_1) for it. As for the test method used for hypothesis test, since the observations are definitely not normally distributed (they get smaller and smaller as the time goes), non-parametric test fits for such a situation in the hypothesis test. Thus, we introduce two non-parametric test methods in Section.5.4.3: Wilcoxon signed-rank test and Wilcoxon rank-sum test that can be used for analyzing the experiment data, and finally we select Wilcoxon rank-sum test as the test method in this paper test after compare these two methods.

5.4.1 Preprocessing

The number of iterations in different rounds of experiments are unpredictable. They differ from each other for the reason of randomness, or the setting of parameter values. However, in order to conduct analysis of data, we do need data from different group in the same shape, i.e. with same number of observations. Therefore, we have to preprocess the data before we do the analysis. Since each group of experiments are run for 5 x 10 minutes, for data from each round, we have to convert the data logged by iteration to the one by seconds.

Each time when the fitness value is updated, the time point is also logged. We can simply take the fitness value at second $1, 2, \ldots, 600$, if there's a corresponding fitness value right at the time point. But if there's no fitness value logged, which means there was an iteration going on when the experiment was run and no fitness value was logged at the time point, we go one step back and take the fitness value from the nearest last iteration. In such a way, the data from iterations of each group experiment is "stretched" to 600 observations, and it is easier to be analyzed and compared to results from other rounds or experiments.

5.4.2 Hypothesis test

Hypothesis test (HT) is a mathematical technique for testing if the data at hand sufficiently support a predefined hypothesis (Wikipedia contributors, 2022a). First we need to provide a null hypothesis H_0 and an alternative hypothesis H_1. H_1 is the hypothesis we want to prove statistically, and we look for proof of it from the data we have. If there's no sufficient proof from the data, we reject H_1

and accept H_0. In this paper, after we have all the data we need from the experiments, we want to compare the benchmark function values from every experiment group and test if the data from DPTP are significantly different from the data from the bad presets, and whether the results of DPTP are equal to the good presets. As defined in **RQ**, for gaining an above-average performance, DPTP should be at least equal to or better than half of the fixed presets. In such a case, the null hypothesis is defined as: $\bar{F}_{DPTP} \leq \bar{F}$. It means that DPTP is the at least as good as a fixed preset. Meanwhile, we can define the alternative hypothesis as $\bar{F}_{DPTP} > \bar{F}$, which means DPTP is worse than one of the fixed preset. $\bar{F}$ is the average data from the experiments of fixed presets, and the average data are obtained after being preprocessed by the steps depicted in Section.5.4.1. Thus, our hypothesis test is a right-side one-tailed test. When we compare the results from DPTP with those from the good presets, we want to test if their results are the same (or very close to each other), and when we compare the results from DPTP with those from the bad presets, we want to test if DPTP performs better than the bad presets.

For the right-side one-tailed test, we reject H_0 and accept H_1 if $p - value \leq \alpha$, while for the two-sided test, we reject H_0 if $p - value \geq \alpha$, where α is the significance level. Figure.2 presents such a rejection case in right-side one-tailed test. Usually the value of the significance level α is 0.05.

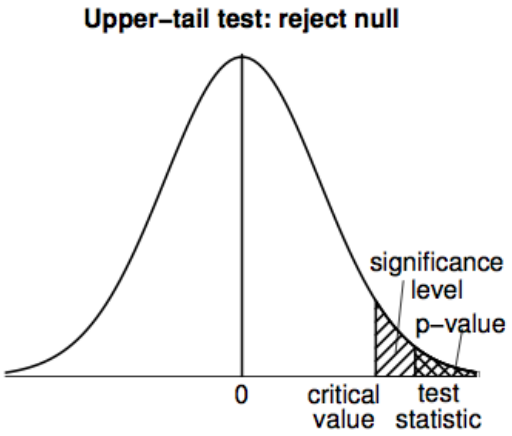


Figure 2: Right-side one-tailed hypothesis test when the
p-value is in the rejection region

The hypothesis test given above lead to another question: how should we determine which presets are good or bad presets? In order to classify the presets into good or bad classes, we can firstly classify by simply visualizing the benchmark function values from DPTP and fixed-preset groups. Then, we roughly classify the fixed presets of which the data are close to or better than DPTP to the good-preset class. Otherwise, the presets are classified to bad-preset class. The visualization of the results and the classification process are presented in Section.6.2. Since this is only a rough classification, we can also use the hypothesis test to test if they are significantly good or bad presets, so we know that how many presets that DPTP is better than or equal to to gain an above-average performance defined by **RQ**.

As for the test method, we should consider both of the attributes of the data distribution and the number of groups we want to compare, as noted by Eftimov et al. (2017). If we assume that the

data follow some specific distribution (usually normal distribution), then we perform a parametric test by using the sample mean and the sample variance. But if we don't assume that the data follow any distributions, we can perform a non-parametric test by using the ranking method as used in the Wilcoxon signed-rank test (Saha et al., 2016; Bui et al., 2018; Van Veldhuizen & Lamont, 2000) for the two-sample test and Friedmen test (Elmazi et al., 2015; Ahmed et al., 2020; Faramarzi et al., 2020; Mousavirad & Ebrahimpour-Komleh, 2017) for the multiple-sample (more than two) test.

Since our data are obviously not normally distributed (the benchmark function values decrease as the run goes), and we want to compare the results of DPTP to the 10 fixed presets, or in other words, we want to perform 10 two-sample tests of DPTP to each of the 10 fixed presets. The data from DPTP and the fixed presets are independent of each other. Therefore, We need to select a non-parametric test method for unpaired data samples.

5.4.3 Non-parametric test

For the hypothesis test, a test method is needed. For observations that follow the normal distribution, or some other specific distributions, t-test is a widely used test method. However, like in our research, the parameter and distribution of the experiment results may not follow a specific distribution, and the parameters of the distribution is unknown to use. In such a case, we can use nonparametric tests. Non-parametric tests are often used when the assumption of parametric tests are violated (Pearce & Derrick, 2019). Wilcoxon signed-rank test and rank-sum tests are two methods of the nonparametric test.

Wilcoxon signed-rank test As Conover (2007) noted, Wilcoxon signed-rank test is "*a non-parametric statistical hypothesis test used either to test the location of a population based on a sample of data, or to compare the locations of two populations using two matched samples*". It was introduced by Wilcoxon (1946). For paired-sample test, our data consists of paired samples:

$$(X_{DPTP1},\ X_{(1,1)}),\ (X_{DPTP2},\ X_{(1,2)}),\ \ldots,\ (X_{DPTPn},\ X_{(m,n)}),$$

where m is the index of preset m, and n is the index of n'th observation. In our case, $m \in \{0, 1, \ldots, 9\}$, and $n \in \{0, 1, \ldots, 599\}$. We can convert the paired two-sample test to one sample test by replacing the n paired observations with $D_n = X_{DPTPn} - X_{(m,n)}$. We rank the absolute value of D_n and compute the sum of the rank of the positive differences t^+ and the sum of negative differences t^-. Then the test statistic is the smaller number between t^+ and t^-:

$$t = min(t^+, t^-).$$

For small size of observations, the distribution of the statistic can be looked up from a table. For larger samples, the distribution of the normalized statistic $Z = \frac{t-\bar{t}}{\sqrt{Var(t)}}$ is approximately a normal distribution, so we can use the distribution of the Z to perform the test. According to Pratt & Gibbons (1981), if there are n samples, the mean of the sum of the sample rank is:

$$\mu = \mathbf{E}(t^+) = \mathbf{E}(t^-) = \frac{n(n+1)}{4},$$

and the variance of the sum of the rank for either positive or negative differences is:

$$Var(t^+) = Var(t^-) = \frac{n(n+1)(2n+1)}{24}.$$

Thus, the rejection region for our right-side one-tailed test is:

$$Z = \begin{cases} \frac{t - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{24}}} > -Z_{1-\alpha/2}, & \text{if } t = t^+, \\ \frac{t - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{24}}} < Z_{1-\alpha/2}, & \text{if } t = t^-, \end{cases}$$

where $Z_{1-\alpha/2}$ is the right-side critical value, and α is the significant level.

Wilcoxon rank-sum test For non-paired samples, there's another non-parametric test method, the Wilcoxon rank-sum test that can be used. Altinbas & Akkaya (2017), Moussa et al. (2018), Abdollahzadeh et al. (2021) and Pholdee & Bureerat (2018) used the method to test the effect of different metaheuristics.

In our research, we test the samples from DPTP and a fixed-preset, noted as X_{DPTP} and $X_{(m,n)}$ respectively, where m is the index of preset m, and n is the index of n'th observation. As described by Pratt & Gibbons (1981), In order to carry out the Wilcoxon rank-sum test, we first combine all the $2n$ observations into one group, but we still keep track of which observations are from which sample. Then we rank the observations by their magnitudes, in the case of our research, the fitness value. In this way, we can assign the rank $1, 2, \ldots, 2n$ to all observations. Next we calculate the sum of assigned ranks of observations from each sample. Here we note R_{DPTP} as the rank sum of observations from the DPTP sample, and R_m as the rank sum of observations from the sample of preset m. Either R_{DPTP} or R_m is called the Wilcoxon rank-sum test statistic. It is easy to know that

$$R_{DPTP} + R_m = 1 + 2 + \cdots + 2n = 2n(2n+1)/2$$

In our case, if the two samples are from the same population, or in other words, they have no significant difference, then we have

$$\mathbf{E}(R_{DPTP}) = \mathbf{E}(R_m)$$

And the statistic of Wilcoxon rank-sum test is

$$R = \sum_{1}^{2n} kI_k$$

where

$$I_k = \begin{cases} 1, & \text{if the observation with rank k is from the DPTP sample} \\ 0, & \text{if the observation with rank k is from the preset m sample.} \end{cases}$$

The mean and variance of the statistic are given by

$$\mathbf{E}(R) = n(2n+1)/2$$

$$var(R) = [n^2/(2n-1)][(4n^2-1)/12] = n^2(2n+1)/12$$

Similar to Wilcoxon signed-rank test, when the sample number n is large, the statistic R approximately follows a normal distribution.

Since our samples are unpaired and independent of each other, we use the Wilcoxon rank-sum test to validate the experiment results with a significance level of $\alpha = 0.05$. As mentioned before, a right-side one-tailed test is needed for the research.

5.5 Section summary

We design experiments in this section for validation of DPTP. First, we need to select the basis algorithm that DPTP works on to adjust its parameter values. Secondly, we also need some benchmark functions to measure the optimization results. We use the coral reefs optimization (CRO) algorithm as the basis algorithm, for it has up to 6 parameters to be set. The number of parameters in CRO is not that small, and there might also exist interactions between different parameters. As for the benchmark functions, we use 7 functions that are widely used to measure the optimization results of metaheuristics. Each of these functions have a huge amount of local optimalities. Therefore, the complexity and the generality of the validation are both ensured.

In the experiments, we compare the results achieved by using DPTP to the results achieved by fixedly using each presets that we provided to DPTP. After all the experiments, we process the result data by using both graphical and mathematical methods to check if DPTP has an above-average performance compared to the fixed presets.

6 Results and Analysis

This section presents the experiment results and the analysis results for the validation of significance. For validation, we use multiple benchmark functions that can be found in Section.5.1. Section.6.1 presents the results from all experiments groups. The initial and final benchmark function values (which are also named as "fitness values" or "fitness"), and the improvements during the experiment can be found in this part. For there is a huge amount of data from the experiments, we only present the average-value summaries for data from each benchmark function in tables. The raw data can be found in Appendix.B. In Section.6.2, the fitness values obtained by time obtained from different benchmark functions are presented graphically, so that we can have a qualitative conclusion from the figures. Then in Section.6.3, the test results are presented and mathematically validates whether the results from groups using DPTP are significantly same with those using good presets.

As mentioned in Section.5.4.2, we use different types of hypothesis tests for different results of DPTP and fixed-preset groups: for the data from fixed-preset groups that are not as good as the DPTP group, we want to testify that DPTP groups perform better than them. Thus, for comparing these fixed-preset group to the DPTP group, we use one-tailed test; for those fixed-preset groups that are close to or better than DPTP groups, we want to testify that these groups are significantly the same as DPTP groups, so here we use two-tailed tests. For recognizing the good and bad fixed-preset groups, in Section.6.2, we present the optimization results graphically, and then we roughly classify the fixed-preset groups into two categories according to the figures. Next in Section.6.3, we use the categorization to compare DPTP groups with the fixed-preset groups by different hypothesis test method.

6.1 Improvement results

Table.3 presents the data of all the experiment groups, and we can see the initial fitness, final fitness and the improvements from all groups using different experiment settings and benchmark functions. As mentioned in Section.5.3.2, all the results are average values of 5 rounds with the identical setup

from their own groups.

Table 3: Improvements by using DPTP and fixed presets of all benchmark functions

Benchmark function	**Experi. setup**	**Initial fitness**	**Final fitness**	**Improvements**	**Improvement percentage**
C23	DPTP	26992.13	2693.673	24298.46	90.02%
	Preset 0	25789.55	2756.774	23032.78	89.31%
	Preset 1	26236.34	2676.916	23559.42	89.80%
	Preset 2	27330.98	2737.67	24593.31	89.98%
	Preset 3	25465.2	2735.844	22729.35	89.26%
	Preset 4	25977.27	2678.884	23298.38	89.69%
	Preset 5	25361.18	2702.239	22658.95	89.34%
	Preset 6	26466.49	2695.999	23770.49	89.81%
	Preset 7	26699.03	2767.44	23931.59	89.63%
	Preset 8	26699.03	2767.44	23931.59	89.63%
	Preset 9	27365.9	2695.156	24670.75	90.15%
C24	DPTP	41321.11	20639.41	20681.7	50.05%
	Preset 0	41543.73	21680.46	19863.27	47.81%
	Preset 1	41117.98	19592.93	21525.04	52.35%
	Preset 2	41557.57	21347.44	20210.13	48.63%
	Preset 3	41312.02	20930.71	20381.31	49.34%
	Preset 4	40669.04	19755.83	20913.2	51.42%
	Preset 5	41373.19	19990.72	21382.47	51.68%
	Preset 6	41055.92	19605.2	21450.72	52.25%
	Preset 7	41092.94	21357.93	19735.02	48.03%
	Preset 8	40650.55	20069.71	20580.84	50.63%
	Preset 9	40413.35	20360.37	20052.98	49.62%
C25	DPTP	7360.97	3843.79	3517.18	47.78%
	Preset 0	7349.66	3765.728	3583.933	48.76%
	Preset 1	7301.543	3807.63	3493.913	47.85%
	Preset 2	7236.729	3782.24	3454.489	47.74%
	Preset 3	7296.342	3794.346	3501.996	48.00%
	Preset 4	7384.173	3817.764	3566.41	48.30%
	Preset 5	7389.313	3766.736	3622.577	49.02%
	Preset 6	7507.889	3783.694	3724.196	49.60%
	Preset 7	7239.824	3967.741	3272.083	45.20%
	Preset 8	7078.604	3789.462	3289.143	46.47%
	Preset 9	7255.945	3799.114	3456.832	47.64%
C26	DPTP	24366.9	13030.68	11336.22	46.52%
	Preset 0	24233.22	14123.02	10110.2	41.72%
	Preset 1	24091.92	12402.09	11689.83	48.52%

Continued on next page

Table 3 – continued from previous page

Benchmark function	Experi. setup	Initial fitness	Final fitness	Improvements	Improvement percentage
	Preset 2	24616.94	14049.56	10567.38	42.93%
	Preset 3	24021.75	13447.69	10574.06	44.02%
	Preset 4	24349.25	12634.16	11715.09	48.11%
	Preset 5	24494.97	12632.68	11862.29	48.43%
	Preset 6	24542.86	12623.4	11919.45	48.57%
	Preset 7	24095.84	13085.9	11009.94	45.69%
	Preset 8	24124.98	12615.13	11509.85	47.71%
	Preset 9	24381.49	13221.99	11159.5	45.77%
C27	DPTP	31438.39	6162.619	25275.78	80.40%
	Preset 0	30640.95	6850.563	23790.39	77.64%
	Preset 1	30993.41	5960.878	25032.53	80.77%
	Preset 2	32075.56	6617.573	25457.98	79.37%
	Preset 3	30950.79	6491.126	24459.66	79.03%
	Preset 4	31417.76	6020.921	25396.84	80.84%
	Preset 5	31178.97	6159.164	25019.81	80.25%
	Preset 6	32115.1	6308.458	25806.64	80.36%
	Preset 7	32114.26	6909.51	25204.75	78.48%
	Preset 8	32165.41	6103.089	26062.33	81.03%
	Preset 9	30801.36	6167.412	24633.94	79.98%
C28	DPTP	71525.71	31925.55	39600.17	55.36%
	Preset 0	71547.2	32907.68	38639.52	54.01%
	Preset 1	71262.71	30594.4	40668.31	57.07%
	Preset 2	71202.56	33488.04	37714.52	52.97%
	Preset 3	70795.6	32601.11	38194.48	53.95%
	Preset 4	72282.05	31888.56	40393.49	55.88%
	Preset 5	70051.37	30923.74	39127.63	55.86%
	Preset 6	71704.23	32607.32	39096.92	54.53%
	Preset 7	71038.72	35871.09	35167.63	49.50%
	Preset 8	72001.26	32081.35	39919.91	55.44%
	Preset 9	71196.85	32868.23	38328.61	53.83%
C29	DPTP	71525.95	31477.07	40048.88	55.99%
	Preset 0	70889.64	33387.96	37501.68	52.90%
	Preset 1	71010.44	32237.18	38773.26	54.60%
	Preset 2	70795.75	33466.33	37329.43	52.73%
	Preset 3	70541.41	32506.64	38034.76	53.92%
	Preset 4	70786.24	31812.70	38973.54	55.06%
	Preset 5	71272.63	31988.74	39283.89	55.12%
	Preset 6	71589.63	32272.45	39317.18	54.92%
	Preset 7	71316.45	36282.94	35033.51	49.12%

Continued on next page

Table 3 – continued from previous page

Benchmark function	Experi. setup	Initial fitness	Final fitness	Improvements	Improvement percentage
	Preset 8	71003.53	31419.18	39584.35	55.75%
	Preset 9	72292.95	33845.07	38447.87	53.18%

From the last column "Improvement percentage" of Table.3, we can see the improvements, comparing the final optimization results with the initial ones, in percentage with different setups (using DPTP tuner or a fixed-preset), measured by all 7 benchmark functions. The improvement percentage values from fixed-preset groups that are lower than the those obtained by using DPTP have been highlighted, so that we can clearly see with the same benchmark function, how many of them are less than the DPTP improvements. Remember that in research question **RQ**, we want DPTP to gain above-average results. For validating this, we need to check if the DPTP result is at least superior to half of the fixed-preset results measured by the same benchmark function. For this purpose, from the benchmark function C23 to C29, we respectively have 9, 5, 4, 5, 5, 6, 10 fixed-preset results that are less than the DPTP result under the same benchmark function. For all the benchmark functions except C25, the DPTP result is superior to at least half of the fixed-preset results. Therefore, according to the data in Table.3, DPTP is up to the standard of above-average level set by the research question **RQ**, considering DPTP results are better than fixed-preset results under most of the benchmark functions.

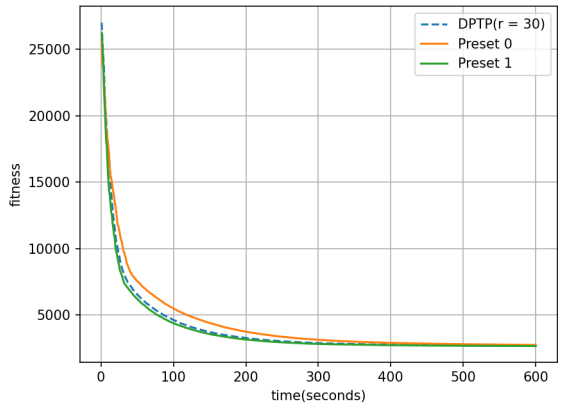
However, we should also notice that for benchmark functions C24, C26, C27, C28, the advantage of DPTP is not that obvious over the fixed-preset groups. The numbers of fixed-preset results that DPTP is superior to in these groups are just over the threshold, half of the fixed presets. In addition, for each benchmark function, the differences of improvements among different setups are not that large. By subtracting the lowest percentage from the highest percentage measured by each same benchmark function, we get ranges of 0.89%, 4.54%, 4.40%, 6.85%, 3.39%, 4.1%, 6.87%. In such a condition, it is harder to tell whether DPTP results are significantly better than fixed-preset results since they are very close to each other. Thus, for drawing more solid conclusions, we need the hypothesis test to significantly validate the relationship among these results.

6.2 Graphical presentation of optimization results

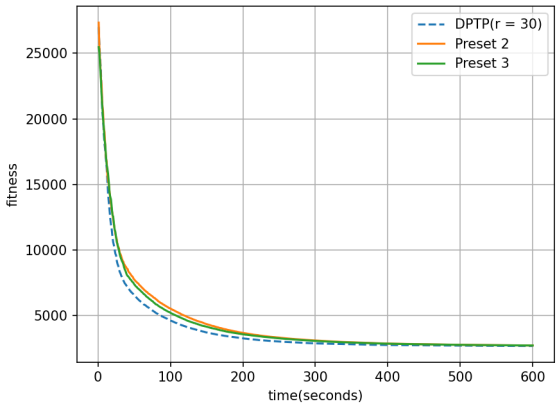
This section graphically presents data from experiments using different benchmark functions C23-C29 in the following subsections. We visualize the results in figures after preprocess the results by following the steps in Section.5.4.1. In these figures, x-axes stand for the running time while y-axes stand for the benchmark function values (also named as fitness values). The figures show the optimization results as the running time goes. Since we have in total 11 setups (using DPTP and 10 fixed presets) for each benchmark function, plotting all the results in the same figure will make it blur and confusing. So for data from each benchmark function group, we use some subplots to compare the results from the DPTP setup and the fixed-preset setup so that they can be observed clearly. After we plot the figures, the presets used in each benchmark function group are roughly

categorized to good- or bad-preset categorizations, according to the plotted figures, so that we can compare the results by using DPTP to those by using fixed presets, and examine the effectiveness of DPTP, as noted by Section.5.3.2. A summary of categorization of good and bad presets for all the benchmark function groups can be found in Section.6.2.8, so that we can have a general conclusion of the good and bad presets measured by all the benchmark functions. Buy such a summary, we know how many presets the DPTP is better than or equal to so that a general impression of the DPTP's effectiveness is formed.

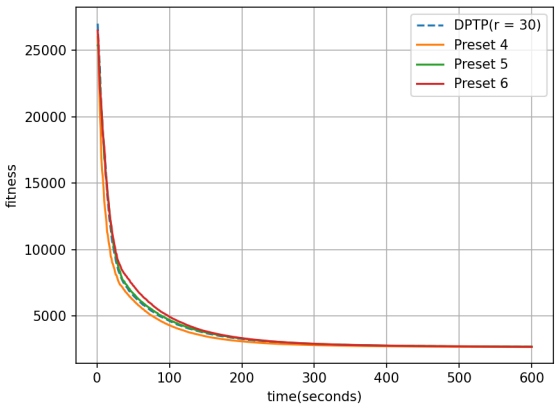
6.2.1 Benchmark function C23



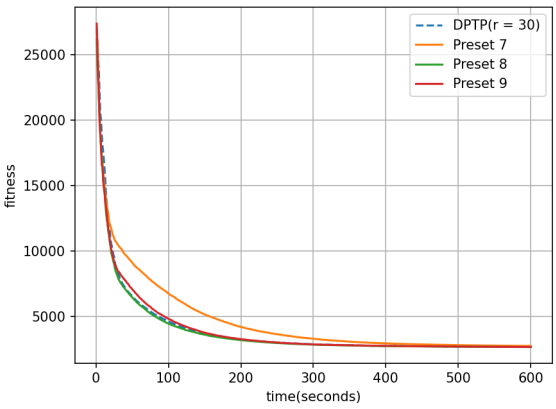
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6

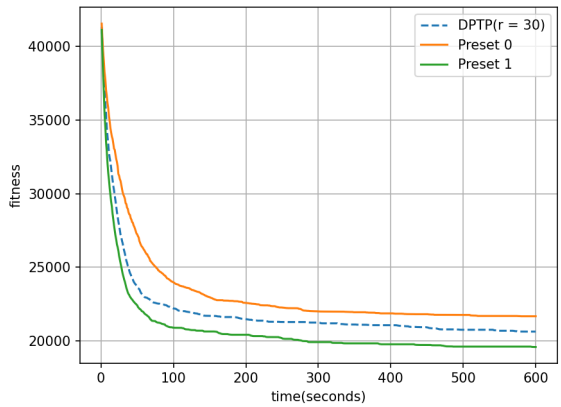


(d) DPTP vs. Preset 7, 8, 9

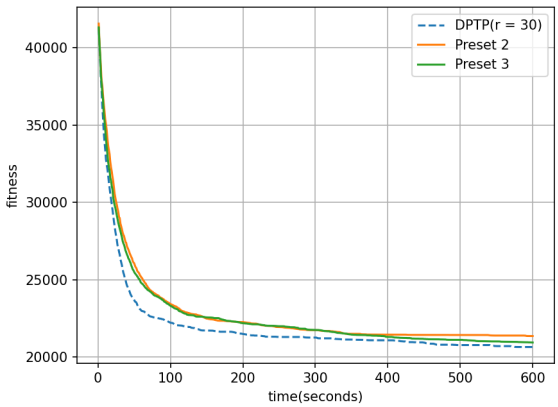
Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C23

Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can roughly see from the figures that DPTP results are close to the results from presets 1, 4, 5, 6, 8 and 9, and better than results from presets 0, 2, 3, 7. According to this we classify the former as good presets, and the latter as bad presets.

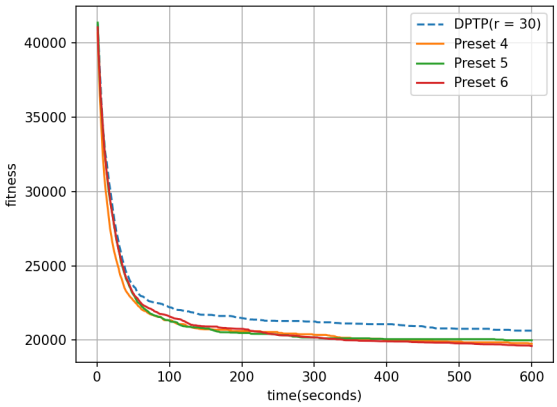
6.2.2 Benchmark function C24



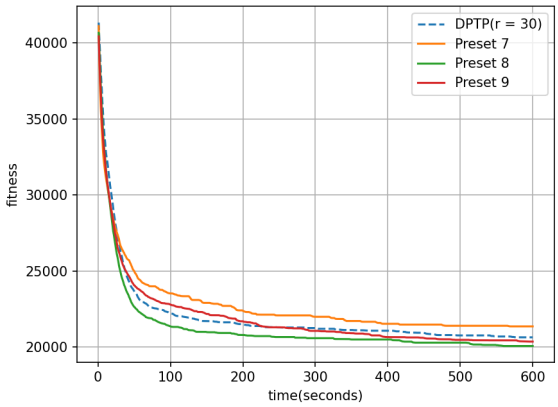
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



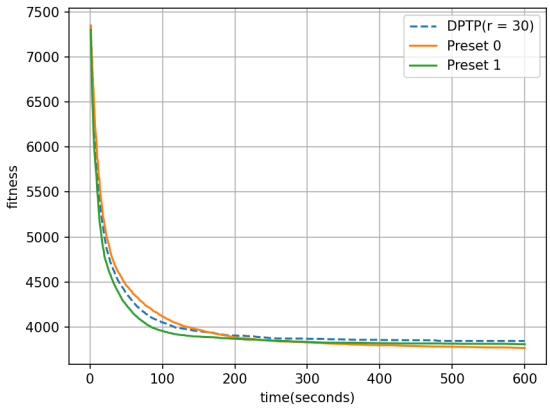
(d) DPTP vs. Preset 7, 8, 9

Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C24

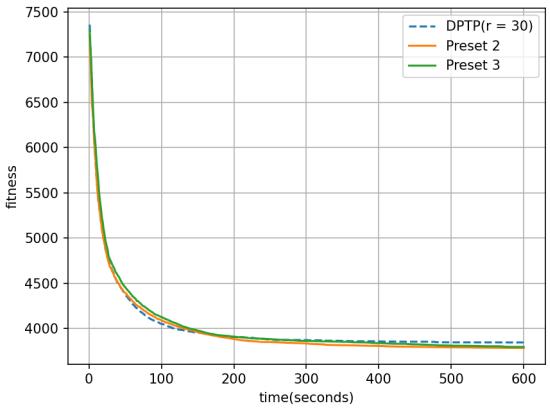
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 1, 4, 5, 6, 8 and 9, and better than results from presets 0, 2, 3, 7. According to this we classify the former as good presets, and the latter as bad presets.

6.2.3 Benchmark function C25

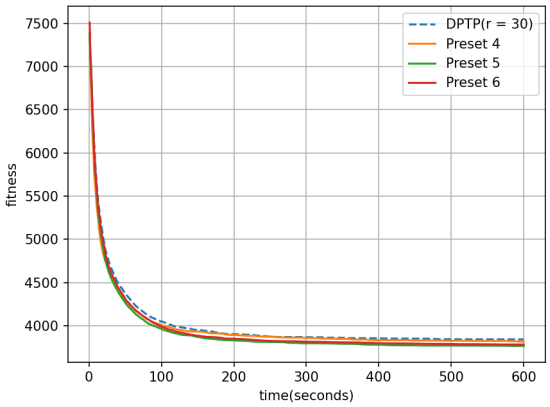
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 0, 1, 2, 3, 4, 5, 6, 8 and 9, and better than results from presets 7. According to this we classify the former as good presets, and the latter as bad presets.



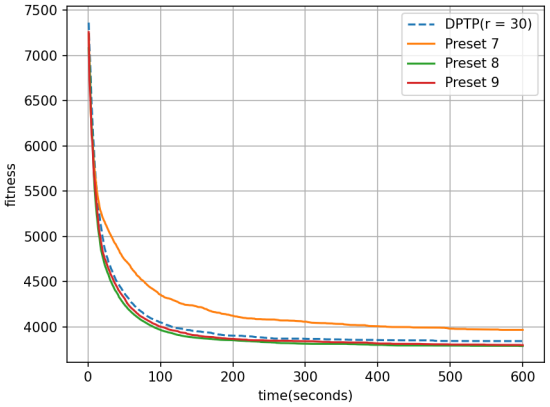
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



(d) DPTP vs. Preset 7, 8, 9

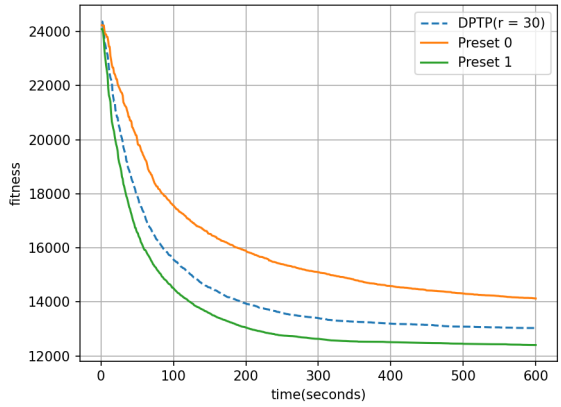
Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C25

6.2.4 Benchmark function C26

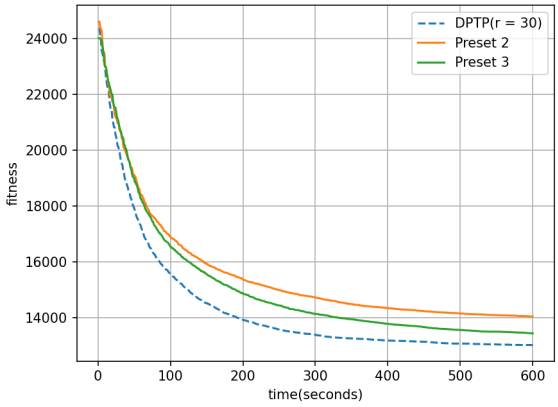
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 1, 4, 5, 6, 7, 8, and 9, and better than results from presets 0, 2, and 3. According to this we classify the former as good presets, and the latter as bad presets.

6.2.5 Benchmark function C27

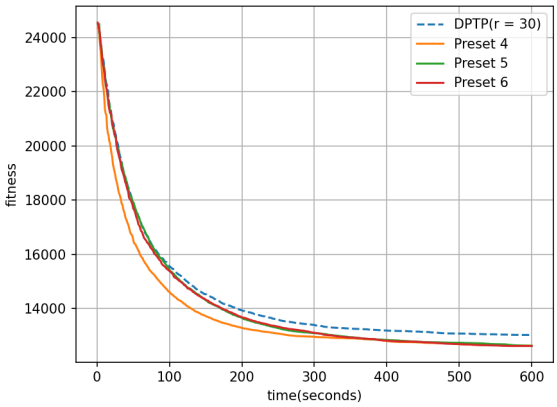
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 1, 4, 5, 6, 8 and 9, and better than results from presets 0, 2, 3 and 7. According to this we classify the former as good presets, and the latter as bad presets.



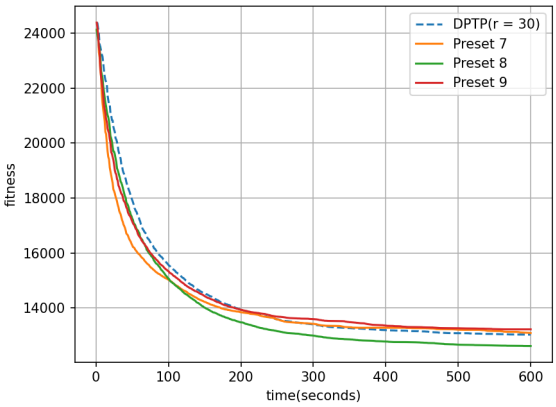
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



(d) DPTP vs. Preset 7, 8, 9

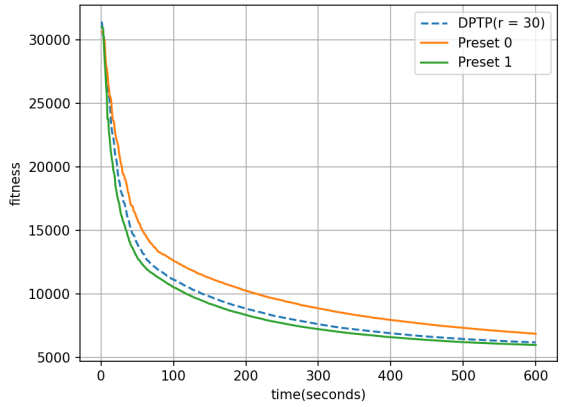
Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C26

6.2.6 Benchmark function C28

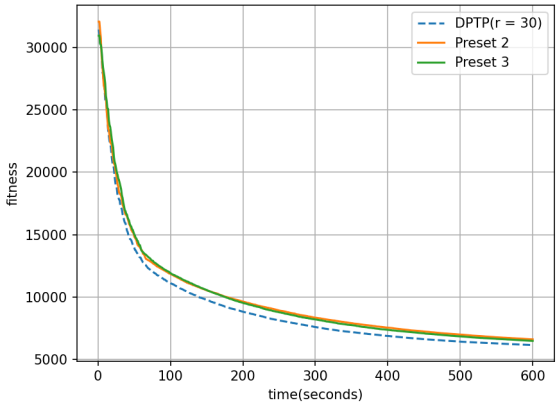
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 1, 4, 5 and 8, and better than results from presets 0, 2, 3, 6, 7 and 9. According to this we classify the former as good presets, and the latter as bad presets.

6.2.7 Benchmark function C29

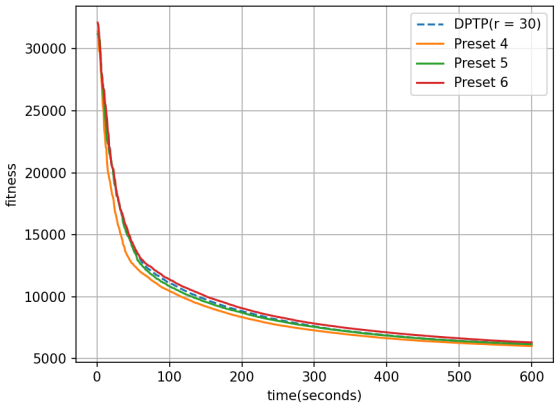
Figure.3 compares the optimization results from DPTP and fixed-preset groups, measured by benchmark function C23. We can see that DPTP results are close to or worse than the results from presets 1, 4, and 8, and better than results from presets 0, 2, 3, 5, 6, 7 and 9. According to this we classify the former as good presets, and the latter as bad presets.



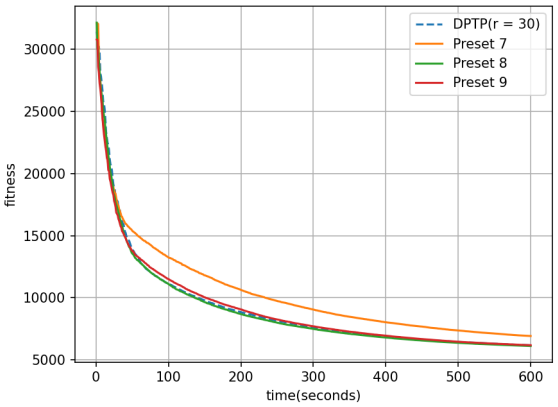
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



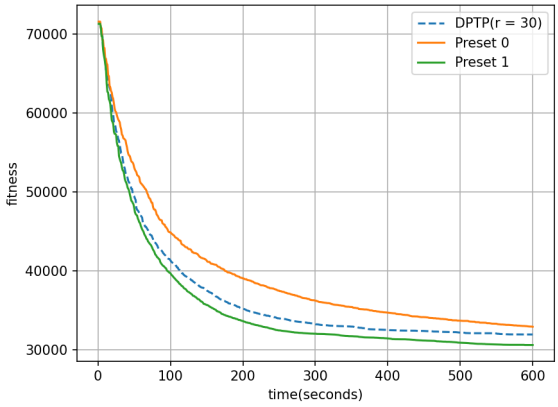
(d) DPTP vs. Preset 7, 8, 9

Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C27

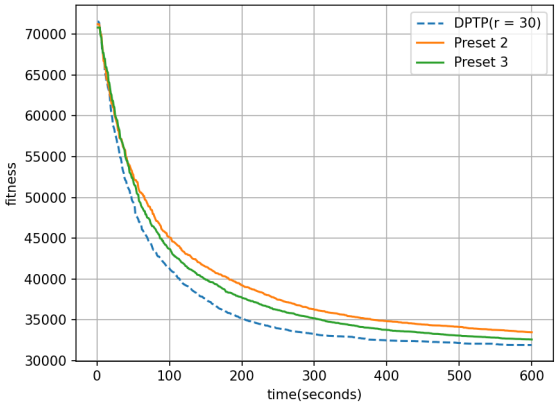
6.2.8 Summary of categorization

Table.4 lists a summary of good and bad presets for all the benchmark functions. From the table, we can see that from the benchmark function C23 to C29, we have respectively 6, 6, 9, 7, 6, 4, 3 good presets, and 4, 4, 1, 3, 4, 6, 7 bad presets.

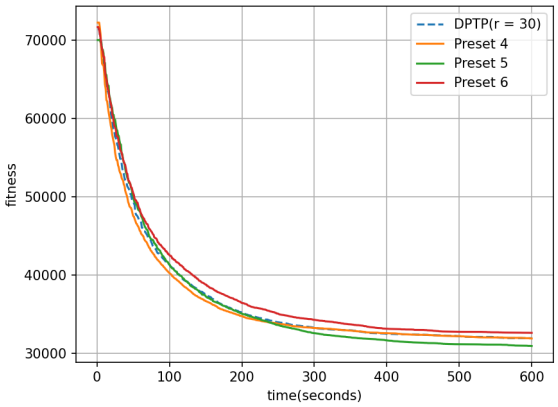
From the number of bad presets which we estimate are not as good as DPTP, it is hard to say that DPTP has an above-average performance. Compared to the presets, DPTP is better than some presets which we categorize as bad presets, but the condition about the presets that are categorized as good presets is hard to determine for now. We have no idea about if these good presets are close to or better than DPTP because many of their lines are twisted with the DPTP lines in the figures. Since such a categorization is just a rough estimation from the figures, it is not surprising that some categorizations are different from the test results so that the hypothesis H_0 is rejected. What we should care about is whether DPTP gains results that are equal to or better than at least half of the fixed presets, as the research question **RQ** marks. If so, we can say that the DPTP is an effective



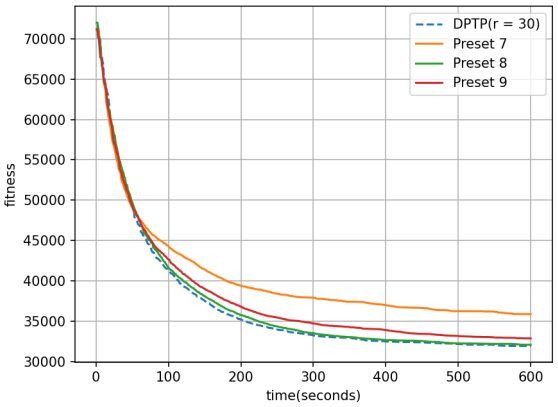
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



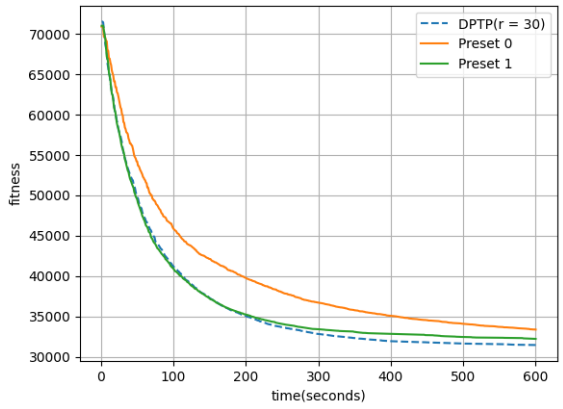
(d) DPTP vs. Preset 7, 8, 9

Figure 3: Experiment results from groups using DPTP vs. using fixed presets, benchmark function C28

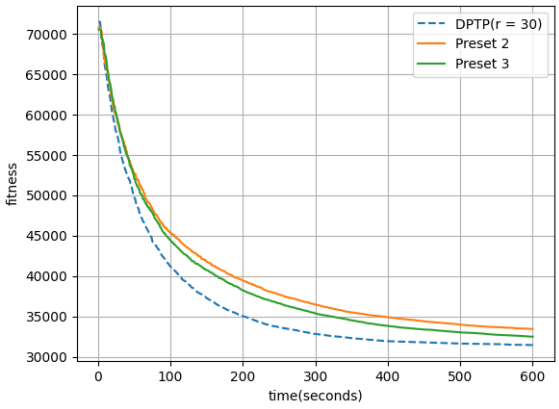
	C23	C24	C25	C26	C27	C28	C29
Good presets	1, 4, 5, 6, 8, 9	1, 4, 5, 6, 8, 9	0, 1, 2, 3, 4, 5, 6, 8, 9	1, 4, 5, 6, 7, 8, 9	1, 4, 5, 6, 8, 9	1, 4, 5, 8	1, 4, 8
Bad presets	0, 2, 3, 7	0, 2, 3, 7	7	0, 2, 3	0, 2, 3, 7	0, 2, 3, 6, 7, 9	0, 2, 3, 5, 6, 7, 9

Table 4: Good and bad presets recognized by figures for different test methods

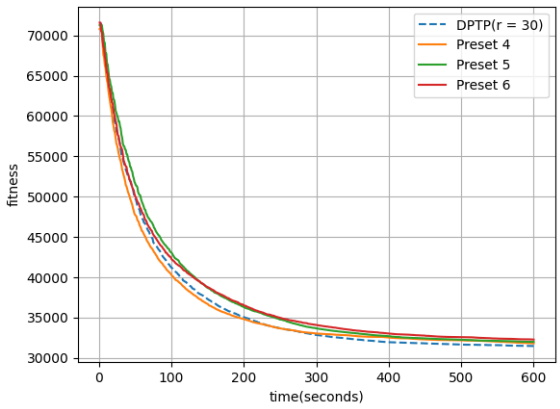
method for the parameter control.



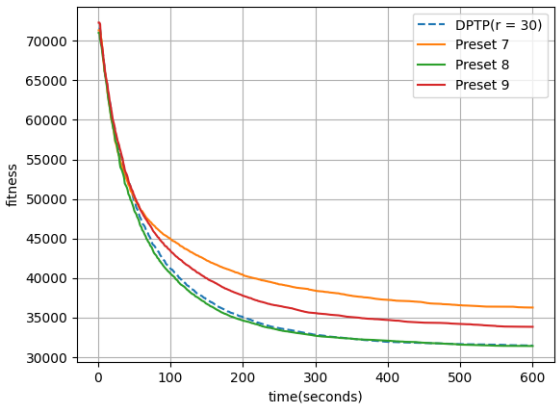
(a) DPTP vs. Preset 0, 1



(b) DPTP vs. Preset 2, 3



(c) DPTP vs. Preset 4, 5, 6



(d) DPTP vs. Preset 7, 8, 9

Figure 3: Experiment results from groups using DPTP vs.
using fixed presets, benchmark function C29

6.3 Test results and probability analysis

As described in Section 5.4.2 and 5.4.3, we compare the results from DPTP with presets by a right-side one-tailed test. The test results are presented in Section.6.3.1. We use the significance level $\alpha = 0.05$, so when the p-value less than 0.05: $p - value > \alpha = 0.05$ we accept H_0. Otherwise, we reject H_0.

In addition, we are also interested in the selection behavior of DPTP. Will DPTP select and use good presets more than bad presets? Whether DPTP gives good presets higher probabilities after a period of learning from the feedback and selects them more, or there can be other cases? After the good and bad presets are validated by the hypothesis test, we can compare the probabilities DPTP assigned to the presets to find answers. In Section.6.3.2, we collect the average probabilities of each preset when an experiment is terminated to see how the probabilities are distributed after DPTP learns from the whole process of experiment running.

6.3.1 Test results

Table.5 shows the p-values of the right-side one-tailed test comparing DPTP with different fixed presets. The columns of the table are for different benchmark functions, and the rows of the table are for different presets. The p-values that are smaller than the significance level $\alpha = 0.05$ are highlighted, meaning that in these cases the null hypothesis $H_0 : \bar{F}_{DPTP} \leq \bar{F}$ is rejected, and we accept the alternative hypothesis $H_1 : \bar{F}_{DPTP} > \bar{F}$.

	C23	C24	C25	C26	C27	C28	C29
Preset 0	1.00	1.00	0.00	1.00	1.00	1.00	1.00
Preset 1	0.17E-03	0.00	0.00	0.00	7.81E-06	0.00	1.00
Preset 2	1.00	1.00	0.00	1.00	1.00	1.00	1.00
Preset 3	1.00	1.00	1.45E-07	1.00	1.00	1.00	1.00
Preset 4	1.06E-06	0.00	8.18E-12	0.00	0.19E-03	0.33	1.00
Preset 5	0.92	0.00	0.00	0.00	0.31	3.17E-12	1.00
Preset 6	0.84	0.00	0.00	0.00	1.00	1.00	1.00
Preset 7	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Preset 8	0.08	0.00	0.00	0.00	0.09	0.99	0.21
Preset 9	0.59	8.23E-06	0.00	1.00	0.75	1.00	1.00

Table 5: Bad presets, p values of two-tailed test for the whole process

In the table, we can see that we have p-values that are larger than the significance level 0.05 in most of the cells (44 / 70), which means the null hypothesis $\bar{F}_{DPTP} > \bar{F}$ is accepted. This indicates that DPTP gains the above-average performance in most cases.

On the other hand, for different benchmark functions, the results of DPTP v.s. fixed presets are quite different. For some benchmark functions, such as C23, C28 and C29, DPTP is having an above average performance since it is overwhelming most of the fixed presets measured by these functions. But for benchmark function C24 and C25, DPTP didn't have the above-average performance. Figure.0 shows us the differences among these benchmark functions, i.e. they have different shapes and local optimalities. Combining with the test result, we can see that the basis problem or the benchmark functions with different optimalities may also affect the performance of DPTP.

In addition, the test results show a high consistency with our figures for the experiment results in Section.6.2. Though some curve lines are illegible because they are too close to each other, we can still see some curve lines that are obviously better worse than DPTP. For example, we can clearly see that the curve line for preset 7 measured by C25 in Figure.3, Preset 0 by C28 in Figure.3, and preset 7 & 8 measured by C29 in Figure.3, etc., are worse than the results of DPTP, and we can see that in these cases, the null hypothesis is accepted by test results. Similarly, good presets like preset 1 measured by C24 in Figure.3 and preset 1 & 4 measured by C26 in Figure.3 are better than DPTP. In such cases, the null hypothesis is rejected, indicating that DPTP is significantly worse than them.

From the test results, Table.6 and Figure.6 give a summary of numbers that presets DPTP is equal

to or better than for each benchmark function. The numbers from different benchmark function groups show how effective DPTP is. If there are at least half of the presets, 5 out of 10 here, that are not as good as DPTP, we say that DPTP has gained an above-average performance.

	C23	C24	C25	C26	C27	C28	C29
No. of winning	8	4	1	5	8	8	10

Table 6: The number of presets that DPTP is better than or equal to for each benchmark function

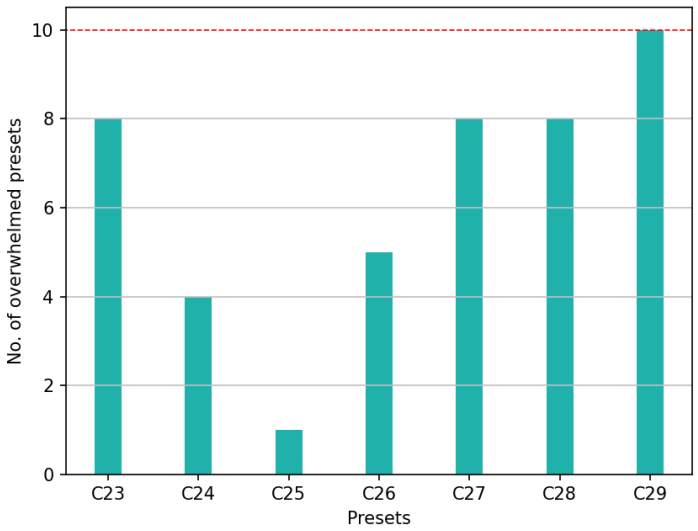


Figure 6: Number of presets DPTP overwhelms measured by different benchmark functions

Table.6 and Figure.6 show that for most of the benchmark functions, DPTP has an above-average performance defined by the research question **RQ**. This result is consistent with the observation from figures in Section.6.2, since we can see the lines representing DPTP are mostly on the bottom part of the figures compare to other lines representing fixed presets, indicating that DPTP always gives the basis algorithm quite low fitness values for most of the benchmark functions. As for benchmark functions C25 and C26, where DPTP does not have an above-average performance, the curve lines representing DPTP in Figure.3 and Figure.3, we can hardly tell the gap between DPTP and the good presets except preset 1 measured by C24. The curve lines of DPTP and most good presets in these two figures are just jammed together to show an illegible picture. This indicates that the differences between DPTP and the good presets are not that large.

6.3.2 Preset selection Probabilities

The probabilities of presets are what DPTP selects presets by. The higher the probability of a preset is, the more likely the preset is selected. In this section we make a qualitative analysis of the selection probabilities. As described in Section.5.3.2, we repeat an experiment with the same setup

and measured by the same benchmark function for 5 times and use the average data obtained from these experiments. The probability values we use in this section are also the average results from the 5 repeated runs. Besides, we only use the probabilities from the last updates in each run to make the analysis, since such probabilities are assigned by DPTP after learning from the feedback of the whole process. Since their values are determined by more feedback information, they are more representative of how the presets are selected and used by DPTP during the whole run. Table.7 presents the average final probabilities of each preset measured by different benchmark functions. In order to make the analysis, we need to make sure that the good or bad presets we use are truly good or bad. If we think a preset is good, but it is actually bad preset, and it has a low final probability, the situation really causes some confusion for us.

	C23	C24	C25	C26	C27	C28	C29
Preset 0	0.0732	0.0547	0.0209	0.1625	0.0802	0.0732	0.0394
Preset 1	0.0759	0.0742	0.0905	0.1049	0.0490	0.0759	0.2596
Preset 2	0.1142	0.0127	0.3087	0.1514	0.1049	0.1142	0.0926
Preset 3	0.0555	0.0274	0.0927	0.1232	0.0951	0.0555	0.0372
Preset 4	0.1024	0.0965	0.0735	0.0488	0.1026	0.1024	0.1816
Preset 5	0.1641	0.1707	0.0473	0.0607	0.0951	0.1641	0.0878
Preset 6	0.0745	0.0797	0.0683	0.0363	0.1271	0.0745	0.0382
Preset 7	0.0736	0.1289	0.2544	0.1701	0.1350	0.0736	0.0412
Preset 8	0.0574	0.1152	0.0169	0.0518	0.1078	0.0574	0.1687
Preset 9	0.2092	0.2400	0.0267	0.0904	0.1032	0.2092	0.0538

Table 7: The average final probabilities updated by DPTP.
Probabilities of the bad presets are highlighted.

Since DPTP shows good performance in both Section.6.2 and 6.3.1, graphically and mathematically, it makes sense that we expect DPTP gives higher probabilities to good presets, and lower probabilities to bad presets, so that it selects and uses more good presets. However, we can see from Table.7 that the bad presets do not always get the lowest probabilities in the final stages of an experiment run. Moreover, some bad-preset probabilities even rank very high among all the 10 presets, e.g., preset 2 has the third-highest probability for benchmark function C23, preset 7 has the highest probability for benchmark function C25, and preset 0 has the second-highest probability for benchmark function C26, etc. In other words, there is inconsistency between the good performance of DPTP and some high probabilities of bad presets can be guessed, implying that the good performance and the good feedback from the running time is not merely the result of always selecting the good presets, or the good presets might not always be the good presets.

We are not going further to study the reasons for such inconsistency in this paper for DPTP has been validated to be an effective method for parameter control after all. But we can make some conjectures about the possible reasons. First, we cannot ignore the existence of the randomness. If DPTP happens to select a bad presets at the end of the run, and the bad preset happens to have even a little improvement, then the probability of the preset will have a quite large weight, since in the final stage of a run, any improvement is harder to get compared to the initial stage of a run.

Actually, it is not that easy to ignore the problem that the randomness affects much more of the weights and the probabilities of presets at the end of an experiment run. In DPTP, the updating of preset probabilities is based on the improvements and the time costs. However, if we run the model long enough, then in the final stages of the running, improvement would be very hard to obtain. In this condition, any improvement can lead to a relative high probability to the preset that is used in that iteration. For example, if we have 3 presets, in which preset $1, 2, 3$ are respectively the best, medium, and the worst presets, with the probabilities of $0.9, 0.05$ and 0.05. Assume that between two updates of probabilities, we try all the presets, and only the worst preset, preset 3 get some improvement, say 1, in unit time. Then we update the probabilities by Algorithm.4, and the probabilities are $0.81, 0.045$ and 0.145 (take the value of the effect factor $\beta = 0.9$). Bad presets can get higher probabilities by such randomness, which is something we don't want.

On the other hand, it might not be the best strategy to always use the good presets for optimization. It is well known that in many optimization problems, if we always make the choice that seems to be the best at the moment, we may get a local optimality finally. Thus, in order to find out the global optimality, we need to make some decisions that seems not to be the best at the moment to jump out of the trap of the local optimality. From the intricate maps of benchmark functions in Figure.0, we have already seen the complexity of them. So we guess that using bad presets can help us to jump out of the local optimality of these benchmark functions. If so, DPTP is smarter than we think since it knows more about the problem from the feedback of running time.

6.4 Section summary

In this section, we first list the numerical data of the improvements obtained by subtracting the initial optimization values from the final ones in each experiment. Numerical results show that measured by 5 out of 7 benchmark functions, DPTP has the same or more improvements compared to at least half of the fixed presets. In the graphs plotted based on the numerical data, we can also roughly see that measured by 5 out of 7 benchmark functions, DPTP is better than or equal to the fixed presets, while in other cases, DPTP performs worse than over half of the fixed presets. However, with either the numerical or graphical method, the comparison is just a rough estimation. Therefore, we perform the hypothesis test and present the results in Section.6.3 by using the Wilcoxon rank-sum test method. The test results significantly validate that measured by 5 our of 7 benchmark functions, DPTP is better than or equal to the fixed presets. Thus, we can conclude that in most of the cases, DPTP is more likely to have an above-average performance, according to the test results from Section.6.3.1.

7 Conclusion and discussion

In this paper, we design an algorithm, DPTP, to tune the parameters of metaheuristics while it is running according to the feedback from the running time. This section draws some conclusions from the test results in Section.6. These conclusions are given in Section.7.1. In addition, we also find some possible improvements that can be done in the future for the algorithm and limitations in the research that deserve to be discussed. These are presented in Section.7.2.

7.1 Conclusion

The experiment data and the test results show that DPTP has a very good performance: it outperforms most of the fixed presets and gains above-average performance in 5 of 7 comparisons with fixed presets. Going back to the research question in Section.2.2:

How to design an algorithm to select parameter values for a metaheuristic when it is running, so that the metaheuristic can gain above-average performance, compared to only applying a fixed preset to the metaheuristic?

We answer the research question by answering its sub-questions.

• *Since we tend to let the tuning algorithm, DPTP, determine the parameter values without any pilot test or prior knowledge about the metaheuristic, based on what criteria should it make decisions about the parameter values?*

We name the algorithm for which the parameter values need to be adjusted as the "basis algorithm", and the algorithm DPTP designed in this paper to adjust the parameter values for the basis algorithm as the "tuning algorithm". To use DPTP, we need to provide the presets, which are just alternative sets of parameter values for the basis algorithm. These presets can be generated by completely random values in sensible ranges of the parameters. They have attributes like names and probabilities, etc. As for determining the number of the presets provided for DPTP, we need to first consider the limitation for the running time. For each preset, we calculate its probability by the improvements and the time spent on the improvements when the preset is used. DPTP needs enough running time to collect feedback and learn from it. Thus, if we provide more presets for DPTP, we need to extend the cycle length so that enough running time for presets and the learning time for DPTP can be ensured. Whereas, longer cycle length means longer running time is needed to get a final solution. On the other hand, if we provide too few presets for DPTP, the effective values in the parameter value ranges may not be explored and used in the preset. Thus, users need to make a balance between the number of presets and the running time when generating presets for DPTP.

After the basis algorithm starts running, there is a selector to select the presets that are to be used by the basis algorithm based on their probabilities. The probabilities of presets are initialized with the value $1/n$, where n is the number of provided presets, and will be updated based on the performance of the presets. Since different parameter values affect the performance of the basis algorithm, we can use improvements the basis algorithm gets per unit time to measure the effectiveness of the selected presets by the DPTP selector. The presets that help the basis algorithm gain higher improvements during the unit time get higher probabilities, and they are considered "good" presets. Otherwise, the presets that does not help the basis algorithm improve much, or even no improvement, are deemed "bad" presets. In other words, we use the feedback from the running time of the basis algorithm to determine the values of the parameters. That is why we do not need any pilot test or prior knowledge of the basis algorithm to adjust its parameter values. All in all, the criteria on which we make decisions about the parameter values are the effectiveness of the presets provided for DPTP, measured by the improvements they help the basis algorithm get per unit time.

• *Once we determine the criteria and method based on which DPTP makes decisions about selecting*

presets, how to collect the data about the performance of DPTP as well as fixed values, and how should we use these data to improve DPTP's selection making?

The data is collected while the basis algorithm is running, so we do not need extra time to run pilot tests for testing which values can give the basis algorithm good performance. Then we update the preset probabilities periodically. For using the data, we calculate the average improvements per unit time of all the presets, and then use the unit-time improvements to calculate the probabilities of these presets.

It is remarkable that sometimes, some presets are not selected and used between two updates of the probabilities. Thus, when we calculate the new probabilities, their weights are badly affected because they have 0 improvement for the basis algorithm, so they get lower probabilities in such a condition. However, if they were used they might have gotten good improvements. Therefore, if we give them low probabilities for no improvements are made by them because they are not selected, the low probabilities are not due to bad performance, but from the randomness that causes them not to be selected and used. So we design another algorithm, Algorithm.4, of updating probabilities with some "protection" for these unused presets. That is, we just keep their current probabilities the same and use it until they are selected and get some feedback.

As described above, we provide some alternative presets to DPTP, then select one from the provided presets by their probabilities. After we use the presets, we calculate and update the new probabilities according to the improvements the basis algorithm get and the time the basis algorithm takes for obtaining such improvements when use the presets. We can see that compared to using fixed parameter values, DPTP dynamically adjusts the parameter values by the running time feedback.

- *How to design and execute experiments and data analysis to validate that the metaheuristic using the tuning method gains above-average performance, compared to only using fixed parameter values during the running?*

In the experiments, we compare the effectiveness of DPTP by comparing it with the effectiveness of the fixed values over the whole run of the basis algorithm. If the result of using DPTP is better than or equal to a fixed-preset group, we say that the DPTP group "outperforms" this fixed-preset group. First we generate 10 presets randomly. We provide these presets as the input to DPTP. Meanwhile, they are also the fixed values we use for comparing with DPTP. Then we run basis algorithms with separately using DPTP and the 10 presets, and collect the data from these experiments. Measured by some benchmark functions, we can see from the resulting data if DPTP overwhelms over half of the fixed presets, which is, the above average performance.

For more solid validation, we also use the hypothesis test to test the relationship among the resulting data from DPTP and the other 10 presets. The hypothesis test can significantly recognize that if the effectiveness of DPTP is equal to the good presets, or better than the bad presets. By the test results, we can have an insight about how many fixed-preset groups the DPTP group outperforms.

- *What How does DPTP affect the performance of the metaheuristics compared to using only fixed presets, according to the experiment data?*

The data visualization and the test results in Section.6 suggest that in most cases, DPTP can effectively select the presets that can help the basis algorithm gain better performance, i.e. gain

more improvements during the same time, or gain the same improvements during shorter time, compared with the only using fixed parameter values over the whole run. It updates the selection probabilities for presets by the feedback from the iterations using different presets. Both the figures in Section.6.2 and the test results in Section.6.3 indicate that DPTP helps the basis algorithm gain an above-average performance defined in **RQ**. The results from DPTP are at least as good as half of the fixed presets it uses. In the test result, DPTP outperforms 5 our of 7 fixed-preset groups.

However, it is remarkable that in few cases DPTP does not have the above-average performance. This implies that DPTP is still possible to be affected by the specific problems or benchmark functions. Thus, we can only say that in most cases, using DPTP is better than using fixed values for parameters of the basis algorithm, according to the experiment results we have.

• *If DPTP has a satisfactory above-average effectiveness, what is its behavior like with regard to the selection of presets?*

In Section.6.3.2, we find that DPTP does not always give good presets high probabilities at the final stages of the algorithm run as we expect. We make some conjectures about the reason why it does so: The possible reason can be that the running time is not enough for the probabilities to converge. We run each experiment for 10 minutes, and it can be shorter than the required time for the probabilities to converge. Also, It could be the reason of the randomness, so that DPTP happens to select the bad presets at the final stages, and using these presets gives good feedback which increases the probabilities of the bad presets. Table.7 shows that at the final stage, some bad presets' probabilities does not really converge to 0. This implies that the randomness still plays an important role in the process of DPTP assuming that we have run the algorithm for enough time. Another possible reason is simply because that we have not run the basis algorithm for enough time. In the experiments, each run takes 10 minutes. Such a running time is long enough for the benchmark function values to converge, as we can see in the experiment results. But the convergence of the benchmark function values does not mean the convergence of the preset probabilities. Thus, to validate this, we need longer running time to observe if the probabilities converge.

Since DPTP gains above-average performance in most of the cases for the basis algorithm in this paper, it is validated to be a promising direction for parameter control of metaheuristic algorithms. However, it still has some limitations. DELMIA continues the research on DPTP after this paper is completed, focusing on why DPTP fails in some cases, and how to improve it to make it work better in these cases. Additionally, they will build the algorithm into their QUINTIQ platform to apply it to more types of real optimization problems in their business, so that the generality of DPTP can be further validated.

7.2 Limitations and further research

The experiments and the test results validates the high effectiveness of DPTP because it does better than most of the randomly generated fixed presets, but there are still some limitations that can be optimized in further researches. In this section, we discuss some limitations so far in our research, and hopefully we can research more on these topics.

Generality of the research An ideal situation for the experiment results of the research is that results from different experiment setups are distinguished from each other significantly. We want the results to be more dispersed rather than converging from the very beginning. If there are clear hierarchies among groups with different setups, so larger dispersion, we can better see in what position DPTP is for a better analysis. But with presets that gives similar results, first it is hard to differentiate them easily in a graphical way. Though the mathematical test helps us differentiate the good and bad presets, comparing DPTP to too many similar results is hard and meaningless, for they are arguably the same or similar results and when the test results indicate that DPTP is better than or equal to these results, it is hard to say that DPTP really outperforms most of the presets. The experiment results and figures show that there are still many of the resulted lines jammed together. For example, in the following Figure.7 and 7, we can see that the dispersion degrees are different. We want to eliminate the results of fixed-preset groups that are too close to each other and increase the dispersion of these groups as much as possible in future researches.

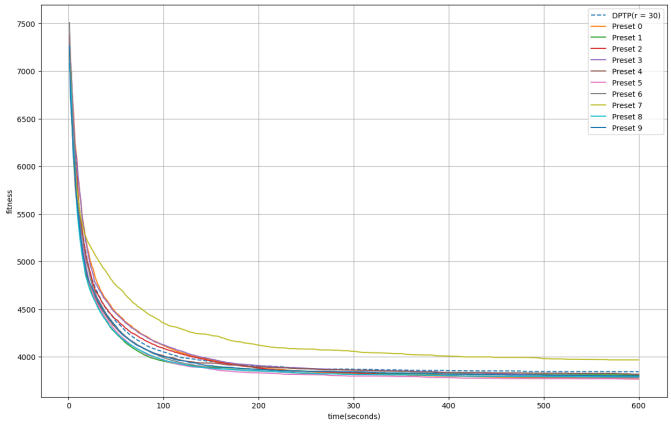


Figure 7: Experiment results of C25 with smaller dispersion, the results of different groups are close to each other and hard to differentiate

The reason of the jammed results can be the problem of generated presets. If we do not intend to use any pre-knowledge about the basis algorithm and use completely random values to generate presets, it is likely that we get the presets that lead to similar results without small degree of dispersion.

Besides, another reason that the experiment results are not easily distinguished from each other can be the basis algorithm we use in the research. We only use one basis algorithm (the CRO) in this paper to test the effectiveness of DPTP, which lacks the generality of validation, though with up to 7 benchmark functions. The selected basis algorithm may be not that sensitive to the change of parameter values. Therefore, we can use more basis algorithm in the future researches to test DPTP. What's more, we see some cases that DPTP does not have an above-average performance as mentioned in the last section. This problem also needs to be studied more. We can try DPTP on more basis algorithm to figure out what kind of problems DPTP does not fit into and how to improve it.

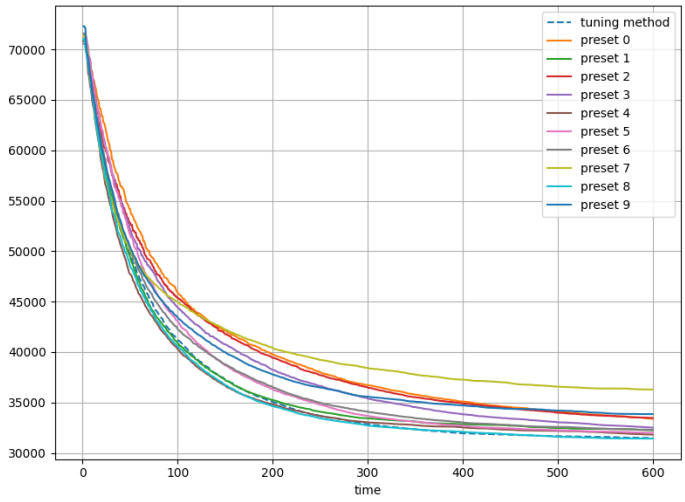


Figure 7: Experiment results of C29 with larger dispersion, the results of different groups are easier to be distinguished

Quantitative study For now, we only have a conclusion that DPTP has an above-average effectiveness among all the setups, but we don't know how much it can improve the basis algorithm quantitatively compared to fixed presets. Such knowledge can be used in the decision-making process whether we use DPTP or just fixed values when we run the basis algorithm. DPTP is more complicated than just using fixed values, so for some simple optimization problems, if we anticipate that the improvement DPTP can make is not high enough to offset the complexity of using DPTP, we can just use fixed values instead to quickly get the results.

Effect of randomness in final stages of a run In Section.6.3.2, we depict a situation that in the final stages of a run, when any improvement is really hard to be obtained, very tiny improvements can lead to quite high probabilities when other presets do not make any improvements. This is the negative effect to the accuracy of DPTP we want to avoid.

To eliminate such effect, a possible solution can be the introduction of a new variable. The value of it decreases as the basis algorithm runs. Such a variable is the weight of improvements obtained. Thus, when the run is at final stages, the weight variable gets a smaller value and the effect of improvements is less and less. In this way, the initial value of the variable and the speed by which its value decreases should be finely set. However, such a solution causes another problem, too many parameters for DPTP. The original intention of DPTP is to get rid of the decision-making process about the parameter values for the basis algorithm, and just let DPTP do the job automatically. If we introduce too many parameters for DPTP, even more than the number of the parameter of the basis algorithm, we need to still consider the value problem of DPTP. Then we find DPTP meaningless in the parameter controlling.

Smarter presets generating In this paper, we provide DPTP with some alternative presets to make DPTP be able to select the presets based on the feedback from the running time, so that it gives the basis algorithm a good performance. However, the provided presets also constrain DPTP in a given restricted range. DPTP can only select possible values from the given presets. We generate the presets by random values in the parameter ranges. The more presets we make and the smaller the parameter ranges are, the more likely that the randomly generated values get close to the values that can really boost the basis algorithm. When the value range of a parameter is very large, The randomly generated values in the presets might not work that well. A large range means that it is harder to randomly generate values that are effective for the basis algorithm. So for larger ranges, we have to provide more presets to DPTP, and more presets requires longer cycle length. Thus, we want a smarter way of generating and using presets. It is better if we find a way to make DPTP to not only use the given presets, but also creating presets themselves while it is being executed.

References

Aarts, E., Aarts, E. H., & Lenstra, J. K. (2003). *Local search in combinatorial optimization.* Princeton University Press.

Abdollahzadeh, B., Soleimanian Gharehchopogh, F., & Mirjalili, S. (2021). Artificial gorilla troops optimizer: a new nature-inspired metaheuristic algorithm for global optimization problems. *International Journal of Intelligent Systems*, *36*(10), 5887–5958.

Abdulkarim, H. A., & Alshammari, I. F. (2015). Comparison of algorithms for solving traveling salesman problem. *International Journal of Engineering and Advanced Technology*, *4*(6), 76–79.

Ahmed, A. M., Rashid, T. A., & Saeed, S. A. M. (2020). Cat swarm optimization algorithm: a survey and performance evaluation. *Computational intelligence and neuroscience*, *2020*.

Ahuja, R. K., Ergun, Ö., Orlin, J. B., & Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, *123*(1-3), 75–102.

Aleti, A., & Moser, I. (2016). A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Computing Surveys (CSUR)*, *49*(3), 1–35.

Altinbas, H., & Akkaya, G. C. (2017). Improving the performance of statistical learning methods with a combined meta-heuristic for consumer credit risk assessment. *Risk Management*, *19*(4), 255–280.

Arcuri, A., & Fraser, G. (2011). On parameter tuning in search based software engineering. In *International symposium on search based software engineering* (pp. 33–47).

Bartz-Beielstein, T. (2010). Spot: An r package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. *arXiv preprint arXiv:1006.4645*.

Bartz-Beielstein, T., Lasarczyk, C., & Preuss, M. (2010). The sequential parameter optimization toolbox. In *Experimental methods for the analysis of optimization algorithms* (pp. 337–362). Springer.

Bemporad, A., Morari, M., Dua, V., & Pistikopoulos, E. N. (2002). The explicit linear quadratic regulator for constrained systems. *Automatica*, *38*(1), 3–20.

Bui, D. T., Panahi, M., Shahabi, H., Singh, V. P., Shirzadi, A., Chapi, K., ... others (2018). Novel hybrid evolutionary algorithms for spatial prediction of floods. *Scientific reports*, *8*(1), 1–14.

Chong, E. K., & Zak, S. H. (2004). *An introduction to optimization.* John Wiley & Sons.

Conover, W. J. (2007). *Practical nonparametric statistics.* Academic Internet Publishers.

Dassault Systèmes - Wikipedia. (2016, jun 1). en.wikipedia.org. ([Online; accessed 2022-07-12])

De Filippo, A., Lombardi, M., & Milano, M. (2021). The blind men and the elephant: Integrated offline/online optimization under uncertainty. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence* (pp. 4840–4846).

Delmia quintiq – dassault systèmes. (2022). Dassault systèmes. Retrieved from `https://www.3ds.com/products-services/delmia/products/delmia-quintiq/` ([Online; accessed 2022-07-12])

Desale, S., Rasool, A., Andhale, S., & Rane, P. (2015). Heuristic and meta-heuristic algorithms and their relevance to the real world: a survey. *Int. J. Comput. Eng. Res. Trends*, *351*(5), 2349–7084.

Eftimov, T., Korošec, P., & Seljak, B. K. (2017). A novel approach to statistical comparison of meta-heuristic stochastic optimization algorithms using deep statistics. *Information Sciences*, *417*, 186–215.

Elmazi, D., Oda, T., Sakamoto, S., Spaho, E., Barolli, L., & Xhafa, F. (2015). Friedman test for analysing wmns: A comparison study for genetic algorithms and simulated annealing. In *2015 9th international conference on innovative mobile and internet services in ubiquitous computing* (pp. 171–178).

Faramarzi, A., Heidarinejad, M., Stephens, B., & Mirjalili, S. (2020). Equilibrium optimizer: A novel optimization algorithm. *Knowledge-Based Systems*, *191*, 105190.

Goldsman, D., Nelson, B. L., & Banks, J. (1998). Comparing systems via simulation. *Handbook of simulation*, 273–306.

Hansen, J. M., Raut, S., & Swami, S. (2010). Retail shelf allocation: A comparative analysis of heuristic and meta-heuristic approaches. *Journal of Retailing*, *86*(1), 94–105.

Hinterding, R., Michalewicz, Z., & Eiben, A. E. (1997). Adaptation in evolutionary computation: A survey. In *Proceedings of 1997 ieee international conference on evolutionary computation (icec'97)* (pp. 65–69).

Hoos, H. H. (2011). Automated algorithm configuration and parameter tuning. In *Autonomous search* (pp. 37–71). Springer.

Huang, C., Li, Y., & Yao, X. (2019). A survey of automatic parameter tuning methods for metaheuristics. *IEEE transactions on evolutionary computation*, *24*(2), 201–216.

Johnson, D. S., Papadimitriou, C. H., & Yannakakis, M. (1988). How easy is local search? *Journal of computer and system sciences*, *37*(1), 79–100.

Liang, J. J., Qu, B. Y., & Suganthan, P. N. (2013). Problem definitions and evaluation criteria for the cec 2014 special session and competition on single objective real-parameter numerical optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore*, *635*, 490.

Maier, H. R., Razavi, S., Kapelan, Z., Matott, L. S., Kasprzyk, J., & Tolson, B. A. (2019). Introductory overview: Optimization using evolutionary algorithms and other metaheuristics. *Environmental modelling & software*, *114*, 195–213.

Michalewicz, Z., & Fogel, D. B. (2013). *How to solve it: modern heuristics*. Springer Science & Business Media.

Michiels, W., Aarts, E. H., & Korst, J. (2007). *Theoretical aspects of local search* (Vol. 13). Springer.

Mousavirad, S. J., & Ebrahimpour-Komleh, H. (2017). Human mental search: a new population-based metaheuristic optimization algorithm. *Applied Intelligence*, *47*(3), 850–887.

Moussa, D. A., Eissa, N. S., Abounaser, H., & Badr, A. (2018). Design of novel metaheuristic techniques for clustering. *IEEE Access*, *6*, 77350–77358.

Nguyen, T., Nguyen, B. M., & Nguyen, G. (2019). Building resource auto-scaler with functional-link neural network and adaptive bacterial foraging optimization. In *International conference on theory and applications of models of computation* (pp. 501–517).

Nguyen, T., Tran, N., Nguyen, B. M., & Nguyen, G. (2018). A resource usage prediction system using functional-link and genetic algorithm neural network for multivariate cloud metrics. In *2018 ieee 11th conference on service-oriented computing and applications (soca)* (pp. 49–56).

Osman, I. H., & Laporte, G. (1996). *Metaheuristics: A bibliography.* Springer.

Pannocchia, G., Rawlings, J. B., & Wright, S. J. (2007). Fast, large-scale model predictive control by partial enumeration. *Automatica*, *43*(5), 852–860.

Pearce, J., & Derrick, B. (2019). Preliminary testing: the devil of statistics? *Reinvention: An International Journal of Undergraduate Research*, *12*(2).

Pholdee, N., & Bureerat, S. (2018). A comparative study of eighteen self-adaptive metaheuristic algorithms for truss sizing optimisation. *KSCE Journal of Civil Engineering*, *22*(8), 2982–2993.

Pirlot, M. (1996). General local search methods. *European journal of operational research*, *92*(3), 493–511.

Pratt, J. W., & Gibbons, J. D. (1981). *Concepts of nonparametric theory.* Springer-Verlag.

Rardin, R. L., & Uzsoy, R. (2001). Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, *7*(3), 261–304.

Ravey, A., Blunier, B., & Miraoui, A. (2011). Control strategies for fuel cell based hybrid electric vehicles: From offline to online. In *2011 ieee vehicle power and propulsion conference* (pp. 1–4).

Reeves, C. R. (1993). *Modern heuristic techniques for combinatorial problems.* John Wiley & Sons, Inc.

Saha, S., Seal, D. B., Ghosh, A., & Dey, K. N. (2016). A novel gene ranking method using wilcoxon rank sum test and genetic algorithm. *International Journal of Bioinformatics Research and Applications*, *12*(3), 263–279.

Salcedo-Sanz, S. (2016). Modern meta-heuristics based on nonlinear physics processes: A review of models and design procedures. *Physics Reports*, *655*, 1–70.

Salcedo-Sanz, S., Del Ser, J., Landa-Torres, I., Gil-López, S., & Portilla-Figueras, J. (2014). The coral reefs optimization algorithm: a novel metaheuristic for efficiently solving optimization problems. *The Scientific World Journal*, *2014*.

Skakov, E., & Malysh, V. (2018). Parameter meta-optimization of metaheuristics of solving specific np-hard facility location problem. In *Journal of physics: Conference series* (Vol. 973, p. 012063).

Stützle, T. (1999). Local search algorithms for combinatorial problems: analysis, improvements, and new applications.

Sun, S., & Lu, H. (2019). Self-adaptive parameter control in genetic algorithms based on entropy and rules of nature for combinatorial optimization problems. In *2019 ieee symposium series on computational intelligence (ssci)* (pp. 2721–2728).

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation.* John Wiley & Sons.

Thierens, D. (2005). An adaptive pursuit strategy for allocating operator probabilities. In *Proceedings of the 7th annual conference on genetic and evolutionary computation* (pp. 1539–1546).

Van Hentenryck, P., Bent, R., & Upfal, E. (2010). Online stochastic optimization under time constraints. *Annals of Operations Research*, *177*(1), 151–183.

Van Veldhuizen, D. A., & Lamont, G. B. (2000). On measuring multiobjective evolutionary algorithm performance. In *Proceedings of the 2000 congress on evolutionary computation. cec00 (cat. no. 00th8512)* (Vol. 1, pp. 204–211).

What we are. (2021, Jun). Dassault systèmes. Retrieved from `https://www.3ds.com/about-3ds/what-we-are` ([Online; accessed 2022-07-10])

Wikipedia contributors. (2022a). *Statistical hypothesis testing — Wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/w/index.php?title=Statistical_hypothesis_testing&oldid=1098096307` ([Online; accessed 23-July-2022])

Wikipedia contributors. (2022b). *Time complexity — Wikipedia, the free encyclopedia.* Retrieved from `https://en.wikipedia.org/w/index.php?title=Time_complexity&oldid=1095794941` ([Online; accessed 20-August-2022])

Wilcoxon, F. (1946). Individual comparisons of grouped data by ranking methods. *Journal of economic entomology*, *39*(2), 269–270.

Appendix A Coral Reef Optimization

The coral reefs optimization (CRO) algorithm, just like the other genetic metaheuristics, is inspired by the process of the natural selection. It simulates the process of the formation and reproduction of coral reefs in different phases. In this section, we introduce how CRO emulates such a process and how it is used for optimization problems.

An important subgroup of corals is reef-building corals, also known as herbaceous corals or simply hard corals. Hard corals are usually shallow water animals that produce hard calcium carbonate skeletons. Coral reefs are made up of hundreds of hard corals connected by their calcium carbonate. Polyps periodically lift from the basal plate and secrete new calcium carbonate, creating chambers that help coral bones grow. Polyps keep building these chambers on the reef, so eventually the entire reef grows larger and larger. Living coral grows on the calcium carbonate bones of their dead ancestors. Coral reefs are usually made up of corals that live in groups or independently. A coral reef usually consists of single type corals, but can also be the environment where multiple other species is located. In addition to coral, many other floras and animals live on coral reefs, such as fish, algae, sponges, sea anemones, mosses, starfish, crustaceans (e.g. shrimp, crab, lobster), octopuses, squid, clams, snails and other mollusks.

Corals require free space to grow. In the real world, free space is usually a kind of limited resource that leads to badly competition between different coral species. One of the strategies the corals mostly use is growing fast. The corals that grow faster easily grow on top of those that grow slower and eventually kill the latter in this way.

The behavior described above for reproduction and competing for more growing space gives inspiration to the use of CRO. Some strategies that CRO emulates include:

- **External reproduction: broadcast Spawning** Sometimes, coral reefs produce a huge amount of male or female gametes, and they are released into the water. When an egg and a sperm meet each other, they combine to produce a larva. A larvae looks for a free space to grow as a polyp. On large reefs, such spawning phenomenon usually happens synchronously, otherwise the encounter will be hard to happen between gametes generated by different corals.
- **Internal reproduction: brooding** Brooding is a method of internal reproducing. With this reproduction mode, some female corals contain eggs that are not release the water. Then the sperm released by other corals of the same species gets inside the female coral and fertilize the egg. The fertilized egg then produce a larva. When the larva grows older, it is released to the water in a more advanced stage of development so that it is easier for it to get onto free space without being preyed upon.
- **Asexual reproduction: budding or fragmentation** When corals grow large enough, they are able to produce budding: new polyps bud off from their parents and look for new colonies to grow. Fragmentation is similar to budding, but it is caused by the external phenomena such as storm or boats grounding, and usually a larger part of the coral reef is divided from the main part. The divided part off the main part is able to keep growing and grow to a new coral reef. Corals reproduced by both the budding and fragmentation are genetically identical to their parents.

- **Reef longevity and causes of death** Some researches show that corals can live as long as centuries. During their lifetime, they face many hazards. In their larva stage, they are preyed upon by fishes and other predators. Thus, corals produce huge number of larvae to ensure that there is an enough number of polyps can ground on the suitable place and grow up. After then find their colony and grow there, they can still be preyed upon by their natural enemies like starfishes and parrot fishes. Besides, human being's activities can also effect the longevity of corals.

CRO emulates these behaviors to make the solution evolve, so the optimization is conducted. Let Λ be a coral reef composed of $N \times M$ square grid, and each of the grid (i, j) of Λ can hold a coral $\Theta_{(i,j)}$. One coral represents one solution. For initialization, we randomly pick some grids and initialize them by some constructive algorithm, while other grids stays empty. The ratio of initialized grids to empty ones at the beginning is a parameter of CRO, ρ_0, where $0 < \rho_0 < 1$. For each non-empty grid that has a solution in it, we measure the solution by the benchmark function we use. The benchmark function is the objective function of the algorithm selected by us, which is seen as the "health function" of each solution. The health function of solution $\Theta_{(i,j)}$ is denoted as $f(\Theta_{(i,j)}) : \mathscr{I} \to \mathbb{R}$.

After the initialization, CRO starts the formation process, which basically emulates the aforementioned reproduction and competition behaviors to evolve the initialized solution:

- **External reproduction: broadcast Spawning** In a given step of the reproduction phase t, a fraction of coral grids, each of which represents a solution, is selected to reproduce new solutions by broadcast spawning. The ratio of the selected grids to all the $N \times M$ grids is denoted as F_b. Meanwhile, The remaining grids that are not selected, i.e. $1 - F_b$ of the grids reproduce new solutions by brooding.

  After the broadcast spawning corals are determined, we make couples out of them. The couples can be selected from them randomly, or by some sorting methods. Each of these couples crossover and produce a new production. For example, we can merge different parts of the parent solutions to reproduce a new one.

- **Brooding** The fraction $1 - F_b$ of brooding solutions reproduce new solutions through random mutation. Specifically, this is similar to the method we use for looking for neighbors. We can slightly change the solution to make it a new one.

- **Larva setting** Just as the situation that not all the larvae can eventually find a place to grow, not all the solutions reproduced by broadcast spawning or brooding are guaranteed to fill a grid and continue to evolve. Larvae may be preyed upon before they get on a place, or they may fail to find a place to grow. Similarly, a solution tries to randomly set in a grid (i, j), and if the grid is empty, the solution will just set in it. However, if the randomly selected grid has already occupied by other solution, then we compare the health function values of these two solutions, and the solution with better health function value (higher function value is better in maximal optimization problems, and vice versa) will take the grid. In this process, there is another parameter k, as the attempt times by which a larva tries to set in a grid: after k times of failed tries, the larva is preyed upon, meaning that a solution that is not able to fill into a grid is discarded.

- **Asexual reproduction** For simulation of asexual reproduction budding and fragmentation, all the corals on the reef, i.e. the solutions on the square, are first sorted by their health function. Then, a fraction of F_a solutions duplicate themselves and try to set into a grid following the process described above.
- **Depredation** Even if some larvae are successfully set in some places, they can die of either predation or other reasons. For the solutions in the grids, they are first sorted by their health function at the end of each phase t, and a fraction F_d of larva that has the worst health values has a probability P_d to be eliminated. If they are eliminated, the grids taken by them are cleared and become empty grids. These empty grids can be used for holding new solutions reproduced in the next phases.

CRO repeats the above processes so that the solutions keep evolving, until certain termination criteria such as running time limits, or the fitness value target are met. So far, all the parameters, ρ_0, F_b, F_a, F_d, P_d, k, which need to be adjusted in CRO, are illustrated. Table.1 gives a summary of these parameters. DPTP works on CRO to select appropriate values for these parameters to give CRO a good performance.

Appendix B Raw Data of Experiments

Due to limited space, it is hard to append all the raw experiment data here. All the experiment data can be downloaded via the link: `https://drive.google.com/file/d/1Z98qSEyXLseBvIsf_sMmX1w4Zq1uhiy_/view?usp=sharing`.

In the folder, the data of benchmark function values are in Excel files that are named in the format "*[Experiment setup] training_[benchmark function name].xlsx*". For example, a file named "*DPTP training_C23.xlsx*" means it is the data from the experiments using DPTP tuning method measured by benchmark function C23, and "*preset 0 training_C23.xlsx*" means it is the data from the experiments using preset 0 measured by benchmark function C23.

In addition, there are also the data of updating probabilities of different presets updated in the running time named as "*probabilities_[benchmark function name].xlsx*". For example, a file named "*probabilities_C23*" means that it contains the data of preset probabilities in the experiments measured by benchmark function C23.