Transferability in Optimization Via Multidomain Hyperheuristics

Etor Arza, Ekhiñe Irurozki, Josu Ceberio, Aritz Pérez

APPENDIX A
DETAILS ON THE OPTIMIZATION PROBLEMS

In this Appendix, we offer additional details regarding the optimization problems in each problem set. In total, four problem sets were considered, i) and ii) with continuous optimization problems and iii) and iv) with permutation-based optimization problems.

i) twelve continuous optimization problems

These are the twelve continuous optimization problems that were considered in this work, six ($A_1 - A_6$) bowl-shaped and six with many local optima ($A_7 - A_{12}$).

- A_1 Sphere, $\sum_{i=1}^{d} x_i^2$
- A_2 Rotated hyper-ellipsoid, $\sum_{i=1}^{d} \sum_{j=1}^{i} x_j^2$
- A_3 Trid function, $\sum_{i=1}^{d} (x_i - 1)^2 + \sum_{i=1}^{d} x_i \cdot x_{i-1}$
- A_4 Log sphere, $\sum_{i=1}^{d} \log(x_i)^2$
- A_5 Sum of powers, $\sum_{i=1}^{d} |x_i|^i$
- A_6 Sum squares, $\sum_{i=1}^{d} i \cdot x_i^2$
- A_7 Langermann, $\sum_{i=1}^{5} G_i \exp(-\frac{1}{\pi} F_{i,j}) \cos(\pi F_{i,j})$ where $G = \{1, 2, 5, 2, 3\}$, $F_{i,j} = \sum_{j=1}^{i} (x_j - D_{i,j|2})^2$, $j|2$ is the reminder of j divided by 2 and

$$D = \begin{bmatrix} 3 & 5 \\ 5 & 2 \\ 2 & 1 \\ 1 & 4 \\ 7 & 9 \end{bmatrix}$$

- A_8 Schwefel, $\sum_{i=1}^{d} x_i \cdot \sin(\sqrt{|x_i|})$
- A_9 Rastrigin, $10d + \sum_{i=1}^{d} \left[x_i^2 - 10 \cos(2\pi x_i) \right]$
- A_{10} Levy, $\sin^2(\pi w_1) + \sum_{i=1}^{d-1} (w_i - 1)^2 [1 + 10 \sin^2(\pi w_i + 1)] + (w_d - 1)^2 [1 + \sin^2(2\pi w_d)]$, where $w_i = 1 + \frac{x_i - 1}{4}$
- A_{11} Griewank, $\sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}}) + 1$
- A_{12} Ackley, $-20 \exp\left(\frac{1}{5} \sqrt{\frac{1}{d}} \sum_{i=1}^{d} x_i^2\right) - \exp\left(\frac{1}{d} \sum_{i=1}^{d} \cos(2\pi x_i)\right)$

Etor Arza, Ekhiñe Irurozki, and Aritz Pérez are with the Basque Center for Applied Mathematics (BCAM)

Josu Ceberio is with the Department of Computer Science and Artificial Intelligence at the University of the Basque Country (UPV/EHU)

Manuscript submitted December 7, 2022.

All of these functions (except for the log sphere function) were taken from the Virtual Library of Simulation Experiments [1]. Also, note that the version of the Langermann function considered in this paper has been modified to be suitable for any positive dimension d. We used the $d = 20$ versions of these functions (functions of 20 dimensions).

The search space for each problem was set according to the Virtual Library of Simulation Experiments [1]. In addition, each time a problem was loaded, the search space was reduced by a random percentage between 0% and 10%. So for example, the search space of A_1 is $[-5.12, 5.12]^d$ as defined in the Virtual Library of Simulation Experiments [1]. Therefore, the reduced search space would be

$$[-5.12 + (5.12 - (-5.12)) \cdot \delta_1, 5.12 - (5.12 - (-5.12)) \cdot \delta_2] =$$
$$[-5.12 + 10.24 \cdot \delta_1, 5.12 - 10.24 \cdot \delta_2],$$

with δ_1, δ_2 chosen uniformly at random from the interval $[0, 0.1]$. This reduction in the search space slightly moves the position of the optimal solution avoiding any possible bias associated with the movement of the solutions in the direction of the coordinate axes [2] or the center of the search space [3].

In Figures 1 to 12, we show the contour plot of the two dimension versions of these functions, with the search space linearly transformed to $[0, 1]^d$ for easier visualization.

ii) continuous problem generator

In addition to classical continuous optimization algorithms, we considered an optimization problem generator that can produce

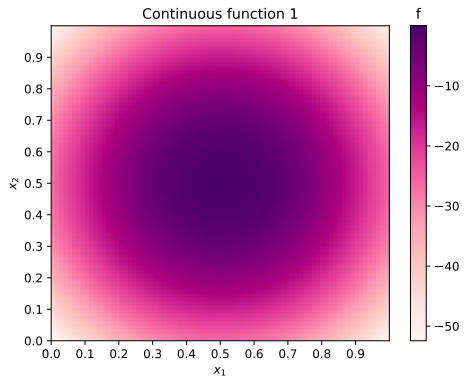


Figure 1: The contour plot of optimization problem A_1.

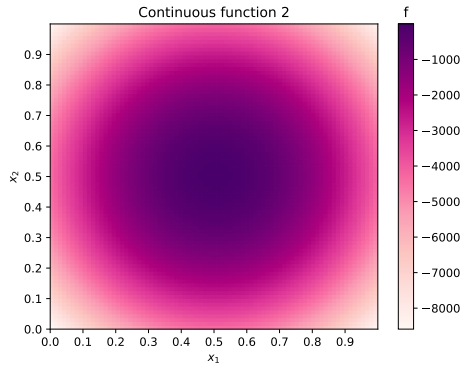


Figure 2: The contour plot of optimization problem A_2.

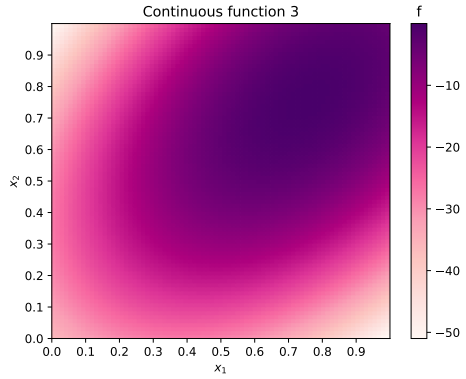


Figure 3: The contour plot of optimization problem A_3.

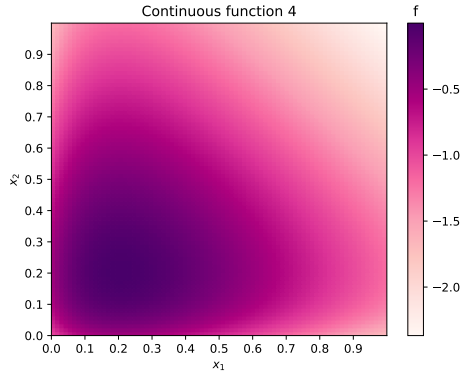


Figure 4: The contour plot of optimization problem A_4.

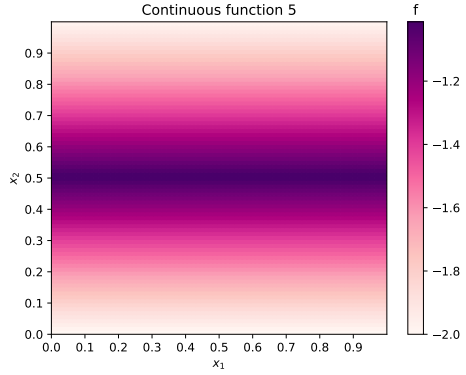


Figure 5: The contour plot of optimization problem A_5.

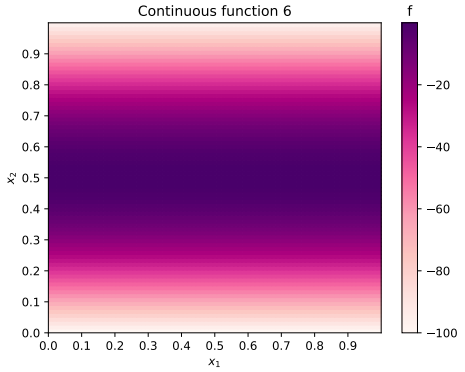


Figure 6: The contour plot of optimization problem A_6.

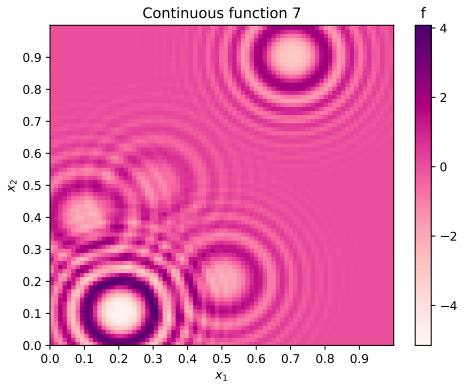


Figure 7: The contour plot of optimization problem A_7.

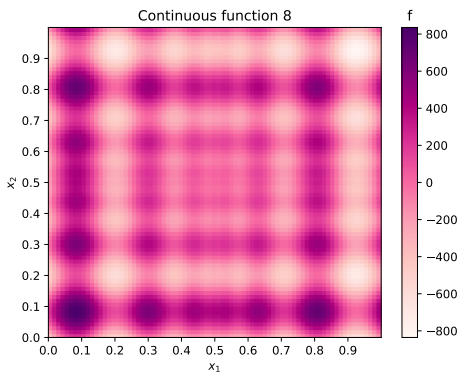


Figure 8: The contour plot of optimization problem A_8.

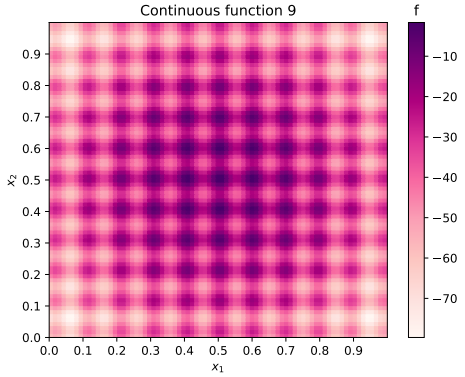


Figure 9: The contour plot of optimization problem A_9.

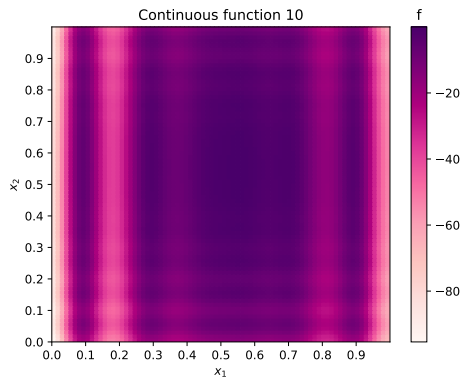


Figure 10: The contour plot of optimization problem A_{10}.

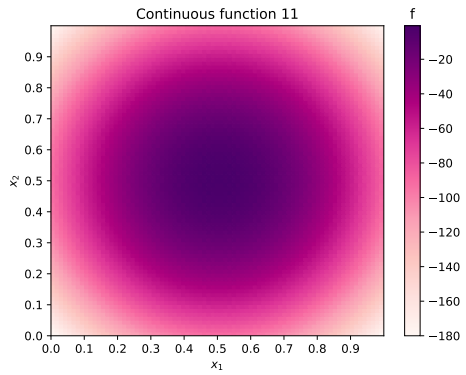


Figure 11: The contour plot of optimization problem A_{11}.

optimization problems with a different number of local optima. Specifically, we considered the "quadratic family" of the optimization problem generator by Rökkönen et al. [4] (see Figure 13 for two examples). Optimization problems with 1, 2, 4, 8, 16, 32, and 64 local optima were generated.

iii) four permutation-based optimization problems

For permutation-based optimization problems, we chose the Traveling Salesman Problem, the Permutation Flowshop Scheduling Problem, the Linear Ordering Problem, and the Quadratic Assignment Problem. Each permutation problem

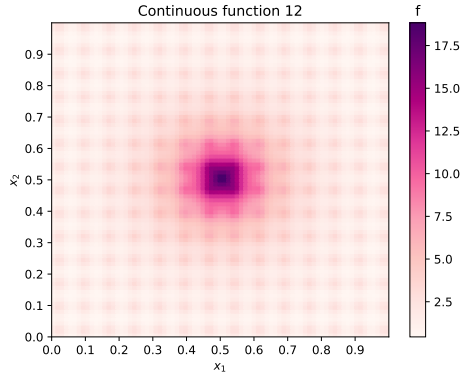
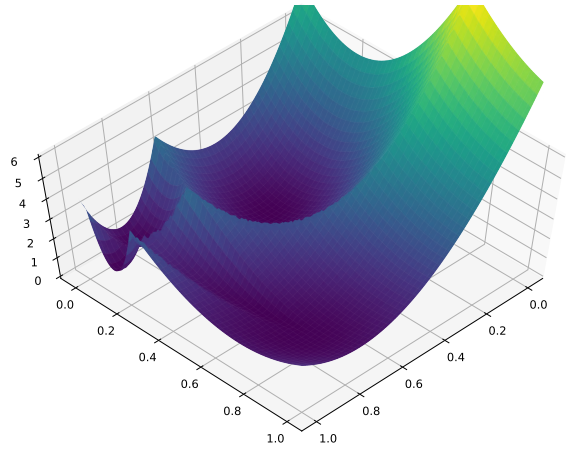
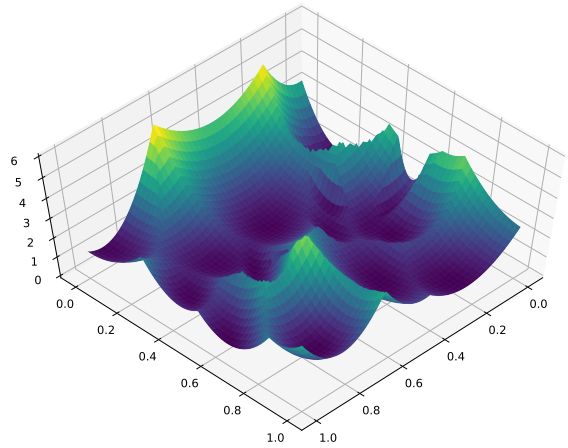


Figure 12: The contour plot of optimization problem A_{12}.



(a) 4 local optima



(b) 16 local optima

Figure 13: Two random minimization problems with a different number of local optima, obtained with the optimization problem generator by Rökkönen et al. [4]. Both of them are two-dimensional problems, with the vertical axis as the objective function.

has many different problem instances, with each instance having an associated objective function with the same structure but different parameters. In the following, we briefly summarize each of the considered permutation problems:

1) Traveling Salesman Problem: Given a set of n cities, the goal of the TSP [5] is to find a path that connects all the cities forming a single cycle while minimizing the length of the path. An instance of the problem is defined by the matrix $\mathbf{D} = [d_{i,j}]_{n\times n}$ containing the distances between any two cities. Given a permutation σ, the objective function value is computed as:

$$f(\sigma) = d_{\sigma(n),\sigma(1)} + \sum_{i=2}^{n} d_{\sigma(i-1),\sigma(i)}$$

2) Permutation Flowshop Scheduling Problem: Given a set of m machines and n tasks, the Permutation Flowshop Scheduling Problem (PFSP) [6] is the problem of optimally ordering the tasks so that a certain criterion is minimized. An instance of PFSP is defined by a matrix $\mathbf{P} = [p_{i,j}]_{n\times m}$, containing the

processing time of task i in machine j. Each of the n tasks has to go through every machine $j \in [m]$ in order, and each machine can only process one task at a time. Unlike the rest of the studied problems, there is more than one possible objective function for this problem. However, we decided to focus on the minimization of the time of completion of all the tasks, $f(\sigma) = c_{\sigma(n),m}$, also known as the makespan. To compute $c_{\sigma(i),j}$, a recursive formula can be used:

$$c_{\sigma(i),j} = \begin{cases} p_{\sigma(i),j} & i = j = 1 \\ p_{\sigma(i),j} + c_{\sigma(i-1),j} & i > 1, j = 1 \\ p_{\sigma(i),j} + c_{\sigma(i),j-1} & i = 1, j > 1 \\ p_{\sigma(i),j} + \max(c_{\sigma(i-1),j}, c_{\sigma(i),j-1}) & i, j > 1 \end{cases}$$

Intuitively, $c_{\sigma(i),j}$ represents the completion time of task $\sigma(i)$ in machine j.

3) Linear Ordering Problem: Given an integer matrix $\mathbf{B} = [b_{i,j}]_{n \times n}$, the goal of the Linear Ordering Problem (LOP) [7] is to find a simultaneous permutation of both rows and columns so that the sum of the values above the diagonal is maximized. The solutions can be encoded as permutations, so that given a permutation σ, the i-th column and row are assigned $\sigma(i)$-th column and row respectively. Formally, the objective function is defined as:

$$f(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} b_{\sigma(i),\sigma(j)}$$

4) Quadratic Assignment Problem: In the QAP [8], we are given a set of n facilities and n locations, along with the distance between any two locations $\mathbf{D} = [d_{i,j}]_{n \times n}$, as well as the workflow between any two facilities $\mathbf{H} = [h_{i,j}]_{n \times n}$. The goal is to find an assignment σ (codified as a permutation) of each of the facilities to one of the locations, such that the objective function $f(\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{i,j} h_{\sigma(i),\sigma(j)}$ is minimized.

Problem instances in permutation problems: Each permutation-based optimization problem has many problem instances [9]. A problem instance is a specific set of parameters that defines the objective function of the optimization problem. The list of considered problem instances is shown in Table I.

Problem instances for problem set iii)

Instance Name	Problem	Size
eil76	TSP	76
rat99	TSP	99
kroA100	TSP	100
kroB100	TSP	100
eil101	TSP	101
pr107	TSP	107
ch130	TSP	130
pr136	TSP	136
tai75e01	QAP	75
tai75e02	QAP	75
tai80a	QAP	80
sko90	QAP	90
sko100a	QAP	100
sko100b	QAP	100
tai100a	QAP	100
tai100b	QAP	100
tai50_5_0	PFSP	(50,5)
tai50_5_1	PFSP	(50,5)
tai100_5_0	PFSP	(100,5)
tai100_5_1	PFSP	(100,5)
tai50_10_0	PFSP	(50,10)
tai50_10_1	PFSP	(50,10)
tai20_20_0	PFSP	(20,20)
tai20_20_1	PFSP	(20,20)
N-be75np_150cut	LOP	75
N-be75oi_150cut	LOP	75
N-stabu3_150cut	LOP	75
N-t65d11xx_150cut	LOP	75
N-t70f11xx_150cut	LOP	75
N-t75n11xx_150cut	LOP	75
N-tiw56r54_150cut	LOP	75
N-tiw56r72_150cut	LOP	75

Table I: Lists the problem instances used in the experimentation on different permutation problems. Note that the LOP instances were originally of size 150 and have been reduced to size 75.

iv) instances of the QAP

In addition to different permutation-based optimization problems, we also considered different problem instance benchmarks within the same problem (the QAP). A problem instance of the QAP is given by two square matrices: the flow matrix **H**, and the distances matrix **D**. The goal of the QAP is to minimize the cost associated with the flow and the distance described in these two matrices. In order to design a diverse experimentation setting, three types of QAP instances were chosen from the QAPLIB (the online encyclopedia of QAP instances [10]), *Taixxa*, *Sko* and *Taixxb*. According to the instance classification of Stützle et al. [11], each of these belongs to a different type of instance class. In the first class, *Taixxa* instances have both matrices **H** and **D** generated uniformly at random with values in the range [1,100]. In the second class, *Sko* instances have the distance matrix **D** based on the Manhattan (L1) distances on a grid. Finally, *Taixxb* instances are real-like generated instances.

Multiple instances from the same type and size are desired to experiment on the transferability and the response in Section III-A. However, it is difficult to find instances of the same type and size for the QAP. To obtain multiple instances of the same size and type, larger instances of the same type were cut, obtaining 5 instances of size 45 of the same type, obtaining a total of 15 instances.

APPENDIX B
NEUROEVOLUTION OF AUGMENTING TOPOLOGIES

In this paper, the neural networks were trained with a reinforcement learning, neuroevolution algorithm. More specifically, neuroevolution of augmenting topologies (NEAT) has been used[1]. Introduced by Stanley et al. [12], NEAT is a genetic algorithm for neural networks that can evolve not only the parameters (weights) of a neural network but also its structure.

NEAT is one of the most influential approaches in the field of neuroevolution to date. In fact, before the introduction of NEAT, most of the neuroevolution techniques without a fixed topology suffered from the competing conventions problem [13]. When two networks have similar functionality but are entirely different, and thus, incompatible structures, we say that these two networks contain competing conventions. NEAT is a revolutionary neuroevolution technique because it defined a new way to efficiently handle the competing conventions problem through a novel speciation mechanism. Specifically, each time NEAT introduces a new structure due to a mutation, this new structure is assigned a unique innovation number. Then, these innovation numbers are used to 1) select individuals that are sufficiently similar to each other for crossover and 2) only perform crossover in the parts of the structure that have the same innovation numbers.

While early studies on the subject suggest that fixed topology networks perform worse than their counterparts [14], there are new promising alternatives that have been found to outperform NEAT even with a fixed topology [15], [16]. New trends in neuroevolution include integrating the Covariance Matrix Adaptation metaheuristic to the framework of neuroevolution [17], designing the architecture of deep neural networks with evolutionary algorithms [18], [19] and HyperNeat [20], in which a two dimension real function is learned that represents the weights of a neural network, achieving a more compact representation that takes into account the position of the neural network links. In this paper, the NEAT algorithm with node and edge removal is used for two reasons: 1) NEAT is a simple yet well-tested neuroevolution algorithm, and 2) the component removal allows the network to adapt its size to the time consumed in the evaluation of the neural network.

To achieve this, one only needs to set a maximum runtime as the stopping criterion for the hyperheuristic solver during the training step and make sure the evaluation time of the solutions is negligible with respect to the evaluation time of the network. The preference towards smaller networks is achieved implicitly: smaller networks are processed faster, and so the hyperheuristic performs more function evaluations in the same amount of time, possibly obtaining better performance.

[1]The source code, along with all the experimentation is available at the GitHub repository https://github.com/EtorArza/TransfHH. The neuroevolution algorithm was taken from the accneat package, with a few minor changes made to it. Accneat is a fork of Stanley et al.'s [12] implementation with some improvements, such as delete mutations and speed improvements, available at https://github.com/sean-dougherty/accneat. The code provided alongside this paper also uses parts of other software projects, see the LICENCE file in the repository for a comprehensive list.

APPENDIX C
DETAILED EXPLANATION OF THE PROPOSED POPULATION-BASED HYPERHEURISTIC

Given a trained controller and a problem instance, Algorithm 1 shows how the hyperheuristic optimizes the given instance guided by the neural network. First, a set of random solutions is initialized and their objective function values are computed (lines 1-2). Next, for each solution σ_i in the population, the information regarding this solution and the state of the optimization process, the feature vector X, is generated by the encoder (line 7). Once X has been obtained, it is fed into the neural network φ. The neural network then outputs the response Y, a real valued vector with values in the interval $(-1, 1)$ (line 8) Next, the decoder interprets the response Y as a modification to be applied to the solution σ_i, obtaining the new solution σ_i' (line 9).

This process is repeated until a stopping criterion is reached, and, upon termination, the best-found solution is reported. It is worth mentioning that this is the general framework of the proposed method, with only the encoder and the decoder having to be specially designed for the search space of the optimization problem to be solved.

Algorithm 1: Multidomain hyperheuristic

Input:
φ: The neural network that guides the hyperheuristic.
f: The objective function of a problem instance.
t_{max}: The maximum number of fitness evaluations.

1 $\bar{\sigma} \leftarrow$ randomly initialize a set of solutions
2 compute the objective function value of the solutions in $\bar{\sigma}$
3 $\bar{\sigma}' \leftarrow$ empty population
4 **while** $t < t_{max}$ **do**
5 **for** $\sigma_i \in \bar{\sigma}$ **do**
6 t $\leftarrow 1 + t$
7 $X \leftarrow$ the encoder collects information about σ_i and the optimization state into a real vector
8 $Y \leftarrow \varphi(X)$
9 $\sigma_i' \leftarrow \text{decode}(\sigma_i, Y)$ `// modify` σ_i
10 compute the objective value of σ_i'
11 add σ_i' to $\bar{\sigma}'$
12 $\bar{\sigma} \leftarrow \bar{\sigma}'$
13 $\bar{\sigma}' \leftarrow$ empty population
14 $\sigma_{best} \leftarrow$ update best solution
15 **return** σ_{best}

A. Encoder, generating the feature vector X

Note that the above algorithm is general and is applicable to many optimization problems, regardless of the search space of the problem. The encoder and the decoder are the problem-specific parts of the hyperheuristic algorithm. The encoder is in charge of encoding the state of the optimization into a real valued vector: the feature vector X. In the following, we explain the two encoders used in this paper, one for continuous problems and one for permutation problems.

1) Encoder for continuous problems: The encoder for continuous problems considered in this paper produces a real valued feature vector X of size 10 $X = (x_1, x_2, ..., x_{10})$, with values in the interval $(0, 1)$. The feature vector is computed for each solution σ_i to be modified. In the following, we briefly explain all the information variables x_j obtained for each solution σ_i in the set of solutions $\bar{\sigma}$ during the encoding step.

- x_1, x_2 indicate the absolute L_1 distance from the current solution $sigma_i$ to the best solution in the population and the average solution respectively.
- x_3, x_4, x_5, describe the relative distance from the current solution to the closest, best, or average solution respectively. First, the absolute distances are computed for every solution in the population and then the relative distances are set in the interval $(0, 1)$ as a relative ranking of their absolute counterparts.
- x_6 is the proportion of the computation budget used so far.
- x_7 is proportional to the relative ranking of solutions in the population σ_i with respect to the rest of the solutions. Since the solutions in the population are sorted according to their objective value before X is computed, a value of $\frac{i}{q}$ is assigned to x_4, where q is the population size and i is the relative ranking of the permutation σ_i with respect to the rest of the solutions in the population.
- x_8 is 1 if σ_i improved its best found solution and 0 otherwise.
- x_9 is 1 if σ_i is the best-found solution so far and 0 otherwise.
- x_{10} is a random number in the interval $(0, 1)$.

2) Encoder for permutation problems: The encoder for permutation problems considered in this paper produces a real valued feature vector X of size 8 $X = (x_1, x_2, ..., x_8)$, with values in the interval $(0, 1)$. The feature vector is computed for each permutation σ_i to be modified. In the following, we briefly explain all the information variables x_j obtained for each solution σ_i in the set of solutions $\bar{\sigma}$ during the encoding step.

- x_1, x_2 and x_3 indicate whether σ_i is a local optimum (1) or not (0), for each of the three operators considered in this paper: adjacent swap, exchange and insert. Given a permutation $\sigma = (\sigma(1), ..., \sigma(n))$, the swap operator exchanges two adjacent items $\sigma(s)$ and $\sigma(s+1)$ for any given $s \in \{1, ..., n-1\}$. The exchange operator, in similar way, exchanges items $\sigma(s), \sigma(k)$ for a given pair $s, k \in \{1, ..., n\}$. Finally, the insert operator inserts item $\sigma(s)$ at position k.
- x_4 is proportional to the relative ranking of solutions in the population σ_i with respect to the rest of the permutations. Since the solutions in the population are sorted according to their objective value before X is computed, a value of $\frac{i}{q}$ is assigned to x_4, where q is the population size and i is the relative ranking of the permutation σ_i with respect to the rest of the permutations in the population.
- x_5, is set according to the computational budget spent so far.
- x_6 is proportional to the relative ranking of the differences in the objective value with respect to the previous

permutation in the population. In other words, a value proportional to the ranking of $f(\sigma_{i-1}) - f(\sigma_i)$, where σ_i is the permutation to be moved, and σ_{i-1} is the permutation that is next to σ_i in terms of relative ranking.

- x_7 and x_8 are proportional to the Hamming and Spearman's footrule [21] distances from σ to a median permutation σ_0 that satisfies $\sigma_0 = \text{argmin}_\sigma \sum_{i=1}^{q} d(\sigma, \sigma_i)$, where d is the Hamming and the Kendall distance [22] respectively, and σ_i is the permutation to be moved. The Hamming median permutation can be obtained by solving a linear assignment problem [23]. In the case of the Kendall distance, the median permutation is approximated using the Borda algorithm [24], [25].

B. Decoder: modifying solution σ_i given the response Y

1) Decoder for continuous problems: The decoder for continuous problems is based on the well-known particle swarm optimization algorithm [26]. In this algorithm, each solution is updated via a linear combination of three vectors. Specifically, each solution σ_i is moved towards the best solution so far (denoted as σ_{best}) and the best solution visited by σ_i ($\sigma_{i,\text{best}}$). The move made in the last iteration is also used as a reference movement for the next iteration. These three moves are represented by three vectors, which are then added to σ_i. Choosing the weights to use in this linear combination is still an open problem [27].

The decoder proposed in this paper interprets a response $Y = (y_1, y_2, y_3)$ of size 3. Specifically, the modified solution σ_i' is obtained as

$$\sigma_i' = \sigma_i + \Delta_t,$$

and

$$\Delta_t = y_1 \cdot (\sigma_i - \sigma_{\text{best}}) + y_2 \cdot (\sigma_i - \sigma_{i,\text{best}}) + y_3 \cdot (\Delta_{t-1}),$$

where σ_{best} is the best solution found, $\sigma_{i,\text{best}}$ is the best solution visited by σ_i and Δ_{t-1} is the Δ_t of the previous iteration[2].

2) Decoder for permutation problems: The decoder for permutation problems considered in this paper uses algorithmic components that modify each solution at each iteration. The algorithmic components proposed in our implementation are inspired by other popular approaches such as simulated annealing [28], local search, and variable neighborhood search [29] among others. It is important to note that the algorithmic components by themselves are not optimization algorithms. Instead, the decoder combines and parameterizes these algorithmic components to modify each solution at each iteration.

Since the neural network operates in the space of real valued vectors, the control of algorithmic components—and hence, how to modify each solution—is represented in a real valued vector Y. The decoder modifies σ_i by interpreting Y, for each solution σ_i. The response vector Y is a real valued vector of size 11 with values in the interval (-1,1). Algorithm 2 shows how the decoder combines and controls heuristic components based on the output of the neural network Y to modify permutation σ_i into a new permutation σ_i'. The first heuristic component, associated with the value y_6 in the response, determines if solution σ_i should be randomly reinitialized or not (lines 1-3). Next, the operator to be used is specified by the argmax of (y_2, y_3, y_4), each value representing swap, exchange, and insert operators, respectively (line 4). Then, y_1 chooses between three heuristic options: if $|y_1| < 0.25$ is satisfied, then the solution remains unaltered, $\sigma_i' = \sigma_i$ (lines 12-13). If $y_1 < -0.25$, then a local search[3]iteration is applied with the chosen operator(lines 14-15). If $|y_1| < 0.25$ is satisfied, then the solution remains unaltered, $\sigma_i' = \sigma_i$. Finally, if $y_1 > 0.25$ then a modification is going to be applied without local search (move step).

In the case that $y_1 > 0.25$ is satisfied (move step), the permutation σ_i is moved towards or away from another reference permutation. The reference permutation is specified by the last five values of the response $(y_7, y_8, y_9, y_{10}, y_{11})$, each value associated with a different reference permutation as specified in Table II. Specifically, the reference permutation will be chosen randomly considering probabilities proportional to the absolute value of the reference coefficients (lines 17-18). The direction of the modification is chosen according to the sign of the corresponding reference coefficient, with a positive value representing a modification towards the reference, and a negative value a modification away from the reference (line 19). Algorithm 3 shows the pseudo-code of the move step procedure considered in the proposed decoder. The modification towards a reference (lines 2-3 of Algorithm 3) tries to modify the permutation into another permutation that is one unit of distance nearer the reference [30], $d(\sigma_i, \sigma_{ref}) = d(\sigma', \sigma_{ref}) + 1$. The distance is defined for each operator as the minimum number of applications of that operator to transform one permutation into the other permutation. A modification away from the reference is defined similarly (line 5 of Algorithm 3), but choosing the parameters of the operator that increases the distance between the reference permutation and the modified permutation by one unit $d(\sigma_i, \sigma_{ref}) = d(\sigma', \sigma_{ref}) - 1$.

The coefficient y_5 coefficient expands the functionality of the decoder shown in Algorithm 2. Through this coefficient, the decoder has a simulated annealing [31] like behavior in the case of $y_1 > 0.25$ (move step, line 20 of Algorithm 2). Specifically, y_5 is proportional to the probability of accepting a solution that is worse than σ_i (lines 14-16 of Algorithm 3).

Once all the solutions σ_i have been modified into a new solution σ' for all the solutions σ_i in the population, the next iteration is executed. The process is repeated until a stopping criterion is reached.

[2]An iteration involves updating all solutions σ_i in the population.

[3]The local search procedure is a best-first procedure, thus the neighborhood of σ_i defined by the chosen operator is explored until a better solution is found or all the neighborhood has been visited.

Table II: Reference permutations available to the decoder

Reference index	Reference permutation (σ_{ref})
$ref = 7$	Hamming median permutation. Already computed when generating the feature vector element x_7.
$ref = 8$	Kendall median permutation. Already computed when generating the feature vector element x_8.
$ref = 9$	Best solution that σ_i has visited in past iterations.
$ref = 10$	Best known solution.
$ref = 11$	σ_{i-1}, or equivalently, the permutation that is closest in objective value to the solution being modified.

Algorithm 2: decode(σ_i,Y)

Input:
$Y = (y_1, ..., y_{10})$: Response for solution σ_i.
σ_i: The permutation to be modified.

if $y_8 > 0.25$ **then**
  $\sigma' \leftarrow$ random permutation
end
selected_operator $\leftarrow$ argmax(y_2, y_3, y_4) `// Corresponding to the swap, exchange and insert operators respectively`
if $|y_1| < 0.25$ **then**
  $\sigma' \leftarrow \sigma_i$ `// No changes to solution` σ_i
else if $y_1 < -0.25$ **then**
  $\sigma' \leftarrow$ local_search_iteration(σ, operator)
else
  $ref \leftarrow$ choose $j \in \{9, ..., 13\}$ with prob. proportional to $|y_j|$
  $\sigma_{ref} \leftarrow$ select reference permutation as shown in Table II
  movement_direction $\leftarrow$ sign(y_{ref})
  prob_accept_worse_solution $\leftarrow \frac{y_4+1}{2}$
  $\sigma' \leftarrow$ move(σ_i, σ_{ref}, movement_direction, operator, prob_accept_worse_solution) `// see Algorithm 3`
end

Algorithm 3: move(σ_i, σ_{ref}, movement_direction, selected_operator, prob_accept_worse_solution)

Input:
σ_i: The permutation to be moved or modified.
σ_{ref}: The reference permutation.
movement_direction: Direction of the movement to be applied.
selected_operator: Operator of the applied movement.
prob_accept_worse_solution: Probability of accepting a worse solution.
Output:
σ': modified solution.

if movement_direction > 0 **then**
  create a minimum length path from σ_i to σ_{ref}, defined as the minimum sequence of applications of the operator "selected_operator" to solution σ_i that are required to obtain solution σ_{ref} (Schiavinotto et al. [30]).
  operator_params $\leftarrow$ parameters of the first operator in the minimum length path
else
  operator_params $\leftarrow$ parameters of "selected_operator" that, if applied to σ_i, would increase the distance between σ_i and σ_{ref} by one in terms of the minimum required applications of "selected_operator" to transform one permutation into the other.
end
$\sigma_{cand} \leftarrow$ create candidate solution by applying selected_operator with parameters operator_params to solution σ_i
if objective value σ_i is better than objective value σ_{cand} **then**
  with probability (1 - prob_accept_worse_solution)
    $\sigma_{cand} \leftarrow \sigma_i$
end
end
$\sigma' \leftarrow \sigma_{cand}$

APPENDIX D
TRANSFERABILITY

In this appendix, we formally define the transferability and give additional details on how to generate the embedding based on the performance of the hyperheuristic. For additional details, we refer the interested reader to the GitHub repository with the source code.

A. Transferability matrix

We begin by introducing some notation to formally define this concept.

Notation 1: Controller trained in a problem
Let A_1 be an optimization problem and φ a controller. We denote that φ was trained in problem A_1 as φ_1.

Definition 1: Performance of a controller
Let A_1, A_2 be two optimization problems and φ_1 a controller trained in optimization problem A_1. We define the performance of φ_1 on A_2, as the result of optimizing A_2 with the proposed hyperheuristic algorithm, using φ_1 as the controller and we denote it as $\varphi_1(A_2)$.

Definition 2: Transferability
Let $A_1, ..., A_k$ be k maximization problems[4] and φ_i, φ_j two controllers trained in the problems A_i, A_j respectively. We define the transferability from problem A_i to problem A_j, $T(A_i \to A_j)$, as the normalized rank in performance that the hyperheuristic guided by controller φ_1 has in problem A_2. Formally,

$$T(A_i \to A_j) = \frac{r_{i,j}}{k-1}$$

where $r_{i,j}$ is the ranking (best to worst) of $\varphi_i(A_j)$ in $\{\varphi_t(A_j)\}_{t=1,...,k}$. In other words, $T(A_i \to A_j)$ is assigned a rank proportional to how well controller φ_i does on problem A_j with respect to the performance of the rest of the controllers in the same problem. The transferability $T(A_i \to A_j)$ is 0 if φ_i is the controller with the best performance in A_j and 1 if it is the worst performing.

Notation 2: Transferability Matrix
Let $A_1, ..., A_k$ be k maximization problems. We denote the transferability matrix as T where each element $T_{i,j}$ is $T(A_i \to A_j)$.

B. Clustermap

The transferability matrix can be further improved by sorting the problems in the heatmap. We can apply clustermap [32] (heatmap clustering) to sort the columns and the rows together,—by swapping the indexes i of the problems— such that adjacent lines and rows are more similar to each other. Note that with this change, the positions in the clustermap no longer match the problem indices.

[4]We assume that all problems considered are maximization problems. Specifically, minimization problems have been transformed into maximization problems by redefining the problem as a maximization of the objective function when multiplied by -1.

To generate the clustermap [32], we first need to choose a distance and then generate a dendrogram. We chose the "Nearest Point Algorithm" [33]–[35] that defines the distance between a cluster and the point as the minimum distance between the point and any point in the cluster. Note that the clustermap is the same as the transferability matrix, but with the indices changed, and this makes it easier to visualize similar rows and columns.

The generated clustermap is an embedding of a set of problems into a $k \times k$ real matrix. We denote this embedding as *transferability heatmap*.

C. Repeated measurements and noise

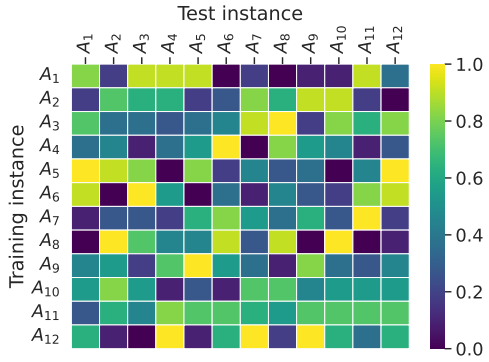
It is important to measure the transferability several times and average the results. The reason is that it is very likely that stochastic optimization algorithms will get different scores each time they are tested. Without this repetition step, it is possible to observe transferability, when in reality, is just due to the random nature of the optimization algorithms. This is better understood with the following example.

Let us assume that the transferability of a random search algorithm is measured in problem set i). Since the random search algorithm has no learning phase, the score will be random each time. When we measure the transferability only once, we get what is shown in Figure 14a. Notice that $T_{3,2} = 0$. From that, it would be erroneous to conclude that the problem A_3 is good at training the random search algorithm for problem A_2. In fact, $T_{3,2} = 0$ has the same probability as $T_{i,2} = 0$, for all $i = 1, ..., 12$.

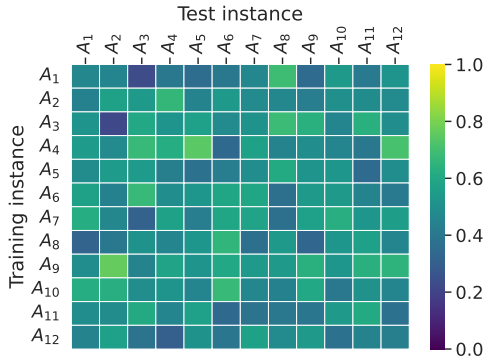
On the other hand, if we average 10 repeated measures of the transferability, we obtain the results shown in Figure 14b. In this figure, we see that most of the values are near 0.5. The interpretation is that there was no transferability: it does not matter which instance is used during the training step. In short, several repeated measurements overcome the limitation of observing transferability in stochastic algorithms when there should be none. For this reason, in the rest of the paper, we averaged 10 measurements of the transferability.

1) Related work: Hong et al. [36] proposed measuring the transferability as the average objective value of the hyperheuristic when training and testing two *problem classes*. Their experimental approach is different, as they study the *transferability among problem classes* instead of *transferability among problems*. They use *problem class* to refer to an infinite set of optimization problems parameterized by one or more real values (for example, $f(x) = ax^2$, with the parameter $a \in [1, 2]$).

Our experimental setup on transferability improves the methodology of Hong et al. [36] in three key aspects. Firstly, by defining the transferability with ranks instead of objective values, we can compare the values across different problems. When objective values are used directly, the differences are



(a) One measurement.



(b) Ten measurements.

Figure 14: The transferability of a random search algorithm with one measurement (a) and with ten averaged measurements (b).

not comparable[5].

Secondly, by repeating several measurements of the transferability and computing the average ranks, we can distinguish between noise and the actual difference in performance (this is not possible with average objective values). Additional details on *repeated measurements* are given in Appendix D-C.

Thirdly, Hong et al. [36] define function classes [37] and compare different function classes among each other in a table. In our work, we take existing optimization problems in the literature and we analyze the transferability of the hyperheuristic among them. This allows us to generate a visualization of optimization problems that is correlated with the properties of the problems.

[5]For example, let us assume we have the optimization problems $f_1(x) = x^2$ and $f_2(x) = 10^{10} \cdot x^2$ both defined in the interval $[0, 1]$. Given two solutions, if the difference in objective value is 0.5, in problem f_1 it would be considered a bigger difference than in problem f_2.

APPENDIX E
RESPONSE

In the previous section, we proposed an embedding for a set of optimization problems via the performance of the hyperheuristic. In this section, we propose another embedding by focusing instead on the behavior of the hyperheuristic.

1) Defining the response: The behavior of the hyperheuristic, *how* it performs the optimization, is determined by every response Y (the output) of the neural network, generated during the optimization: the decoder modifies each solution according to its corresponding response Y. Therefore, it makes sense to summarize the behavior of the hyperheuristic in an optimization problem by averaging every response Y that the neural network produces when solving the problem.

Definition 3: Response
Let A_1, A_2 be two problems. We define the response from A_1 to A_2, $R(A_1 \rightarrow A_2)$, as the average of all response Y generated[6]when solving problem A_2, with a neural network obtained by training the hyperheuristic in problem A_1. The response can be interpreted as a summary of the behavior of the hyperheuristic when solving problem A_2 with a controller trained in problem A_1.

The response is a random variable in the sense that two independent computations of the response will not produce the same result. the response depends on two factors that cannot be fixed or predicted, the first one a) is related to the training stage, and the second b), is related to the testing stage. Firstly a), two executions of the neuroevolution algorithm on the same problem will (almost) never produce two identical neural networks. Secondly b), due to the probabilistic nature of the hyperheuristic, two executions on the same problem with the same neural network are likely to diverge into a different final solution. This is in part due to the random initialization of the population. Luckily, variability regarding b) is easy to overcome by averaging the response of many executions of the hyperheuristic in the same test problem.

The variability introduced during training a) is determined by the random seed used to train the neural network. Therefore, each observation of the response can be notated as $R(i, j, s)$ where i denotes the training problem, j denotes the test problem, and s denotes the random seed. An observation of the response, hence, averages several measurements of the response in the same test problem.

To find out what determines the variability in response, we measured the average distance between two responses

$$||R(i_1, j_1, s_1) - R(i_2, j_2, s_2)||_1$$

in these four cases:

[6]The response-hyperheuristic behavior mapping is not always injective, and depends on the implementation of the decoder. The decoder for permutation problems proposed in this paper is not injective. In other words: two controllers with different responses may produce the same behavior in the decoder. A possible solution is to transform the output so that injectivity is preserved. These transformations are simple functions, such as the indicator function $\chi_{(0.25,1]}$. For more details on the exact functions used to reduce the duplicity, we refer the interested reader to the source code provided.

Average L1 distance between two responses

Case	1)	2)	3)	4)
Train problem	same	same	different	different
Test problem	different	different	same	different
Training seed	same	different		
Problem set				
i)	0.050	0.436	0.516	0.520
ii)	0.004	0.371	0.367	0.370
iii)	0.174	2.718	2.832	2.812
iv)	0.181	3.172	3.122	3.182

Table III: Average distance between two responses trained in the same problem (with or without the same seed) and tested in the same problem.

1) $i_1 = i_2$, $j_1 \neq j_2$ and $s_1 = s_2$
2) $i_1 = i_2$, $j_1 \neq j_2$ and $s_1 \neq s_2$
3) $i_1 \neq i_2$, $j_1 = j_2$ and $s_1 \neq s_2$
4) $i_1 \neq i_2$, $j_1 \neq j_2$ and $s_1 \neq s_2$

for the four problem sets i) - iv) defined in Section IV-A.

As seen in Table III case 1), the distance is really small when the same training problem & training seed is used (even if tested in different problems). In addition, the distance is much larger in the rest of the cases. The interpretation is that, given the training problem i and the random seed s, the response of the hyperheuristic is determined (disregarding some small variability associated with the test instance j).

Consequently, it makes sense to model the response as a function $R(i, s) = k^{-1} \sum_{j=1}^{k} R(i, j, s)$. For each problem i, we have 10 samples of the behavior of the hyperheuristic $R(i, s)$ in that problem (one for each seed s). Notice that the response modeled like this depends on the training problem. Now we want to see if there is any correlation between the behavior of the hyperheuristic and the optimization problem used to train it, regardless of the training seed.

A naive approach to achieve this would be to average all the responses in one problem $R(i) = 10^{-1} \sum_{s=1}^{10} R(i, s)$ and embed every $R(i)$ into $\mathbb{R}^2$ with principal component analysis [38]. However, this can hide the differences in behavior of the hyperheuristics. A possible explanation is that $R(i_1, s_1)$ and $R(i_1, s_2)$ are almost as different from each other as $R(i_2, s_1)$ and $R(i_1, s_2)$ (see Table III), with $i_1 \neq i_2$ and $s_1 \neq s_2$.

Linear Discriminant Analysis [39], [40] (LDA) overcomes this limitation. First, we fit the LDA with every $R(i, s)$ and set i as the class. The LDA will try to maximize the distance between every two $R(i, s)$ with different i and minimize the distance between every two $R(i, s)$ with the same i. Once we have fitted the LDA, we project every $R(i)$ in the embedded space.

Notice that the LDA takes into account which training problem was used to produce each $R(i, s)$, but it gets no information about the properties of the training problem i: every problem is equal for the LDA. This is an important point because it means that a projection obtained from the LDA does not require this information, and thus, it can be used in the clustering of unlabeled optimization problems or even the classification

of previously unseen optimization problems.

REFERENCES

[1] S. Surjanovic and D. Bingham, "Virtual library of simulation experiments: Test functions and datasets," Retrieved November 8, 2021, from http://www.sfu.ca/~ssurjano.

[2] S. Janson and M. Middendorf, "On Trajectories of Particles in PSO," in *2007 IEEE Swarm Intelligence Symposium*. Honolulu, HI, USA: IEEE, Apr. 2007, pp. 150–155.

[3] C. K. Monson and K. D. Seppi, "Exposing origin-seeking bias in PSO," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 241–248.

[4] J. Rönkkönen, X. Li, V. Kyrki, and J. Lampinen, "A Generator for Multimodal Test Functions with Multiple Global Optima," in *Simulated Evolution and Learning*, X. Li, M. Kirley, M. Zhang, D. Green, V. Ciesielski, H. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. C. Tan, J. Branke, and Y. Shi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5361, pp. 239–248.

[5] D. E. Goldberg and R. Lingle, Jr., "Alleles, Loci and the Traveling Salesman Problem," in *Proceedings of the 1st International Conference on Genetic Algorithms*. L. Erlbaum Associates Inc., 1985, pp. 154–159.

[6] J. N. Gupta and E. F. Stafford, "Flowshop scheduling research after five decades," *European Journal of Operational Research*, vol. 169, no. 3, pp. 699–711, Mar. 2006.

[7] T. Schiavinotto and T. Stützle, "The Linear Ordering Problem: Instances, Search Space Analysis and Algorithms," *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 4, pp. 367–402, 2004.

[8] T. C. Koopmans and M. Beckmann, "Assignment Problems and the Location of Economic Activities," *Econometrica*, vol. 25, no. 1, p. 53, Jan. 1957.

[9] A. Elorza, L. Hernando, and J. A. Lozano, "Taxonomization of combinatorial optimization problems in fourier space," *arXiv preprint arXiv:1905.10852*, 2019.

[10] R. E. Burkard, S. E. Karisch, and F. Rendl, "QAPLIB–a quadratic assignment problem library," *Journal of Global optimization*, vol. 10, no. 4, pp. 391–403, 1997.

[11] T. Stützle, "Iterated local search for the quadratic assignment problem," *European Journal of Operational Research*, vol. 174, no. 3, pp. 1519–1539, Nov. 2006.

[12] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, Jun. 2002.

[13] J. D. Schaffer, D. Whitley, and L. J. Eshelman, "Combinations of genetic algorithms and neural networks: A survey of the state of the art," in *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE, 1992, pp. 1–37.

[14] K. O. Stanley and R. Miikkulainen, "Competitive Coevolution through Evolutionary Complexification," *Journal of Artificial Intelligence Research*, vol. 21, pp. 63–100, Feb. 2004.

[15] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *Journal of Machine Learning Research*, vol. 9, no. May, pp. 937–965, 2008.

[16] P. Pagliuca, N. Milano, and S. Nolfi, "Maximizing adaptive power in neuroevolution," in *PloS One*, 2018.

[17] H. Moriguchi and S. Honiden, "CMA-TWEANN: Efficient optimization of neural networks via self-adaptation and seamless augmentation," in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference - GECCO '12*. Philadelphia, Pennsylvania, USA: ACM Press, 2012, p. 903.

[22] M. A. Fligner and J. S. Verducci, "Distance Based Ranking Models," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 48, no. 3, pp. 359–369, Jul. 1986.

[18] Y. Sun, G. G. Yen, and Z. Yi, "Evolving Unsupervised Deep Neural Networks for Learning Meaningful Representations," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 1, pp. 89–103, Feb. 2019.

[19] F. E. Fernandes Junior and G. G. Yen, "Particle swarm optimization of deep neural networks architectures for Image classification," *Swarm and Evolutionary Computation*, p. S2210650218309246, Jun. 2019.

[20] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, Apr. 2009.

[21] P. Diaconis and R. L. Graham, "Spearman's footrule as a measure of disarray," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 2, pp. 262–268, 1977.

[23] E. Irurozki, B. Calvo, and J. A. Lozano, "Mallows and generalized Mallows model for matchings," *Bernoulli*, vol. 25, no. 2, pp. 1160–1188, May 2019.

[24] W. D. Cook and L. M. Seiford, "On the Borda-Kendall Consensus Method for Priority Ranking Problems," *Management Science*, vol. 28, no. 6, pp. 621–637, Jun. 1982.

[25] J. C. de Borda, "Memoire sur les Elections au Scrutin," *Histoire de l'Academie Royale des Sciences, Paris*, 1781.

[26] J. Kennedy and R. Eberhart, "Particle swarm optimization (PSO)," in *Proc. IEEE International Conference on Neural Networks, Perth, Australia*, 1995, pp. 1942–1948.

[27] A. Engelbrecht and C. Cleghorn, "Recent advances in particle swarm optimization analysis and understanding," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 747–774.

[28] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers operations research*, vol. 13, no. 5, pp. 533–549, 1986.

[29] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & operations research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[30] T. Schiavinotto and T. Stützle, "A review of metrics on permutations for search landscape analysis," *Computers & operations research*, vol. 34, no. 10, pp. 3143–3153, 2007.

[31] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing*. Dordrecht: Springer Netherlands, 1987, pp. 7–15.

[32] M. Waskom, "Seaborn.clustermap — seaborn 0.11.2 documentation."

[33] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[34] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: An overview," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 86–97, 2012.

[35] The SciPy community, "Scipy.cluster.hierarchy.linkage — SciPy v1.8.1 Manual."

[36] L. Hong, J. H. Drake, J. R. Woodward, and E. Özcan, "A hyper-heuristic approach to automated generation of mutation operators for evolutionary programming," *Applied Soft Computing*, vol. 62, pp. 162–175, Jan. 2018.

[37] J. Woodward, S. Martin, and J. Swan, "Benchmarks that matter for genetic programming," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 1397–1404.

[38] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1-3, pp. 37–52, Aug. 1987.

[39] P. Singh, "Dimensionality reduction approaches," https://towardsdatascience.com/dimensionality-reduction-approaches-8547c4c44334, 2020-08-18, 2020.

[40] C. R. Rao, "The utilization of multiple measurements in problems of biological classification," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 10, no. 2, pp. 159–203, 1948.