Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Marek Šrank**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Portfolio Algorithms for Combinatorial Optimization**

Guidelines:

1. Learn about the principles of portfolio algorithms.
2. Choose a set of benchmark problems from the area of combinatorial optimization.
3. Choose at least 2 portfolio algorithms with online schedule creation and compare their efficiency with restarted algorithms and with portfolios using known schedules.
4. Evaluate the contributions and potential of portfolio algorithms.

Bibliography/Sources:

1. György and L. Kocsis (2011) "Efficient Multi-Start Strategies for Local Search Algorithms", Volume 41, pages 407-444
2. Shiu Yin Yuen, Chi Kin Chow, Xin Zhang (2013) "Which Algorithm Should I Choose at any Point of the Search: An Evolutionary Portfolio Approach", GECCO 2013
3. Peng, F., Tang, K., Chen, G., and Yao, X. 2010. Population-based algorithm portfolios for numerical optimization, IEEE Trans. Evol. Comput. 14(5), 782-800.

Diploma Thesis Supervisor: Ing. Petr Pošík, Ph.D.

Valid until the end of the summer semester of academic year 2014/2015

L.S.

Prague, March 3, 2014

Czech Technical University in Prague
Faculty of Electrical Engineering

diploma thesis

Portfolio algorithms for combinatorial optimization

Bc. Marek Šrank

Department of Computer Science
advisor: Ing. Petr Pošík, Ph.D.

January 2015

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on ....

Marek Šrank

Abstrakt

Táto diplomová práca skúma vybrané portfólio algoritmy pre kombinatorickú optimalizáciu. Cieľom je zistiť, či tieto algoritmy poskytujú efektívnu alternatívu k najčastejšie používaným reštartovacím stratégiám. Porovnáva niekoľko verzií MetaMax algoritmu, s fixnou ako aj neobmedzenou veľkosťou portfólia a algoritmus MultiEA, ktoré používajú online plánovanie. Sú tiež navrhnuté dve modifikované varianty MultiEA algoritmu. Do porovnania je zahrnutý aj zástupca bandit stratégií, algoritmus Epsilon-greedy, ktorého plán je vopred známy a dve reštartovacie stratégie. Všetky tieto algoritmy sú testované na skupine úloh kombinatorickej optimalizácie a je vyhodnotená ich rýchlosť konvergencie a tiež schopnosť nájsť riešenie.

Abstract

This diploma thesis studies selected portfolio algorithms for combinatorial optimization. The goal is to find out whether these algorithms provide an efficient alternative to the most commonly used restarting strategies. It compares several versions of MetaMax algorithm with fixed and also unbounded portfolio sizes and the MultiEA algorithm, that both use an online schedule creation. Two modified variants of MultiEA algorithm are also proposed. A representative of bandit strategies, Epsilon-greedy algorithm, whose schedule is known in advance, and two different restarting strategies are included in comparison as well. All these algorithms are tested on the set of combinatorial optimization tasks and their convergence rate as well as the ability to find a solution is evaluated.

Acknowledgements

I would like to express my thanks to Mr. Petr Pošík, my supervisor, for his guidance and help. My thanks also go to my family and friends for supporting me.

This diploma thesis was typeset in LaTeX using the FELthesis template by Vít Zýka.

Computational resources were provided by the MetaCentrum under the program LM2010005 and the CERIT-SC under the program Centre CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, Reg. no. CZ.1.05/3.2.00/08.0144.

Contents

1 Optimization problems and local search algorithms 2
1.1 Local optimization . . . 2
1.2 Travelling salesman problem . . . 2
1.3 Centroid-based clustering . . . 4
1.4 Warehouse Location Problem . . . 5

2 Evaluation of algorithm quality 7
2.1 Fitness function . . . 7
2.2 ECDF of running times . . . 8
2.3 Aggregated performance index . . . 8

3 Meta-strategies 10
3.1 Restarting strategies . . . 10
3.2 Bandit strategies . . . 11
3.2.1 Epsilon-first . . . 11
3.2.2 Epsilon-greedy . . . 11
3.2.3 Epsilon-decreasing . . . 12
3.3 Portfolio algorithms . . . 12
3.3.1 MetaMax . . . 12
MetaMax(k) . . . 13
MetaMax(∞) . . . 15
MetaMax . . . 17
3.3.2 MultiEA . . . 17

4 Experiments 21
4.1 Optimization problems and their instances . . . 21
4.2 Meta-strategies and their configuration . . . 22
4.3 Results . . . 24
4.3.1 Travelling salesman problem . . . 25
4.3.2 K-means clustering . . . 29
4.3.3 Single-swap clustering . . . 33
4.3.4 Warehouse location problem . . . 37

5 Conclusion 41

Appendices

Bibliography 44

A API by strategy and parameter settings 45
A.1 Travelling salesman problem . . . 45
A.2 K-means clustering . . . 48

A.3 Single-swap clustering 51
A.4 Warehouse location problem 54

B Convergence curves 57
B.1 Travelling salesman problem 57
B.2 K-means clustering 59
B.3 Single-swap clustering 61
B.4 Warehouse location problem 63

List of Figures

1 Convergence curves for the best strategies on tsp-42. 25
2 Convergence curves for the best strategies on tsp-175. 26
3 MultiEA strategies with different prediction functions on tsp-175. 26
4 Convergence curves for the best strategies on km-18. 29
5 Convergence curves for the best strategies on km-29. 30
6 MultiEA strategies with different prediction functions on km-29. . 30
7 Convergence curves for the best strategies on ss-18. 33
8 Convergence curves for the best strategies on ss-29. 34
9 MultiEA strategies with different prediction functions on ss-29. . . 34
10 Convergence curves for the best strategies on wlp-50. 37
11 Convergence curves for the best strategies on wlp-500. 38
12 MultiEA strategies with different prediction functions on wlp-500. 38

13 Convergence curves for the best strategies on tsp-42. 57
14 Convergence curves for the best strategies on tsp-175. 57
15 Convergence curves for the best strategies on tsp-535. 58
16 Convergence curves for the best strategies on tsp-1032. 58
17 Convergence curves for the best strategies on km-5. 59
18 Convergence curves for the best strategies on km-10. 59
19 Convergence curves for the best strategies on km-18. 60
20 Convergence curves for the best strategies on km-29. 60
21 Convergence curves for the best strategies on ss-5. 61
22 Convergence curves for the best strategies on ss-10. 61
23 Convergence curves for the best strategies on ss-18. 62
24 Convergence curves for the best strategies on ss-29. 62
25 Convergence curves for the best strategies on wlp-50. 63
26 Convergence curves for the best strategies on wlp-200. 63
27 Convergence curves for the best strategies on wlp-500. 64

List of Tables

1 APIs of the best settings of tested strategies for TSP. 27
2 APIs of portfolio algorithms and fixed restarting strategy for TSP. 28
3 APIs of the best settings of tested strategies for k-means. 31
4 APIs of portfolio algorithms and fixed restarting strategy for k-means. 32
5 APIs of the best settings of tested strategies for single-swap. . . . 35
6 APIs of portfolio algorithms and fixed restarting strategy for single-swap. 36
7 APIs of the best settings of tested strategies for WLP. 39
8 APIs of portfolio algorithms and fixed restarting strategy for WLP. 40

9 APIs of restarting strategies for instances of TSP. 45
10 APIs of MultiEA strategies for instances of TSP. 46
11 APIs of MetaMax strategies for instances of TSP. 46
12 APIs of Epsilon-greedy strategies for instances of TSP. 47
13 APIs of restarting strategies for instances of k-means. 48
14 APIs of MultiEA strategies for instances of k-means. 48
15 APIs of MetaMax strategies for instances of k-means. 49
16 APIs of Epsilon-greedy strategies for instances of k-means. 50
17 APIs of restarting strategies for instances of single-swap. 51
18 APIs of MultiEA strategies for instances of single-swap 51
19 APIs of MetaMax strategies for instances of single-swap. 52
20 APIs of Epsilon-greedy strategies for instances of single-swap. . . 53
21 APIs of restarting strategies for instances of WLP. 54
22 APIs of MultiEA strategies for instances of WLP. 54
23 APIs of MetaMax strategies for instances of WLP. 55
24 APIs of Epsilon-greedy strategies for instances of WLP. 56

List of Algorithms

1 MetaMax(k) algorithm 14
2 MetaMax(∞) algorithm 16
3 MetaMax algorithm 18
4 MultiEA algorithm 20

Abbreviations

List of abbreviations used in the text:

API	Aggregated performance index
ECDF	Empirical cumulative distribution function
ERT	Expected running time
KDE	Kernel Density Estimation
TSP	Travelling salesman problem
WCSS	Within-cluster sum of squares
WLP	Warehouse location problem

Introduction

The aim of our work is to compare some promisingly-looking metaoptimization portfolio algorithms with commonly used restarting strategies in the domain of combinatorial optimization by local search.

Many problems that we encounter in real world could be described as searching the best solution that can be described by some configuration in a discrete state-space. Configurations in such space are often being searched by local search, that is by trying to find a better configuration near the best one found so far. In the continuous space we can often employ some of the methods from mathematical analysis that can help us to approximate the direction in which the search for better solution should follow, for us to be as efficient as we can. There are usually no such analytical methods that could be used in a discrete space which accounts for the higher complexity of searching for optimal configurations.

Another problem, found both in combinatorial and continuous optimization by local search is a problem of getting trapped in a local optimum. This problem is often tried to be overcome by restarting the search multiple times from different points of the configuration space. An alternative is to conduct more instances of such search in parallel and give computational time to the ones that seem to be the most promising.

In our work we will analyse algorithms that can be used to build such parallel strategies — MetaMax and MultiEA — and we will compare them with the most common restarting strategies and also Epsilon-greedy algorithm, the representative of bandit strategies. Our aim is to find out whether these algorithms provide viable alternatives that would be comparable with restarting strategies in efficiency. Therby we also continue the work of Viktor Kajml who compared effectivity of MetaMax algorithm with restarting strategies in the domain of continuous optimization (Kajml 2014).

One of the most fundamental questions when solving any computational problem is the question of which of the possible algorithms to choose. The answer to this question depends primarily on the criteria that should such algorithm satisfy. In the case of optimization tasks it will often be some kind of requirement to converge as fast as possible to the solution that is as good as possible. The algorithms examined by us can also help us to find an answer to this question.

The first chapter is devoted to the optimization problems and local search algorithms that we will use to test our selected meta-strategies. In the following chapter we address the question of evaluating the quality of algorithms. We also introduce the measure that we will use to compare the meta-strategies. In the third chapter we look at meta-strategies that we study and explain how they work. Finally in the fourth chapter, we describe our test setup and share our results. Discussion of the results is in the following chapter.

1 Optimization problems and local search algorithms

In a sense, we will deal with two *types* of algorithms in our work. First, there will be local search algorithms that are used to solve the optimization tasks. Then there will be meta-algorithms that will manage and run the local search algorithms, whether repeatedly or more of them in parallel, trying to improve their performance and efficiency when finding the solution. For these meta-algorithms we will prefer to use the term *meta-strategies* (or simply *strategies*) to distinguish them from local search algorithms themselves, to which we will refer simply as *algorithms*.

Throughout our text, we will also use a term *fitness function*, by which we will mean an objective function which value we want to minimize or maximize. We will define one *step* of an algorithm as the smallest amount of fitness function evaluations needed by the algorithm to obtain a new solution.

Selected meta-strategies we tested on a few chosen combinatorial optimization problems that will be described in this chapter.

1.1 Local optimization

Under optimization task we understand a problem of finding minimum or maximum value of some function over a set of all possible values S. In the combinatorial optimization this set is finite. Local search is one way to solve optimization tasks. Having a neighbourhood structure that defines the relations between the elements of set S, local search algorithms start with some initial solution which they iteratively improve. In each iteration they move along the defined neighbourhood structure, generating a new solution from the previous one. This way the search continues unless no improving solution can be found in the neighbourhood.

Since local search algorithms need at any time just one active instance of a problem, they can be used to solve even tasks with very large state-space without problems with available memory.

When a local search ends, it does mean that no improving solution could be found nearby an actual one, but it doesn't mean that no such solution exists globally. Therefore we say that we reached just a *local optimum*.

1.2 Travelling salesman problem

Travelling salesman problem (TSP) is one of classical problems in combinatorial optimization. We are given a set of cities and mutual distances between them.

Our task is to find the shortest path starting in one of the cities, visiting all the other cities and returning to the city it started in. This problem belongs to the class of NP-complete problems.

It can be mathematically described as follows (Hahsler and Hornik 2006). We have a list of n cities $C_1, C_2, ..., C_n$ and their distances in the form of a distance matrix D in which an element at a position d_{ij} represents the value of a distance between cities C_i and C_j. We can construct such permutation π consisting of elements $1, 2, \ldots, n$ where each element at position $\pi(i)$ is an index of a city that follows the city C_i on the path that describes one particular solution to the TSP problem. The goal here is to find such permutation π that minimizes the total distance:

$$\sum_{i=1}^{n} d_{i\pi(i)} \tag{1}$$

There are algorithms that can find exact solutions of TSP, but as it is NP-complete, they are not very fast on larger instances. In these cases algorithms incorporating various heuristics are used instead. They are not guaranteed to find an optimal solution, however in many cases this is not so important as long as they find a sufficiently good one.

Heuristics for solving TSP can be divided into two main groups. There are *construction heuristics* that build solution from scratch. Another group consists of *iterative improvement* heuristics that iteratively try to construct a better path from the existing one. This group contains also local search algorithms that we are interested in. They impose some additional rules that enable us to build a valid path from another one, thus allowing to use the local search.

K-opt heuristic

This heuristic first removes a set of k edges that are not adjacent, thereby dividing the path into multiple disconnected segments. It then reconnects these segments by adding another k edges so that the segments will form a cyclic path again. This is called a *k-opt move*. Usually, *2-opt* or *3-opt* moves lead to good solutions in acceptable time.

Lin-Kernighan heuristic

This heuristic, described by Lin and Kernighan in (Lin and Kernighan 1973), is a generalization of *k-opt* heuristic explained above. Instead of using *k-opt* moves with the fixed value of k during the search, it uses the value of k that leads to the biggest improvement in that particular move. Implementation of this heuristic is simplified by the fact that for any *k-opt* move there exists a sequence of *2-opt* moves such that the path that is a result of applying all the moves from this sequence is the same as the one created by given *k-opt* move.

1.3 Centroid-based clustering

In centroid-based clustering we are given a set of n samples, represented by points in euclidean space, and goal is to find k representatives for this set, such that the distance between a point and its closest representative is minimized. The most common measure is called *within-cluster sum of squares* (WCSS) and is formulated as:

$$\sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2 \tag{2}$$

where S_i is a set of points that are assigned a representative μ_i - i.e. a cluster. This problem is NP-hard but there exists many heuristic algorithms that can be used to find a sufficiently good solution.

K-means algorithm

K-means is one of the most popular heuristic algorithms used to solve the tasks of centroid-based clustering. It begins by finding an initial solution and then it works by improving the solution in iterations comprised of two phases. In the first phase, each point x from n input samples is assigned a centroid to which the point has the lowest within-cluster sum of squares. Points that have the same centroid i together form a cluster, a set S_i:

$$S_i = \Big\{ x : ||x - \mu_i||^2 \leq ||x - \mu_j||^2 \;\; \forall j = \{1, ..., k\} \backslash \{i\} \Big\} \tag{3}$$

In the second phase the recomputation of centroids is being done. Each set of points S_i is assigned a new centroid which becomes the mean of all the points from the set:

$$\mu_i = \frac{1}{|S_i|} \sum_{x \in S_i} x \tag{4}$$

While the *k-means* algorithm is guaranteed to converge to the local optimum, overall quality of such solution is heavily dependant on the initialization. One simple initialization approach is to take k random points of the euclidean space and make them centroids. There are also advanced initialization techniques such as doing preliminary clustering on the subset of the input samples or choosing the random sample as the first centroid, then selecting the next centroid as the sample that is the most distant from the centroids already chosen and continuing in this way until all k centroids have been specified[1].

[1] This is called *k-means++* algorithm

Single-swap algorithm

Single-swap algorithm uses a simple local search heuristic that creates a new solution from the previously obtained one by swapping one centroid with another as described in (Kanungo et al. 2002). Specifically, it begins by choosing initial k centroids from the set of n sample points. Then it chooses the centroid i that will be replaced by another one. New set of centroids is then constructed by removing the selected centroid and replacing it with a new centroid taken from the set of sample points as well. If the resulting solution has better value of the fitness function [2] than the previous one, it is kept. Otherwise the algorithm returns to the previous solution. This process is repeated until no substantially better solution can be found by swapping the centroids.

1.4 Warehouse Location Problem

Warehouse location problem (WLP, see (Kuehn and Hamburger 1963)), also known as Facility location problem, is an example of optimization problem with constraints. Using similarity with set-packing, Krarup and Pruzan in (Krarup and Pruzan 1983) showed that this problem is NP-complete. It has many variants and definitions. The one we used in our work can be formulated as follows. Assume that a company has m customers which it want to serve with goods located in its warehouse facilities. Each of these customers has a demand d_c. The company has N warehouses, each with the capacity cap_w. Every customer has to be served by exactly one of the warehouses. To serve a customer c by a warehouse w, an amount of t_{cw} equal to the transport cost from the warehouse to the customer is needed to be spent. Warehouses can satisfy the demands of their customers just within the limits of their capacities. In the case that some warehouse serves at least one customer, the amount of s_w, needed to open the new warehouse, has to be added to the costs. Goal is to find such assignment of warehouses to customers that minimizes the cost to satisfy all their demands.

Mathematically, we can describe the problem as follows. Let's introduce a symbol a_w representing a set of customers served by warehouse w. We want to minimize the value of:

$$\sum_{w \in N} \left((|a_w|) > s_w + \sum_{c \in a_w} t_{cw} \right) \tag{5}$$

subject to the conditions:

$$\sum_{c \in a_w} d_c \leq cap_w \quad \forall w \in N \tag{6}$$

$$\sum_{w \in N} (c \in a_w) = 1 \quad \forall w \in M \tag{7}$$

[2] Usually WCSS, see above for definition.

One simple local search algorithm that can be used to solve this problem is based on the idea similar to the clusterswap algorithm described above. At the beginning we assign a warehouse to all customers in the following manner. Starting from the customers with the highest demand, to each customer we assign a warehouse with the lowest transport cost that has any capacity left to serve this customer. If there is no warehouse able to satisfy the demand of the customer, we assign the customer to a randomly selected one. Then in each step of the algorithm we randomly select a customer and try to assign it to other randomly selected warehouse. If this move results in a better fitness value, we change the assignments appropriately. Otherwise we revert to the previous configuration. To avoid stepping into the invalid solution, that would violate the constraints, we make such solutions practically unacceptable by giving them a very large fitness value.

2 Evaluation of algorithm quality

In this chapter we will look at the problem of evaluating the algorithm quality. This is a key question as there exists many ways how to do so and the answer is crucial when it comes to comparing the performance of different algorithms among themselves.

Another area when this is quite important is finding a good algorithm configuration. Almost every algorithm usually has several parameters, setting values of which is left to the concrete implementation. These settings are often problem-specific and their optimal values may be different for each instance of the problem. Coming up with optimal configuration can be a complex problem in itself and likely will include running the algorithm many times with different values of parameters. Such parameter tuning may, however, dramatically improve the performance of an algorithm.

2.1 Fitness function

Essential aspect of the quality of optimization algorithm is the quality of solutions it finds, which is reflected in the fitness function.

A simple way how to compare the ability of algorithms to find a good solution is, for example, graph of the average fitness value of the best solution found so far, dependent on the number of fitness evaluations available to the algorithm. In addition to the quality of the final solutions it also shows the whole search process and the convergence of the fitness.

When comparing algorithms, we often want one concrete number according to which we could determine which of them will be considered better for our needs.

One option to obtain such number from the graph of a fitness value is to specify a certain number of fitness evaluations b and to take as a metric the quality of the best solution found with a budget of b fitness evaluations.

However this quality measure has a few problems. One of them can be shown in the case of two algorithms, which converge to the same value of a fitness function using different number of fitness evaluations. In both cases, the number of evaluations used is lower than the specified budget b, so this measure will report the same quality although one of the algorithms is clearly superior.

Another option is to specify a certain value of fitness function v and compare the algorithms according to how much fitness evaluations they consume until they find the first solution with fitness value that is equal or better than v.

Though here remains a question what to do with algorithms whose fitness value has never reached the value v. One solution could be to penalize such algorithms with large value of metric (or even infinity).

Common problem, that both metrics described above share, is that the quality of algorithm does not take into account the ongoing progress in the quality of the solutions found during the search.

2.2 ECDF of running times

Another indicator that is used when comparing the quality of algorithms in general is their expected running time, that is, the time that we can expect that the algorithms will take to find a solution of a particular quality. We use the definition from (Hansen et al. 2010):

$$ERT(f_t) = \frac{fe(f \leq f_t)}{sr} \quad (8)$$

where f_t is target value of a fitness function, $fe(f \leq f_t)$ is the number of fitness evaluations in which the target value was reached and a total number of evaluations otherwise and sr is the number of successful runs of the algorithm, i.e. those in which the target fitness value was achieved.

ERT of algorithms can be compared, for example, using the graph of empirical cumulative distribution function (ECDF), which shows what fraction of algorithm runs achieved some target value of a fitness function, depending on the size of the fitness evaluations budget for given runs of the algorithm.

2.3 Aggregated performance index

For similar reasons as mentioned above, we have compared the tested meta-strategies using a measure called Aggregated performance index (API), as defined in an article of P. Pošík which has not been published yet (Pošík 2012). It gives us one number that takes into account the whole ECDF graph of ERTs and can be derived as follows.

If one had a perfect algorithm that immediately finds the optimal solution for any problem, its ECDF graph of ERT would consist of a line defined by equation $y = 1$. In other words, the better the algorithm the smaller the area above the curve of the ECDF graph. Consider now ERT of some algorithm for a given problem and target function. This ERT can be depicted by horizontal line connecting a point on the y axis with a point on the curve. The length of this segment is proportional to the value of ERT. If we place more such lines that represent various problems on the graph, the sum of their lengths will be proportional to the area above the ECDF curve. Instead of the sum we can use their arithmetic mean which is equivalent to the logarithm of the geometric mean. And since the logarithm is an increasing funcion we can take the geometric mean

of ERTs for many problems as our measure, the Aggregated performance index:

$$API_A = \exp\left(\frac{1}{|P|}\sum_{p\in P}\log(ERT_{A,p})\right) \tag{9}$$

3 Meta-strategies

In this chapter we will describe various strategies that can be used to improve the efficiency of problem-solving by local search, some of which we tested in our work.

3.1 Restarting strategies

Quality of solutions found by search algorithms to some extent always depends on their initial parameters. As for local search algorithms, probably the most important initial parameter is the location from which the search is started. Without taking any special measures there is a high probability that the solution found by local search will be at most as good as the local optimum accessible from that initial location, even when better solutions may exist in other parts of state-space. This problem is inherent to many of the local search tasks.

Restarting strategies are one means of how to overcome this problem. Run of an algorithm is interrupted and the search is started anew from another location. This is done repeatedly, every time certain conditions are met. Using restarting, algorithm searches in different parts of state-space and finds a solution that is equal to the best of their local optima.

There are many restarting strategies, some of which are algorithm-dependent. We will focus on a few basic ones that are easy to implement regardless of the type of a search algorithm used and often dramatically improve results obtained by local search.

One of the simplest restarting strategies is to start a new search from a different location every time after some predefined, constant number of steps n. We will call it *constant restarting strategy*. Proper selection of parameter n is important, because its optimal value can be different for each used search algorithm and instance of a problem that is being solved.

Another approach is to restart the search only when the algorithm is not able to find another improving solution. The question is how to detect that.

One option is to keep track of where the algorithm currently is in the solution-space and start a new search when it has already searched certain part of it. This approach, however, is dependent on the particular representation of a search space and cannot be used, for example, in the case of black-box optimization where we don't explicitly see the solution-space.

Another option is to look at the succesive development of the quality of solutions found by algorithm and restart the search if no further improvement can be observed. Since we don't look into the solution-space in this case, we also cannot decide at any time whether a better solution will be found somewhere in the

future or not. Therefore we need some method to make at least a guess. We can say, for example, with certain probability, that no improving solution will be found when it hasn't been found during the last k steps taken by the algorithm. In our work we will call this strategy *stagnation-detecting restarting strategy.*

3.2 Bandit strategies

Multi-armed bandit is a decision problem of an agent choosing from some available actions, each of which provides a reward, in a such way as to maximize the sum of the rewards obtained. More formally, let's have a set of n different actions $a_1, a_2, \ldots, a_n$. The agent is selecting actions in repeated trials, one action at a time. At time t he chooses the action $a_i(t)$ and obtains the reward $r_i(t)$ related to taking the selected action. Rewards of each action are random, unknown and independently distributed.

Multi-armed bandit problem can be also viewed as an explore-exploit task. To perform well, agent should choose the action providing the best reward as many times as possible. But to find out which of the different available actions are the most promising ones at the moment, he should also try as many of them as possible during the trials. This is also called an exploration-exploitation trade-off. There have been proposed many strategies that can be used to select an appropriate action while maintaining some form of balance between exploration and exploitation.

3.2.1 Epsilon-first

Epsilon-first is a simple bandit strategy that keeps the phases of exploration and exploitation unmixed. Pure exploration is used first for $\epsilon \cdot T$ trials to explore the rewards of the actions at the beginning. For the remaining $(1 - \epsilon) \cdot T$ trials, the most promisingly looking action is being selected (where T is the desired number of trials). The value of ϵ must belong to the open interval $(0, 1)$. Question is how to choose this parameter ϵ as optimally as possible and there's no specific receipt defined by the strategy.

3.2.2 Epsilon-greedy

Epsilon-greedy is probably the most commonly used from the family of simple bandit strategies. First introduced in (Watkins 1989), this strategy chooses random action uniformly with the probability of ϵ and with the probability of $P = (1 - \epsilon)$ it chooses the action with the highest reward. This distribution of explorative vs. exploitative behavior has the advantage that the agent can compare the quality of the chosen action with the other ones during the whole duration of the process. The choice of parameter ϵ is left to the specific implementation here as well.

3.2.3 Epsilon-decreasing

Epsilon-decreasing strategy is similar to the *epsilon-greedy* in that that it chooses the best action with the probability of ϵ. However, this time the value of ϵ decreases over time with the factor $1/t$. This provides plenty of space for exploration in the beginning when rewards of all the actions are not known very well yet. As the process progresses and agent tries more and more actions he gains more knowledge about their suitability and he gradually reserves more and more time to spend on the best one.

3.3 Portfolio algorithms

When solving an optimization problem we are often faced with the fact that there are several different algorithms that we can use to find a solution. However, it is often difficult to say in advance which of these algorithms will be the best when dealing with the given problem instance. One possible solution to this problem is to solve the given instances of an optimization problem with all algorithms. Thus we are guaranteed to find as good solution as is able to find an algorithm that is the most suitable for the given instance. On the other hand, running all of these algorithms sequentially would be a very unefficient process. Here come the so-called *portfolio algorithms*.

Portfolio algorithm is a meta-strategy that solves an instance of a problem using multiple algorithms at once, assesing their suitability for given instance of an optimization problem and trying to allocate computing resources efficiently, i.e. so that the amount of resources allocated to a particular algorithm corresponds to its success in finding a solution.

3.3.1 MetaMax

MetaMax presented by Gyorgy and Kocsis in (György and Kocsis 2011) is a portfolio algorithm that can be used as a multi-start optimization strategy that dynamically allocates processing resources to the algorithm instances that seem to be the most promising ones.

It works in iterations divided into two parts. Firstly it runs all local search algorithms from the portfolio on the given problem instance for one step. After that it computes the value of a fitness function of their new solutions and based on that it choses those of them that will be run in the next iteration, using some heuristic function.

There are three different variants of this meta-strategy that we will describe. Please note that although we used a version of *MetaMax* modified for searching of minima in our experiments, here we describe its variant designed for solving maximization tasks, as it is presented by its authors in (György and Kocsis 2011).

MetaMax(k)

MetaMax(k) is a version of the strategy having a constant number of algorithms in its portfolio whose goal is to achieve the solution quality similar to the quality of the solution that the best of the algorithms in the portfolio is able to achieve.

Inputs (see Algorithm 1) are portfolio of algorithms and a heuristic function h, that must be positive, monotone and decreasing with $h(0) = 1$ and $lim_{n\to\infty}h(n) = 0$. In the initial phase all algorithms in the portfolio are stepped once and the values of a fitness function of their respective solutions are saved.

In the next step there is a selection of algorithms that will be run in the following iteration. In each iteration we choose algorithms A_i for which the point $(h(n_i, \hat{f}_{X_i}))$ lies on the corner of an upper right convex hull of a set:

$$\left\{(h(n_i), \hat{f}_i) : j = 1, ..., k\} \cup \{(0, max_{i=1,...,k}(\hat{f}_i))\right\} \tag{10}$$

Role of the point $(0, max_{i=1,...,k}(\hat{f}_i))$ in this set is to ensure that we will subtract left convex hull from the upper one defined by points $\{(h(n_i), \hat{f}_i)\}$. We do that because we are not interested in algorithms that use equal or grater number of steps (and thus have lower or equal value of a decreasing function $h(n_i)$) to find a solution with at most equal value of a fitness function then other ones. This selection principle is somewhat similar to the process of finding a pareto-optimal front (set of non-pareto-dominated solutions) with two criteria: maximizing value of a fitness function and minimizing the number of steps used.

This selection process guaranties that in each iteration we select algorithms that achieve the highest values of a fitness function and also those that have been run for the lowest number of steps so-far. Thus we approach an optimal distribution of computational resources between exploitation of the effective and exploration of new and potentionally promising algorithms.

In the case that multiple algorithms with equal number of steps are about to be selected we will choose just one of them at random because we don't want to assign unnecessarily much of the computational resources to similarly behaving algorithms.

Mathematically, the selection process described above is derived as follows. In (György and Kocsis 2011) authors show a lemma that states that there exist such non-negative non-increasing function $g(n)$ with $lim_{n\to\infty}g(n) = 0$ that:

$$P\left(\hat{f}^* - f(\hat{X}_n) \leq g(n)\right) \geq 1 - \delta \tag{11}$$

where $\hat{f}^*$ represents a local optimum and $f(\hat{X}_n)$ is a sequence of fitness function values of the best solutions found so-far by the algorithms in the portfolio. The problem is that in most cases an equation of a function g is unknown and therefore also the convergence rate of a local search algorithm can't be accurately

```
    input  : Algorithm instances A_1, A_2, ..., A_k
             Positive monotone decreasing function h with h(0) = 1
             and lim_{n→∞} h(n) = 0
    output : best solution found and corresponding value of fitness function
    Initialization:
 1  for i = 1 to k do
 2  |  Step algorithm instance A_i to the location X_i and set its variables:
 3  |  number of steps n_i = 1
 4  |  best fitness reached f̂_i = f(X_i)
 5  |  location of maximum X̂_i = X_i
    end
    Evaluation:
 6  for r = 1, 2, ... do
 7  |  for i = 1 to k do
 8  |  |  Select algorithm A_i if there exists c > 0 such that:
    |  |  f_{X_i} + ch(n_i) > f_{X_j} + ch(n_j)) for all j = 1, ..., k such that
    |  |  (n_i, f_{X_i}) ≠ (n_j, f_{X_j}). If there are more algorithms A_i selected that
    |  |  have the same number of steps n_i, select only one of them at
    |  |  random.
    |  end
 9  |  Step each selected algorithm A_i to the location X_i and update its
    |  variables:
10  |  number of steps n_i = n_i + 1
11  |  best fitness reached f̂_i = max(f̂_i, f(X_i))
12  |  if best fitness has changed then
13  |  |  new location of maximum X̂_i = X_i
    |  end
14  |  index of currently best algorithm I = argmax_{i=1,...,k}(f̂_i)
15  |  estimate of location of optimum X̂ = X̂_I
16  |  fitness of the estimated optimum f̂ = f̂_I
    end
17  return (X̂, f̂)
```

Algorithm 1: MetaMax(k) algorithm

determined. In some cases, however, it can be estimated at least asymptotically, with unknown constant factor c. In those cases we can use the lemma mentioned above to construct a group of several estimates of a local optima $\hat{f}^*$ as:

$$\hat{f}_i + cg(n_i) \qquad \forall c \in \mathbb{R} \tag{12}$$

In each round we can then select those algorithms that for some value of a

constant factor c achieve the greatest estimates of a fitness function value in the optimum. However, since in most cases the equation of a function g is not known even up to a constant factor, in the *MetaMax* strategy this condition is loosened and function g is replaced with any positive monotone decreasing function h with $\lim_{n\to\infty}(n) = 0$ and $h(0) = 1$.

Algorithms chosen by the selection process described above are then stepped once and respective variables storing the number of steps used and the best solution found are then updated for each of them. This whole process of selection and stepping the selected algorithms is then repeated. When designated termination conditions are met[1] the best solution found by all algorithms from the portfolio is returned.

By optimal algorithm we will mean such an algorithm that converges to the best estimate of $\hat{f}^*$. The authors in (György and Kocsis 2011) prove that with increasing number of iterations the quality of a solution found by *MetaMax(k)* strategy becomes asymptotically identical to that of the best algorithm in its portfolio. For any two suboptimal algorithms converging to the same solution applies that in each iteration just one of them will be selected. *MetaMax(k)* therefore prevents a large number of such algorithms to overtake the computational resources. There also exists such iteration r that from this iteration further on the optimal algorithm will be selected as well. And if the local search algorithms from the portfolio converge at least exponentially fast, provided that some additional conditions [2] placed on the convergence rate of a function h are met, it holds that half of the total number of fitness function evaluations is used by the optimal algorithm.

MetaMax(∞)

MetaMax(∞) (see Algorithm 2) is an adaptation of *MetaMax(k)* with unlimited number of algorithms in the portfolio. At the beginning the portfolio contains only one algorithm and in each iteration another one is added. Thus we can bypass the problem that when there is a constant number of algorithms in the portfolio our search is inconsistent (except when the algorithms are guaranteed to be able to find a global optimum, but with the local search algorithms this case is unlikely). However, when we are adding more and more new algorithms to the portfolio, and run the strategy for unlimited time, *MetaMax(∞)* can guarantee that the optimum will be found. Note that the term *new algorithm* can be understood both as a completely new local search algorithm, as well as an instance of the

[1] E.g. after some fixed number of iterations, after reaching some value of the fitness function, after exceeding some predefined time-period during which no improving solution has been found, etc.

[2] See (György and Kocsis 2011) for mathematical details.

```
input  : Positive monotone decreasing function h with h(0) = 1
         and lim_{n→∞} h(n) = 0
output: best solution found and corresponding value of fitness function
1  actual round r = 1
2  for r = 1, 2, ... do
3      Add new algorithm A_r and initialize it by setting its variables:
4      number of steps n_r = 0
5      best fitness reached f̂_r = 0
6      for i = 1 to r do
7          Select algorithm A_i if there exists c > 0 such that:
           f_{X_i} + ch(n_i) > f_{X_j} + ch(n_j)) for all j = 1, ..., r such that
           (n_i, f_{X_i}) ≠ (n_j, f_{X_j}). If there are more algorithms A_i selected that
           have the same number of steps n_i, select only one of them at
           random.
       end
8      Step each selected algorithm A_i to the location X_i and update
       variables:
9      number of steps n_i = n_i + 1
10     best fitness reached f̂_i = max(f̂_i, f(X_i))
11     if best fitness has changed then
12         new location of maximum X̂_i = X_i
       end
13     index of currently best algorithm I = argmax_{i=1,...,r}(f̂_i)
14     estimate of location of optimum X̂ = X̂_I
15     fitness of the estimated optimum f̂ = f̂_I
16     actual round r = r + 1
   end
17 return (X̂, f̂)
```

Algorithm 2: MetaMax(∞) algorithm

same algorithm that is initialized to another (random) location in the state-space in which we search an optimum. In this way we can easily create an unlimited number of algorithms that we can add to the portfolio.

The authors also mention the possibility to use a different function h in each round by which we could control the length of individual iterations. The function h could, for example, depend on the overall number of steps used by all algorithms selected in the previous round. In our work we don't examine this possibility further.

MetaMax

MetaMax (see Algorithm 3) differs from the *MetaMax(∞)* in that that each newly-found best algorithm A_{I_r} is stepped so many times until his number of steps n_r equals to the number of steps taken by previously best known algorithm $A_{I_{r-1}}$. Random selection of algorithm from the ones having equal number of steps is also replaced with chosing the first one (the one that has the smallest index).

Practical problem which the authors in their work don't address is that with growing number of algorithms in the portfolio the space and time complexity associated with their management grows as well. This is why we limit the number of algorithms contained in the portfolio at the same time. In each iteration of *MetaMax*, before adding a new algorithm to the portfolio, we remove the algorithm that was lately selected by the *MetaMax* selection in the iteration with the smallest index r among all algorithms in the portfolio. In cases where multiple algorithms satisfy these conditions, the algorithm with the lowest index number[3] is always selected.

For variants of *MetaMax* with unlimited number of algorithms in the portfolio the authors show that these are consistent in the sense that with high probability it holds that:

$$\lim_{r \to \infty} \hat{f}_r = f^* \tag{13}$$

They further demonstrate that suboptimal algorithms can be selected only a finite number of times and thus they also show the upper bound for the duration of each round. For *MetaMax* variant of the strategy, they then restrict the number of steps of an optimal algorithm as:

$$n_{I,r} \geq \frac{\sqrt{2t_r + 7} - 1}{2} \tag{14}$$

where t_r represents the total number of steps taken by all algorithms in the portfolio. From that they derive the convergence rate of *MetaMax* as:

$$f^* - \hat{f}_r \;\leq\; g\left(n_{I,r} \geq \frac{\sqrt{2t_r + 7} - 1}{2}\right) \tag{15}$$

3.3.2 MultiEA

Multiple Evolutionary Algorithm (MultiEA) is a portfolio algorithm proposed by Yuen, Chow and Zhang in (Yuen, Chow, and Zhang 2013) that can be used as a strategy that selects algorithm that is the most suitable for solving given

[3] The one that was the earliest added to the portfolio.

input : Positive monotone decreasing function h with $h(0) = 1$ and $lim_{n\to\infty} h(n) = 0$
output: best solution found and corresponding value of fitness function

```
1  actual round r = 1
2  for r = 1, 2, ... do
3      Add new algorithm A_r and initialize it by setting its variables:
4      number of steps n_r = 0
5      best fitness reached f̂_r = 0
6      for i = 1 to r do
7          Select algorithm A_i if there exists c > 0 such that:
           f_{X_i} + ch(n_i) > f_{X_j} + ch(n_j)) for all j = 1, ..., r such that
           (n_i, f_{X_i}) ≠ (n_j, f_{X_j}). If there are more algorithms A_i selected that
           have the same number of steps n_i, keep only the one with the
           smallest index.
       end
8      Step each selected algorithm A_i to the location X_i and update
       variables:
9      number of steps n_i = n_i + 1
10     best fitness reached f̂_i = max(f̂_i, f(X_i))
11     if best fitness has changed then
12         new location of maximum X̂_i = X_i
       end
13     index of currently best algorithm I_r = argmax_{i=1,...,r}(f̂_i)
14     if I_r ≠ I_{r-1} then
15         step algorithm A_{I_r} (n_{I_{r-1}} - n_{I_r} + 1) times
16         set number of steps n_{I_r} = n_{I_r} + 1
       end
17     estimate of location of optimum X̂ = X̂_{I_r}
18     fitness of the estimated optimum f̂ = f̂_{I_r}
19     actual round r = r + 1
   end
20 return (X̂, f̂)
```

Algorithm 3: MetaMax algorithm

optimization problem. Authors describe implementation of *MultiEA* aimed at minimization tasks.

Basic instrument that *MultiEA* uses to compare algorithms from the portfolio between themselves is so called *convergence curve*. It consists of the set of points $C = \{(j, f_i(j))\}$, where j is an ordinal number of actual *MultiEA* iteration and $f_i(j)$ is a fitness function value of the best solution found by algorithm A_i up

to the j-th iteration (inclusive). Convergence curves of each algorithm are then used to make estimates of their fitness values in some future point, common to all algorithms. These estimates are constructed using novel prediction measure introduced by authors, which according to them overcomes some of the problems found in standard extrapolation techniques and also takes into account the fact that convergence curve of a fitness function is non-increasing.

Estimation works as follows (Yuen, Chow, and Zhang 2013). For each convergence curve defined by set of points $\{(j, f_i(j))\}$ we define a subset:

$$C(l) = \Big\{(j - l, f_i(j - l)), (j - l + 1, f_i(j - l + 1)), ..., (j, f_i(j))\Big\} \quad (16)$$

Symbol l denotes the history length. We do linear regression of multiple such subsets with different history length and obtain lines in a form $y = ax + b$ where $(x, y) \in C$. We choose some future point t for which we want to estimate the fitness values. Using subsets of a convergence curve we than compute these estimates as $pf(l, t) = at + b$. We construct a bootstrap probability distribution $bpd(t)$ over the set of these estimates obtained from subsets with different history length. By sampling from this distribution we get the final estimate $\hat{f}(t)$ of a fitness function value. Authors implemented computing the bootstrap probability distribution using kernel smoothing estimate function *ksdensity* from Matlab, stating they used default values for all of its optional parameters.

MultiEA (see Algorithm 4) starts by stepping every algorithm A_i that is in the portfolio so many times until the fitness value of its best found solution changes. After that the common future point t is chosen. For each algorithm we then construct subsets of its convergence curve described above, compute corresponding estimates $pf(l, t)$ and from them compute final estimates $\hat{f}_i$ using bootstrap probability distribution. Algorithm which gives the best estimate is stepped in the next iteration of MultiEA.

Authors in their work assume that portfolio consists only of evolutionary algorithms that work with a population of solutions and therefore they use some additional variables denoting e.g. the population size of an algorithm. For our purposes this was not needed so we ommited those symbols from the description of the algorithm. Note that we also modified the notation as a whole to make it somewhat similar to the one used when describing *MetaMax*.

input : Algorithm instances $A_1, A_2, ..., A_k$
history length l
output: best solution found and corresponding value of fitness function

Initialization:

1 **for** $i = 1$ **to** k **do**
2   number of steps $n_i = 0$
3   X_i = initial solution
4   best fitness reached $\hat{f}_i(0) = f(X_i)$
  repeat
5   step algorithm A_i to new location X_i and update its variables:
6   number of steps $n_i = n_i + 1$
7   best fitness reached $\hat{f}_i(n_i) = f(X_i)$
8   location of minimum $\hat{X}_i(n_i) = X_i$
  until $\hat{f}_i(n_i) < \hat{f}_i(n_i - 1)$;
end

Evaluation:

9 **for** $r = 1, 2, ...$ **do**
  nearest common future point $t = max(n_1 + 1, n_2 + 1, ..., n_k + 1)$
10   **for** $i = 1$ **to** k **do**
11   from convergence curve $C_i = \{(j, \hat{f}_i(j)) | j = 1, ..., n_i\}$ create subcurves $C_i(l), ..., C_i(n_i - 1)$
12   for each subcurve compute a prediction $pf(l, t) \quad \forall l = 1, ..., n_k - 1$
13   construct bootstrap probability distribution $bpd(t)$ and sample from it to get $pf(t)_i$.
  end
14   index of algorithm with the best prediction $p = argmin_{i=1,...,k}(pf(t)_i)$
15   step algorithm A_p once and update its variables:
16   number of steps $n_p = n_p + 1$
17   best fitness reached $\hat{f}_i = min(\hat{f}_i, f(X_i))$
18   **if** *best fitness has changed* **then**
19   new location of maximum $\hat{X}_i = X_i$
  end
20   index of currently best algorithm $I = argmin_{i=1,...,k}(\hat{f}_i)$
21   estimate of location of optimum $\hat{X} = \hat{X}_I$
22   fitness of the estimated optimum $\hat{f} = \hat{f}_I$
end
23 **return** $(\hat{X}, \hat{f})$

Algorithm 4: MultiEA algorithm

4 Experiments

In the following text we describe how we tested selected algorithms and meta-strategies and the results we observed.

4.1 Optimization problems and their instances

For our tests we have chosen four problems that were introduced in the chapter 1 alongside with the algorithms that we use for solving them.

The first one is the Travelling salesman problem (TSP), a classical problem of combinatorial optimization. We used four TSP instances from TSPLIB project[1] with different number of cities, namely 42, 175, 535 and 1032. They all have known optima, so we could see how will our algorithm behave. While the optimum of the easiest TSP instance is rather easy to find, solving the largest one takes much more computational resources. We implemented a local search algorithm that uses *2-opt* heuristic to find solutions for the instances.

Another problem we were testing meta-strategies with is the centroid-based clustering. It is a common problem that is being used in many real-world tasks and it is also one of the tasks which authors of (György and Kocsis 2011) used to illustrate the abilities of MetaMax. We studied two different versions of clustering task. First is the *k-means* algorithm that uses informed search with heuristic and converges relatively quickly. We therefore expected that this will be the task in which portfolio algorithms with unbounded number of instances, like *MetaMax(∞)* or *MetaMax*, will shine. The question is, how will they compare to restarting strategies for which this task is suited as well. We synthetically generated many instances of a clustering problems in the two-dimensional space from which we selected a few that were nicely separable with the following number of clusters: 5, 10, 18 and 29. We also employed another clustering algorithm, *single-swap* local search described in the first chapter. It uses less clever heuristic than *k-means*, so it will converge slower and therefore the portfolio algorithms with unbounded number of instances shouldn't have so much of an advantage against the ones with the fixed number of them, we suppose.

We also included the Warehouse location problem (WLP) that represents a task of combinatorial optimization with constraints. This types of problems are somewhat harder to solve with just a simple modifications of local search algorithms and we are curious to see how will meta-strategies help here. We use the instances with 50, 200 and 500 warehouses (the number of customers in each instance is equal to the number of warehouses) that have known optima.

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

4.2 Meta-strategies and their configuration

Meta-strategies that we tested are of a several varieties. There is a group of commonly used restarting strategies, then there is a family of portfolio algorithms consisting of MetaMax variants and MultiEA strategy. Bandit strategies are represented by Epsilon-greedy strategy. They are described in the previous chapter. Here we will outline what specific configuration we used for our experiments.

Sometimes we wanted to make certain parameters of a strategy dependent on the size of a problem instance. For these cases, we introduce a symbol s, that denotes the size of a problem instance, depending on the type of a problem. When we talk about TSP, it will mean the number of cities in a particular TSP instance. In clustering tasks it represents the designated number of clusters and in WLP it refers to the number of the warehouses.

The first and the most simple restarting strategy that we tested is *constant restarting*. In this strategy we restarted local search every $r \times s$ evaluations of a fitness function, where r is a restarting coefficient. This is a simple stategy whose efficiency when solving the problems will probably heavily depend on the proper setting of a parameter r. For our purposes we used these values of the coefficient: $100, 200, 500, 1000, 2000$ and 5000.

Another restarting strategy that we use we called *stagnation-detecting restarting*. This strategy restarts the local search when no improving solution has been found during the last $rc \times s$ evaluations of a fitness function. This strategy is more robust and it should be less sensitive to the proper setting of the parameter r. We also expect this strategy to achieve better results than constant restarting, especially in larger (and more difficult) problem instances. The values used for tuning the parameter r are the same that we used with the previous strategy.

The other group of meta-strategies we examined is the *MetaMax* family of portfolio algorithms. It should be able to automatically detect and assign more computational resources to promising instances of local search.

We did experiments with all three variants of *MetaMax* described in the previous chapter. *MetaMax(k)* is the most simple one of them, with fixed number of algorithms in its portfolio. For the number of local search algorithms in the portfolio, we tried six different settings: $10, 20, 50, 100, 200$ and 500 to see which value will be the most efficient for different instances of the tasks.

MetaMax(∞*)*, the modification of *MetaMax(k)* with unbounded number of instances adds an effect similar to that of random restarts. In every iteration it adds a new instance of local search into its portfolio. Because with increasing number of iterations this brings consistency, it should be more successful than *MetaMax(k)*, especially when solving more difficult problems. *MetaMax* is another variant from this family of strategies that improves the convergence of a promising local search instances. If some instance of an algorithm from the portfolio becomes the best one, it is run for increased number of evaluations.

Due to practical reasons we modified *MetaMax(*∞*)* and *MetaMax* strategies so that before adding a new algorithm to the portfolio, one algorithm instance

is also removed, as we described in the previous chapter. For both of them, we tested the same six portfolio sizes as with *MetaMax(k)*.

Another portfolio algorithm we looked at is the *MultiEA*. Despite the fact that the authors in (Yuen, Chow, and Zhang 2013) state that it is parameterless, in fact it has a few parameters which values could vary. The first one is history length, the number of the last l fitness values achieved by a particular algorithm from the portfolio that will be recorded to predict its potential in the future. While the authors describe a variant of *MultiEA* that keeps the whole history of runs for each instance, because of practical reasons we limited it to the last 15 records. We believe this shouldn't affect the prediction in a significant way.

Another, even more important parameter of *MultiEA* is function used for the prediction of future values of solutions found by algorithms contained in its portfolio. Authors in their work stated: *"We use Matlab statistical toolbox function ksdensity. Strictly speaking, ksdensity has some parameters that can be varied; in this paper, we have used the standard values provided in Matlab for a «standard» interpolation."* (Yuen, Chow, and Zhang 2013). With default values of its parameters, this implementation of a kernel density estimator estimates probability density function using gaussian kernel and bandwidth defined according to the *Silverman's rule of thumb*, that is:

$$h = \sigma \times \left(\frac{4}{3n}\right)^{\frac{1}{5}} \tag{17}$$

where σ is the standard deviation and n is the number of samples from which we want to estimate the density. This, according to the documentation of Matlab *ksdensity* function, *"is optimal for estimating normal densities"* [2]. One of the *MultiEA* variants that we use is implemented with the same prediction process, sampling from the density estimated by kernel density estimator with gaussian kernel and bandwidth value determined according to the rule described above. We also implemented another two variants of this meta-strategy that replace the sampling process with the arithmetic mean and median of the interpolations of the last l fitness values achieved by given algorithm instance. We think that the process of selecting the predicted values by KDE bootstrapping is unnecessarily complicated and we expect these two modified versions of MultiEA to deliver similar results than the original one, despite their simplicity.

The last strategy that we tested is the *Epsilon-greedy* that belongs to the larger group of bandit strategies. *Epsilon-greedy* is one of the simplest and the most commonly used of them. With the probability of $1 - \epsilon$ it runs the instance of a local search algorithm from the portfolio that achieved the best results so-far and otherwise it runs another randomly selected instance. For parameter tuning we used the values of $\epsilon : 0.1, 0.3, 0.5$ and 0.7. To try also other values of ϵ would mean twice the number of the test cases which would require too much computational resources. We also didn't consider values of ϵ higher than 0.7 because values

[2] http://www.mathworks.com/help/stats/ksdensity.html

higher than that bring too much randomness into the selection process in our opinion. Both *MultiEA* and *Epsilon-greedy* we tested with the same number of instances in the portfolio as the *MetaMax* strategies.

Although we initially wanted to run all of the tests with the generous budget of $100000 \times s$ fitness function evaluations and to repeat each single test of the particular strategy configuration on a given problem instance 25 times, we found out that in some of the tasks this would need very large number of computational resources and would take larger amount of time that we could afford. Therefore, after we've come to this conclusion while running the tests of the TSP task, each of the remaining optimization problems we tested with the fitness function budget lowered to $1000 - 10000 \times s$ evaluations, according to the difficulty of each task instance. We also changed the number of runs for a particular test to 10.

4.3 Results

For each of the optimization tasks we provide a section describing the results we observed. To compare the behaviour of the tested meta-strategies we use the convergence curves of each of the strategy with its best settings, determined by the value of the Aggregated performance index (API). The lower the API value the better the convergence of that particular strategy. The convergence curves are shown in the log scale. The `x`-axis shows the number of fitness function evaluations used and the `y`-axis shows the distance from the optimum or the best known solution for the task instances when the optimum is not known. In each plot we include just one of the MultiEA variants and one of the MetaMax or MetaMax(∞) with the best API value.

We provide two different tables. One shows API values of all the meta-strategies with their best parameters. The other table shows values of API for constant restarting strategy and all of the strategies with the fixed portfolio sizes. Portfolio algorithms in each of the horizontal sections of such API table have the portfolio size that is similar to the number of restarts done by the constant restarting strategy, in each of the problem instances. Columns of the API tables represent the instances of optimization tasks tested.

Labels of the column consist of the name of the task (using two non-standard abbreviations: `km` and `ss` to denote the k-means and single-swap algorithms, respectively) and the number indicating the size of the task instance, for example the number of the cities in the case of TSP. The same labels are also used in the captions of the figures.

Labels of the rows consist of the name of the meta-strategy (where the `eps` denotes the Epsilon-greedy strategy) and the number that shows the portfolio size or the value of the restarting coefficient in the case of restarting strategies. The letter `k` is sometimes used as an abbreviation of thousands to conserve space (e.g. `5k` represents the value of `5000`).

4.3.1 Travelling salesman problem

Figures 1 and 2 show the convergence of different meta-strategies tested on the Travelling salesman problem (TSP) instances with 42 and 175 cities[3]. Of all the TSP instances we used, the instance with 42 cities was the only one for which the optimum was reached.

From the figures we can see that MultiEA and Metamax strategies converge slower than restarting strategies and Epsilon-greedy. In fact, in the beginning the convergence curve of the Epsilon-greedy strategy looks very similar to the convergence of the restarting strategies.

The only strategies that were able to find the optimum in the instance with 42 cities are the stagnation and constant restarting strategies and the meta-max. These three strategies also exhibit the best convergence trend of all meta-strategies tested and are the ones that seem to have the potential to reach better solutions with larger evaluations budget also in more difficult TSP instances.

On the other hand, MultiEA, MetaMax(k) and Epsilon-greedy are the strategies that cease to converge more after about $10^4 - 10^5$ and $10^5 - 10^6$ evaluations in instances with 42 and 175 cities respectively. This is to be expected, as these three strategies are the ones with the fixed number of algorithms in their portfolios. As each of the search instances reaches its local optimum, the meta-strategy converges as well. In larger instances, namely the ones with the 535 and 1032 cities, MetaMax(k) seems to have a potential to find a little bit better solution than MultiEA or Epsilon-greedy, but we would need larger evaluations budget to confirm this.

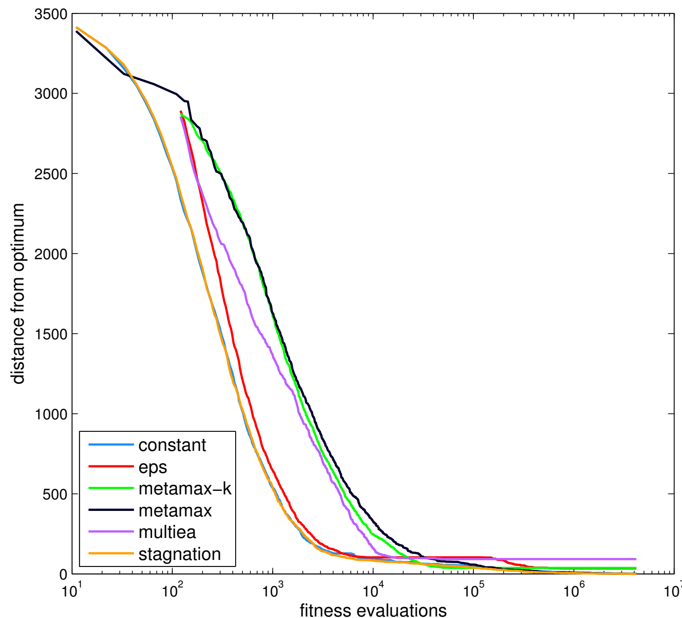

Figure 1 Convergence curves for the best strategies on tsp-42.

[3]See appendix B.1 for convergence curves of more TSP instances.

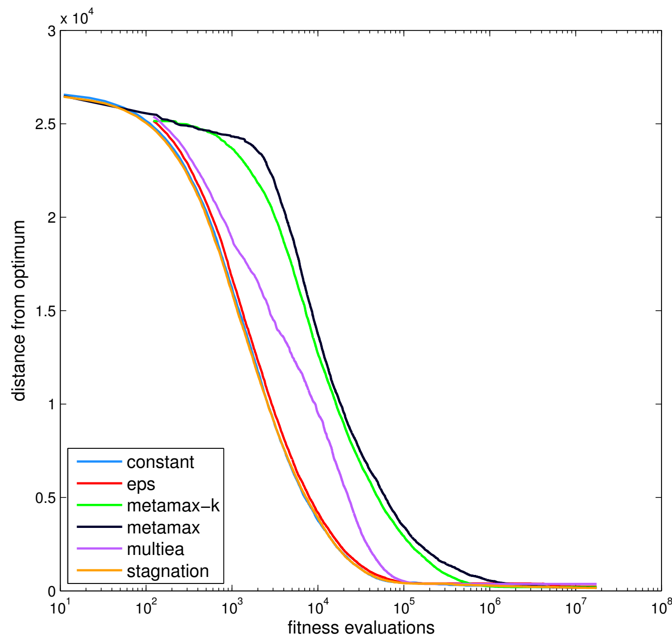

Figure 2 Convergence curves for the best strategies on tsp-175.

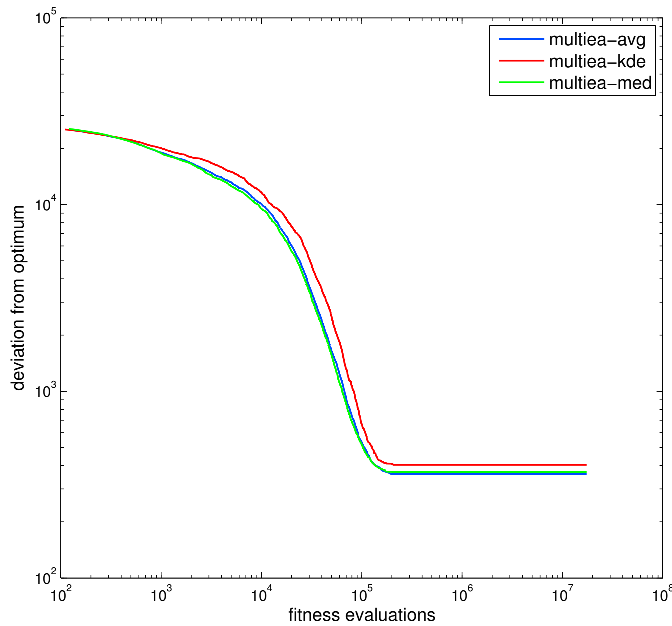

Figure 3 MultiEA strategies with different prediction functions on tsp-175.

Table 2 compares the APIs of different portfolio algorithms with fixed number of local search instances in their portfolios and constant restarting strategy. With the increasing complexity of the TSP task the optimal value of restart coefficient for constant restarting strategy increases as well. In all TSP instances this strategy has the best API value, which confirms that it converges the most quickly. Stagnation-detecting restarting strategy is not so sensitive to the proper setting of the restarting coefficient as the values of API for this strategy are similar across wider spectrum of portfolio sizes[4].

With smaller portfolio sizes, when increasing the number of the cities in the TSP instance, MetaMax(k) converges at a slower rate than MultiEA. However, the portfolio size of 100 algorithm instances is a point at which this trend changes and MetaMax(k) starts to exhibit better API values than MultiEA even for larger TSP instances.

Figure 3 shows the performance of MultiEA strategies with different types of prediction. When we compare them, we see that the strategy that uses bootstrapping from the probability estimated by kernel density estimator (kde), is always a little bit behind the other ones that use just the average or median. Differences betwen the latter two are negligible, with the average slightly better in smaller and median in larger instances of TSP. This is also shown in the table 2 where the APIs of the `multiea-kde` strategies are higher than `multiea-avg` or `multiea-med` in almost all cases.

	tsp-42	tsp-175	tsp-535	tsp-1032
constant-best	5.85	7.72	9.19	9.86
eps-best	6.68	8.02	9.30	10.01
metamax-k-best	7.38	9.50	11.02	11.76
multiea-kde-best	7.63	9.02	10.14	10.85
multiea-avg-best	7.44	8.77	9.87	10.44
multiea-med-best	7.58	8.73	9.83	10.40
stagnation-best	5.82	7.72	9.17	9.85
metamax-inf-best	7.00	9.61	11.26	11.99
metamax-best	6.99	9.59	11.25	11.96

Table 1 APIs of the best settings of tested strategies for TSP.

[4]See the table with more API values in appendix A.1. It is more evident in the larger TSP instances.

	tsp-42	tsp-175	tsp-535	tsp-1032
constant-5k	6.10	7.75	9.19	9.86
eps-20	6.99	8.15	9.35	10.04
metamax-k-20	7.62	9.75	11.30	12.10
multiea-kde-20	8.00	9.51	10.75	11.33
multiea-avg-20	7.96	9.20	10.28	10.82
multiea-med-20	7.97	9.21	10.25	10.80
constant-2k	6.00	7.74	9.21	9.95
eps-50	7.44	8.43	9.46	10.13
metamax-k-50	7.92	10.02	11.59	12.44
multiea-kde-50	8.63	10.34	11.53	12.05
multiea-avg-50	8.59	9.94	10.97	11.48
multiea-med-50	8.68	9.87	10.91	11.41
constant-1k	5.90	7.73	9.19	10.11
eps-100	7.91	8.68	9.59	10.23
metamax-k-100	8.20	10.18	11.78	12.64
multiea-kde-100	9.32	10.94	12.12	12.65
multiea-avg-100	9.06	10.52	11.56	12.04
multiea-med-100	9.21	10.47	11.46	11.93
constant-500	5.91	7.72	9.36	10.31
eps-200	8.39	9.00	9.78	10.37
metamax-k-200	8.59	10.30	11.89	12.80
multiea-kde-200	9.85	11.54	12.73	13.29
multiea-avg-200	9.65	11.08	12.17	12.63
multiea-med-200	9.72	11.08	12.07	12.53
constant-200	5.86	7.97	9.88	10.83
eps-500	9.32	9.52	10.11	10.63
metamax-k-500	9.18	10.60	12.02	12.92
multiea-kde-500	10.58	12.36	13.63	14.13
multiea-avg-500	10.34	11.89	12.99	13.46
multiea-med-500	10.45	11.87	12.91	13.34

Table 2 APIs of portfolio algorithms and fixed restarting strategy for TSP.

4.3.2 K-means clustering

The K-means clustering is different to other combinatorial optimization tasks that we used for testing the meta-strategies in that that the k-means algorithm converges relatively quickly in comparison to the other local search algorithms. This is also reflected in the fact that the optima were reached for all but the last problem instance.

Figures 4 and 5 show the convergence of meta-strategies on the k-means clustering task instances with 18 and 29 clusters respectively[5].

As was expected, the strategies with unbounded number of local search instances achieve better results than those with the fixed portfolio size. Portfolios with the fixed number of local search algorithms converge very quickly and then the exploration of the state-space ends, while the unbounded portfolios are able to fully utilize the good k-means heuristic and explore much of the promisingly looking regions of the solution space.

In this clustering task MetaMax(∞) shows slightly better convergence rate than MetaMax. Even more interestingly, the convergence rate of the best MetaMax(∞) strategy surpasses even that of the constant and stagnation-detecting restart strategies. In the largest tested clustering instance by far the worst results are delivered by the Epsilon-greedy strategy.

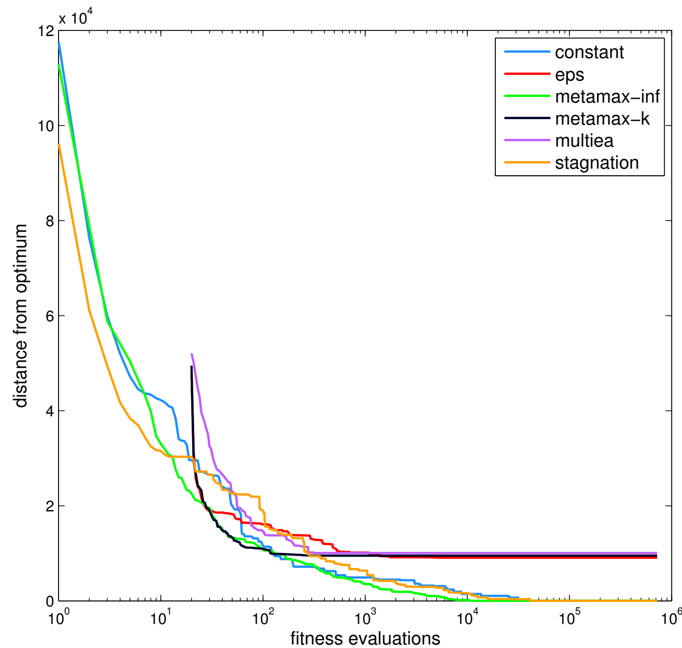

Figure 4 Convergence curves for the best strategies on km-18.

[5] See appendix B.2 for more instances.

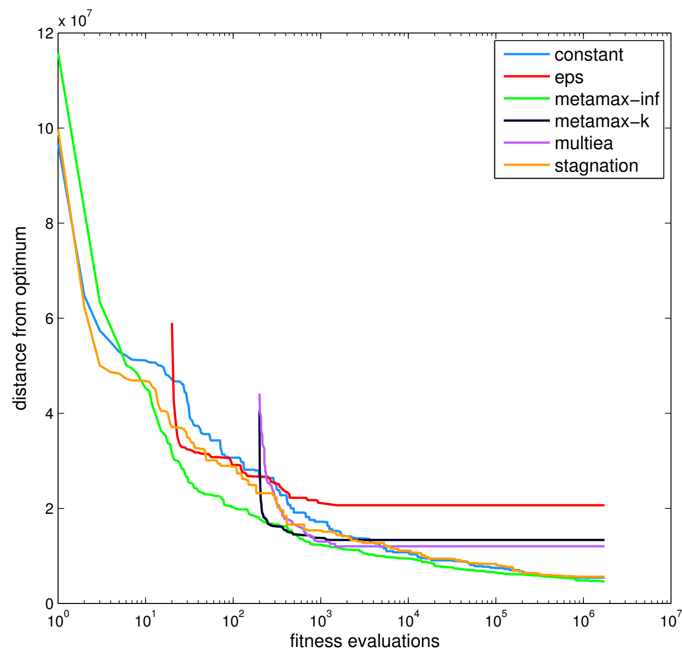

Figure 5 Convergence curves for the best strategies on km-29.

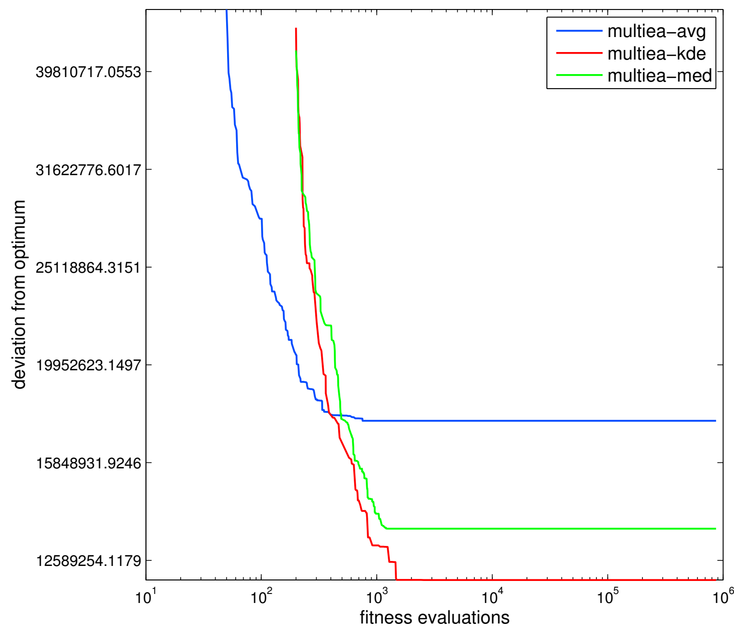

Figure 6 MultiEA strategies with different prediction functions on km-29.

In contrast with the TSP task, in k-means clustering the variant of MultiEA that uses kernel density estimator (KDE) reaches better results and with the optimal protfolio size even converges faster than the variants using the average or median. The figure 6 compares the convergence rate of these MultiEA strategies with the portfolio sizes that achieve the best API values. The `multiea-kde` surpasses both `multiea-avg` and `multiea-med`. This could be due to the fact that predicting the future performance of the local search instance by bootstrapping from the probability density estimated by KDE leaves more space for exploration than more simple approaches that predict it using median or average of the extrapolated fitness values.

As we noted before, in the k-means task the best MetaMax(∞) strategies for each tested instance converge a little bit faster than MetaMax strategies. This can also be seen in the table 3, which shows APIs of the best strategies for each of the k-means instances. This is probably because of the informed search, backed by k-means heuristics, which is fast and so the additional runs of the best local search instances performed by MetaMax aren't worth the extra evaluations of the fitness function.

With such quickly converging local search algorithm as this one it is also not neccessary to increase the restarting coefficient so much to improve the convergency of the constant restarting strategy. For example, the differences between the APIs of the `constant-200` and `constant-2000` in the largest instance, shown in the table 4, are not so large as it was e.g. in the case of the TSP.

	km-5	km-10	km-18	km-29
constant-best	0.77	1.34	1.94	4.75
eps-best	2.38	3.23	4.97	8.54
metamax-k-best	2.37	3.22	5.32	9.02
multiea-kde-best	2.42	3.43	4.93	8.57
multiea-avg-best	2.43	2.78	5.11	8.75
multiea-med-best	2.40	3.23	5.26	8.97
stagnation-best	0.62	1.28	1.99	4.66
metamax-inf-best	0.76	1.28	1.79	4.27
metamax-best	0.89	1.32	1.80	4.34

Table 3 APIs of the best settings of tested strategies for k-means.

	km-5	km-10	km-18	km-29
constant-/500/1k/2k/2k/	1.04	1.45	2.17	4.76
eps-10	2.38	3.39	5.10	9.47
metamax-k-10	2.37	3.55	5.41	9.56
multiea-kde-10	2.42	3.51	5.10	8.85
multiea-avg-10	2.43	2.78	5.25	9.39
multiea-med-10	2.40	3.34	5.32	9.05
constant-/200/500/1k/1k/	0.77	1.49	1.94	4.98
eps-20	3.02	3.23	4.97	8.54
metamax-k-20	3.01	3.22	5.32	9.12
multiea-kde-20	3.03	3.43	4.93	9.58
multiea-avg-20	3.05	3.35	5.11	9.51
multiea-med-20	3.05	3.23	5.26	9.25
constant-/100/200/500/500/	0.85	1.34	2.11	5.01
eps-50	3.92	4.06	5.27	9.04
metamax-k-50	3.92	3.95	5.61	9.37
multiea-kde-50	3.93	4.09	5.77	9.35
multiea-avg-50	3.93	4.09	5.30	8.75
multiea-med-50	3.93	4.08	5.37	9.22
constant-/—/100/200/200/	—	1.44	1.98	4.80
eps-100	4.61	4.67	5.65	8.86
metamax-k-100	4.61	4.62	5.80	9.15
multiea-kde-100	4.61	4.70	5.84	9.27
multiea-avg-100	4.61	4.73	5.82	8.80
multiea-med-100	4.62	4.73	5.60	9.19
constant-/—/—/100/100/	—	—	1.97	4.95
eps-200	5.30	5.35	6.01	9.01
metamax-k-200	5.30	5.31	6.07	9.02
multiea-kde-200	5.30	5.38	6.37	8.57
multiea-avg-200	5.30	5.38	6.02	8.84
multiea-med-200	5.31	5.39	5.99	8.97
eps-500	6.22	6.23	6.66	9.14
metamax-k-500	6.21	6.22	6.60	9.57
multiea-kde-500	6.22	6.26	6.74	9.34
multiea-avg-500	6.22	6.27	6.69	9.03
multiea-med-500	6.22	6.26	6.71	9.31

Table 4 APIs of portfolio algorithms and fixed restarting strategy for k-means.

4.3.3 Single-swap clustering

The single-swap algorithm, in contrast with k-means, doesn't use any clever heuristic to determine the direction of the search. This results in strategies having low convergence rate and finding solutions with not so good fitness as in the case of k-means.

Figures 7 and 8 show the same instances of a clustering task that were displayed for the k-means task, with 18 and 29 clusters and plots showing the average convergence rates for the settings of strategies that yield the best values of API. We can see that none of the displayed strategies reached the optima.

Epsilon-greedy strategy converges almost at the same rate as the restarting strategies and the MultiEA strategy shows similar rate, albeit a little bit slower in the beginning. Both also converge faster than any MetaMax strategy.

Using the single-swap algorithm, MetaMax strategy exhibits a little bit higher convergence rate than MetaMax(∞), except for the first instance with 5 clusters. This is opposite to the k-means, as well as is the fact that in the largest clustering instance the MetaMax(k) doesn't seem to achieve better results than the restarting strategies, although admittedly, larger budget of fitness function evaluations would be needed to confirm this.

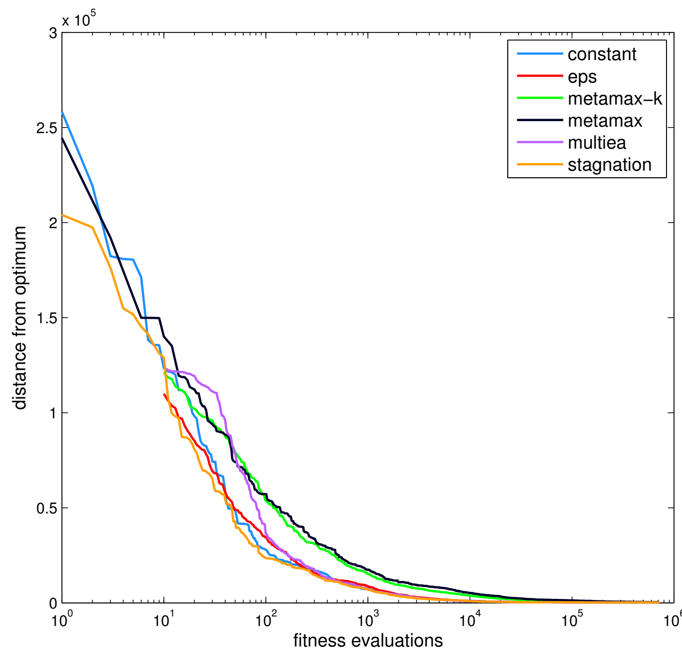

Figure 7 Convergence curves for the best strategies on ss-18.

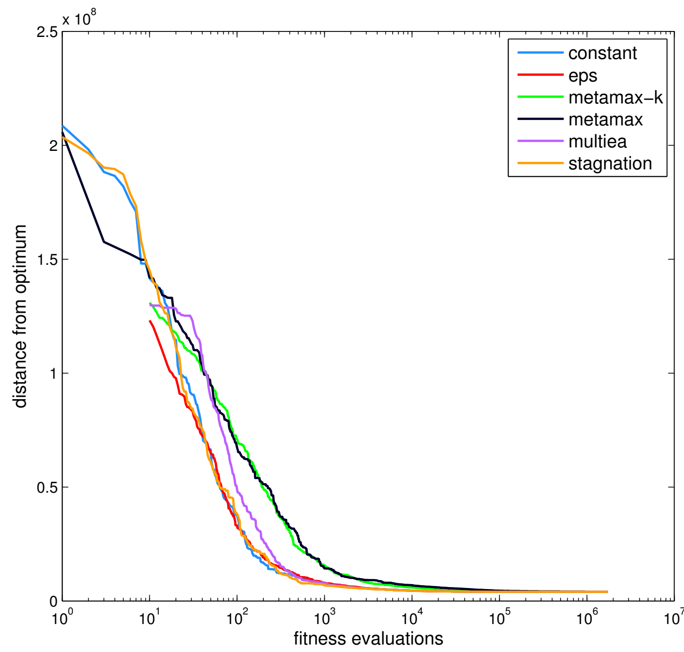

Figure 8 Convergence curves for the best strategies on ss-29.

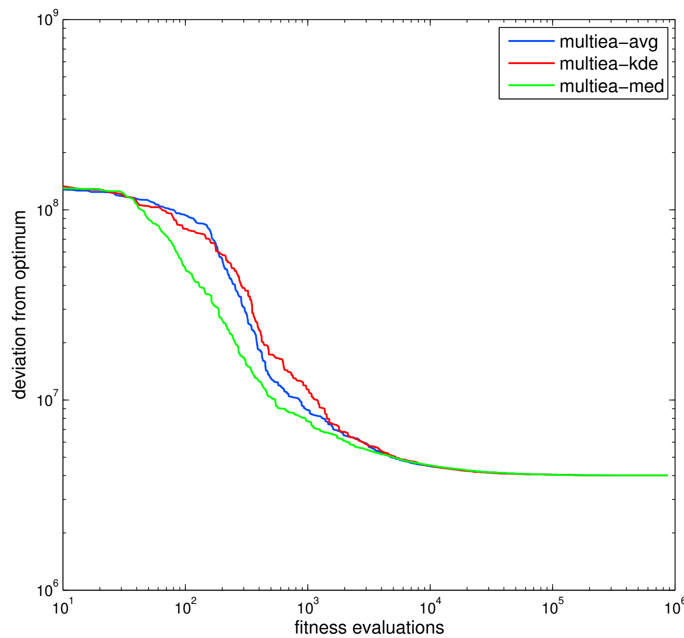

Figure 9 MultiEA strategies with different prediction functions on ss-29.

From the figure 9 which shows the convergence curves of different MultiEA strategies on the largest single-swap clustering instance, we can see that the convergence curves shown here resemble more the ones from the TSP than the ones from the k-means problem. However, in this task the `eps-med` strategy has the highest convergence rate, especially in the beginning phase and by fair margin. From the table 6 we see that this phenomenon occurs in all but the smallest problem instance.

The table 5 shows the best APIs reached by a particular strategy in a single-swap tasks. Comparing restarting strategies, the constant restarting strategy has better API value only in the smallest clustering instance. In all other instances, the more difficult ones, stagnacy-detecting restarting converges faster. MetaMax strategies with unbounded number of algorithms in their portfolios dominate the MetaMax(k) strategy, but with much smaller improvements over it as it was in the case of the k-means clustering. Except for the smallest clustering instance, MetaMax has slightly better convergence than MetaMax(∞), which is the opposite to the k-means. Overall we can tell that the restarting strategies are the ones most performant here.

In the table 6 we see the similar trend as was noted before - MetaMax(k) seems to be using larger portfolios more efficiently than the MultiEA strategy, more so with increasing the number of clusters in a single-swap task. Take for example the task *ss-18*. While with the portfolio comprised of 10 local search instances the MetaMax(k) has the API value of 4.17 whereas the best MultiEA variant has the API of 3.91, with increasing portfolio size the API values of MultiEA strategies increase faster than that of MetaMax(k). With the portfolios of 100 algorithms they are equal and further on MultiEA strategies have higher API values than MetaMax(k).

	ss-5	ss-10	ss-18	ss-29
constant-best	1.70	2.48	2.84	3.89
eps-best	3.15	3.53	3.77	4.47
metamax-k-best	3.27	3.82	4.17	5.22
multiea-kde-best	3.22	3.77	4.13	5.04
multiea-avg-best	3.27	3.81	4.11	4.98
multiea-med-best	3.25	3.66	3.91	4.66
stagnation-best	1.83	2.36	2.69	3.85
metamax-inf-best	2.09	2.92	3.40	4.82
metamax-best	2.15	2.83	3.37	4.72

Table 5 APIs of the best settings of tested strategies for single-swap.

	ss-5	ss-10	ss-18	ss-29
constant-/500/1k/2k/2k/	2.11	2.48	3.05	3.89
eps-10	3.15	3.53	3.77	4.47
metamax-k-10	3.27	3.82	4.17	5.22
multiea-kde-10	3.22	3.77	4.13	5.04
multiea-avg-10	3.27	3.81	4.11	4.98
multiea-med-10	3.25	3.66	3.91	4.66
constant-/200/500/1k/1k/	2.23	2.52	2.85	3.94
eps-20	3.68	3.94	4.23	4.78
metamax-k-20	3.81	4.28	4.63	5.56
multiea-kde-20	3.75	4.38	4.68	5.51
multiea-avg-20	3.82	4.29	4.61	5.50
multiea-med-20	3.77	4.18	4.39	5.06
constant-/100/200/500/500/	1.70	2.63	3.20	4.00
eps-50	4.41	4.60	4.82	5.20
metamax-k-50	4.54	4.90	5.28	6.08
multiea-kde-50	4.58	5.07	5.41	6.24
multiea-avg-50	4.57	5.03	5.39	6.16
multiea-med-50	4.52	4.86	5.13	5.71
constant-/—/100/200/200/	—	2.71	3.00	4.08
eps-100	5.03	5.17	5.32	5.64
metamax-k-100	5.11	5.45	5.69	6.31
multiea-kde-100	5.19	5.65	5.95	6.84
multiea-avg-100	5.19	5.62	5.96	6.72
multiea-med-100	5.11	5.49	5.69	6.20
constant-/—/—/100/100/	—	—	2.93	4.17
eps-200	5.66	5.77	5.87	6.12
metamax-k-200	5.72	5.98	6.22	6.70
multiea-kde-200	5.80	6.24	6.60	7.38
multiea-avg-200	5.79	6.20	6.53	7.32
multiea-med-200	5.75	6.08	6.30	6.79
eps-500	6.53	6.60	6.66	6.80
metamax-k-500	6.57	6.74	6.91	7.27
multiea-kde-500	6.67	7.04	7.42	8.21
multiea-avg-500	6.67	7.03	7.36	8.11
multiea-med-500	6.62	6.93	7.15	7.58

Table 6 APIs of portfolio algorithms and fixed restarting strategy for single-swap.

4.3.4 Warehouse location problem

The Warehouse location problem (WLP) is a combinatorial optimization task that is hard to solve with a simple local search algorithm, because many of the generated solutions will inevitably violate some of the constraints the task defines.

Figures 10 and 11 show the average convergence curves of tested meta-strategies with the best settings determined by the smallest API value. Portfolios with unbounded number of algorithms as well as the restarting strategies continue to perform best. With the given budget of fitness evaluations, MetaMax(∞) delivers the best solutions in both displayed wlp instances. It is however closly followed by constant and stagnacy-detecting restarting strategies. Regarding the convergence rate, both variants of Metamax converge at the slowest pace up to about 10^5 fitness evaluations, where the convergence rate of the other strategies considerably slows down while the MetaMax(∞) continues to converge at almost the same rate, even higher than constant or stagnation-detecting restarting strategies. While the MultiEA strategy in the beginning converges almost as quickly as the restarting strategies, in the end it performs worst.

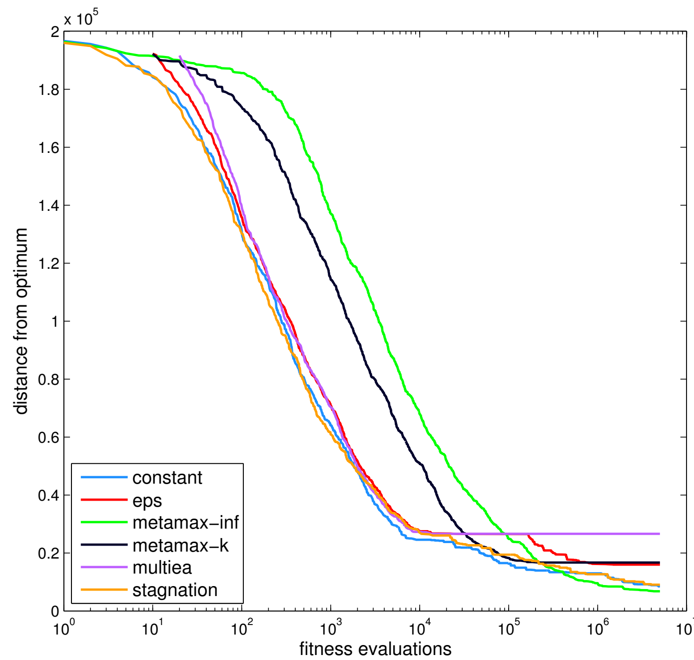

Figure 10 Convergence curves for the best strategies on wlp-50.

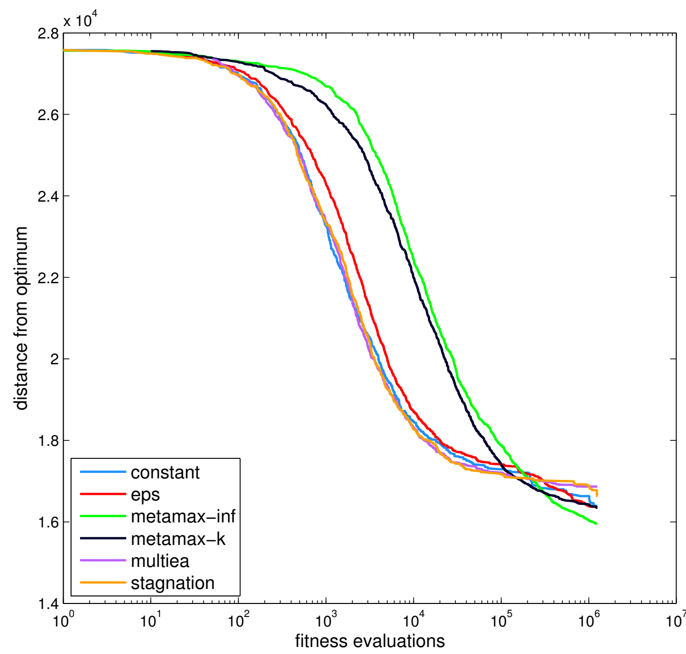

Figure 11 Convergence curves for the best strategies on wlp-500.

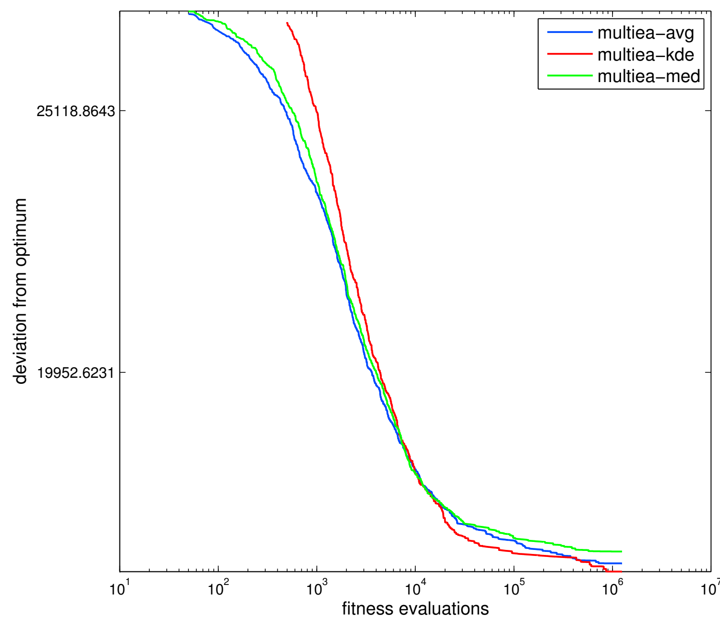

Figure 12 MultiEA strategies with different prediction functions on wlp-500.

In WLP instances with 50 and 200 warehouses, there are practically no differences in the convergence rate of the best MultiEA strategies with the different means of predicting the future performance of the algorithm instance. Their convergence curves look almost the same. In the WLP instance with 500 warehouses there's a small difference though. While the `multiea-avg` and `multiea-med` have the best API values with the portfolio size of 50 algorithms, in the case of `multiea-kde` the best API value is achieved with 500 algorithms. However, given that the `multiea-kde` uses 50 times larger portfolio, the difference in the convergence that is visible in the figure 12 seems minor.

When we compare the best API values achieved by the strategies, shown in the table 7, we see that restarting strategies converge faster, with the constant restarting having slightly better API values than stagnacy-detecting restarting strategy. MetaMax doesn't make any improvement over the convergence of MetaMax(∞). In fact, it performs slightly worse.

In the table 8, that compares the constant restarting strategy and various strategies with fixed portfolio sizes, we can see similar trends as in some of the other problems tested. Constant restarting strategy converges the most quickly. Epsilon-greedy shows better convergence than MultiEA or MetaMax(k). Note that in the WLP tasks with 200 and 500 warehouses the differences between the best and the worst APIs shown in the table for any portfolio size are quite small.

	wlp-50	wlp-200	wlp-500
constant-best	6.86	15.96	17.73
eps-best	7.54	16.12	17.80
metamax-k-best	8.60	16.43	18.01
multiea-kde-best	8.46	16.33	17.87
multiea-avg-best	8.00	16.24	17.87
multiea-med-best	8.19	16.27	17.90
stagnation-best	6.90	16.00	17.87
metamax-inf-best	8.84	16.40	18.03
metamax-best	8.89	16.42	18.05

Table 7 APIs of the best settings of tested strategies for WLP.

	wlp-50	wlp-200	wlp-500
constant-/—/1k/200/	—	15.96	17.73
eps-10	7.54	16.12	17.80
metamax-k-10	8.60	16.43	18.01
multiea-kde-10	8.46	16.34	17.89
multiea-avg-10	8.00	16.30	17.90
multiea-med-10	8.19	16.46	17.93
constant-/5k/500/100/	7.45	16.01	17.85
eps-20	7.55	16.12	17.89
metamax-k-20	8.89	16.52	18.14
multiea-kde-20	8.57	16.33	17.93
multiea-avg-20	8.00	16.24	17.99
multiea-med-20	8.19	16.27	17.93
constant-/2k/200/—/	7.05	16.04	—
eps-50	7.84	16.15	17.84
metamax-k-50	9.09	16.43	18.10
multiea-kde-50	8.73	16.38	17.88
multiea-avg-50	8.36	16.46	17.87
multiea-med-50	8.34	16.39	17.90
constant-/1k/100/—/	7.22	16.09	—
eps-100	7.93	16.21	17.92
metamax-k-100	9.14	16.47	18.21
multiea-kde-100	9.15	16.59	17.98
multiea-avg-100	8.85	16.49	18.01
multiea-med-100	8.47	16.56	17.98
constant-/500/—/—/	7.02	—	—
eps-200	8.26	16.27	17.98
metamax-k-200	9.08	16.45	18.20
multiea-kde-200	9.62	16.40	18.05
multiea-avg-200	9.22	16.27	18.07
multiea-med-200	8.91	16.36	17.94
constant-/200/—/—/	6.86	—	—
eps-500	8.71	16.51	17.97
metamax-k-500	9.25	16.50	18.18
multiea-kde-500	9.99	16.44	17.87
multiea-avg-500	9.77	16.70	17.97
multiea-med-500	9.38	16.47	18.07

Table 8 APIs of portfolio algorithms and fixed restarting strategy for WLP.

5 Conclusion

The goal of this work was to compare the performance and efficiency of selected portfolio algorithms for combinatorial optimization with commonly used restarting strategies for local search.

We tested two portfolio algorithms that use a different approaches to build an online schedule, MetaMax and MultiEA, two commonly used restarting strategies, the one with constant period of restarts and the other one that restarts the search when it detects that the quality of the solutions found by the search process isn't improving anymore (the search process stagnates). We also included the Epsilon-greedy strategy whose schedule is known in advance. Some of the strategies have more variations. We studied three versions of MetaMax. MetaMax(k) with fixed number of algorithm instances in the portfolio, MetaMax(∞) that adds a new algorithm instance to its portfolio in every iteration and the MetaMax itself, which is a modification of MetaMax(∞) that performs some additional evaluations of promising algorithm instances. We also tested two new modifications of MultiEA strategy, that replaces the usage of kernel density estimator (KDE) in its prediction process with median or average.

All these strategies we tested on four combinatorial optimization tasks. The first one is the Travelling salesman problem (TSP). We also used two different clustering tasks. One uses relatively fast k-means algorithm and the other one uses randomized local search algorithm, that we call single-swap, which converges slower. The last task is the Warehouse location problem (WLP), a combinatorial optimization task with constraints.

For comparing the convergence of the strategies and also selecting the best values of their parameters for every instance of an optimization task, we used the measure called Aggregated Performance Index (API).

We found out that the constant and stagnation-detecting restarting strategies give very similar results, both in the terms of convergence and the quality of the solutions found. In the case of TSP the stagnation-detecting strategy seemed to be less sensitive to the proper setting of its parameters, giving better API values than the constant restarting on a wider range of values of its parameter, especially on larger problem instances. Similar behaviour was seen in the WLP task.

Comparing the three MetaMax variants, MetaMax(k) is the one that clearly gives the worst results, although it often has steeper convergence curve in the beginning than the other MetaMax variants that use unbounded number of algorithm instances. From these two, MetaMax exhibits slightly better convergence in the TSP and single-swap clustering and MetaMax(∞) in the other two tasks. We speculate that in some cases, when the convergence of the local search is too fast the extra fitness evaluations that the MetaMax gives to some perspective algorithm instances won't pay out. This could be the case of the k-means. In

the WLP task, the problem probably is, that given an algorithm instance that looks like it could deliver some good solution in the future doesn't guarantee that this solution will also satisfy the constraints defined by the task. Therefore many of the extra fitness evaluations are probably used on some promisingly-looking algorithm instances that in the end turn out to find solutions violating the constraints. In this case, the more conservative strategy MetaMax(∞) delivers a little bit better results.

As for the different variants of the MultiEA strategy, they all gave similar results except for the k-means task, where the MultiEA variant using KDE showed better convergence rate in the largest problem instance. It may be the case that when the optimization uses efficient heuristic, the KDE process which gives more space for exploration than the variants using average or median has an advantage. This can be explored further in the future. However, we think that at least with difficult blackbox combinatorial optimization tasks, like the other ones we tested, it is perfectly possible to replace the process of bootstrapping from the probability density estimated by KDE with more simple and direct median or average of the historical fitness values.

From the strategies with fixed number of algorithms in the portfolio, the Epsilon-greedy has the best API values most of the time. In the beginning of the search process it converges fast, sometimes on par with the restarting strategies. MetaMax(k) exhibits the worst convergence rates in both TSP and single-swap problems. However, regarding the solution quality, it is one of the better performing fixed portfolios. On the other side, the MultiEA algorithm is the worst performant both in TSP and WLP tasks. In fact, this strategy dissappoints us a bit, especially when compared with much more simple Epsilon-greedy strategy that gives similar or better results with higher convergence rate overall.

One of our main questions that we wanted to find an answer for is how will these portfolio algorithms compare with the restarting strategies. We must say that when it comes to the solution quality, the only portfolio algorithms that could be compared with the restarting strategies are the unbounded versions of MetaMax strategies, the algorithms MetaMax and MetaMax(∞). In all of the combinatorial optimization tasks we tested, only these were able to obtain the solution quiality similar to those of restarting strategies within the provided budget of fitness function evaluations. However both of these algorithms usually converge slower than the restarting strategies. In the k-means task they showed slightly better convergence than restarting strategies in large clustering instances and in the WLP task they surpassed the convergence rate of restarting strategies later in the search process. While the advantage of MetaMax is that the user doesn't have to do extensive parameter tuning to obtain good results, stagnation-detecting restarting strategy showed similar behaviour in some of the tasks. When we also consider that the implementation cost of MetaMax is higher than that of the restarting strategies, we come to conclusion that overall the restarting strategies seem to be more effective approach and the user should start with them. However, MetaMax seems to be delivering good long term results in a

combinatorial optimization tasks with constraints and it would be interesting to investigate this more in the future. Another thing that might be worth exploring is a modification of MultiEA strategy that would have an unbounded number of algorithms in its portfolio.

Bibliography

György, András and Levente Kocsis (May 2011). "Efficient Multi-start Strategies for Local Search Algorithms". In: *J. Artif. Int. Res.* 41.2, pp. 407–444.

Hahsler, Michael and Kurt Hornik (2006). "TSP – Infrastructure for the Traveling Salesperson Problem". In: *Journal of Statistical Software*.

Hansen, N. et al. (2010). *Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup*. Tech. rep. RR-7215. INRIA.

Kajml, Viktor (Jan. 2014). *Black box optimization: Restarting versus MetaMax algorithm*. Prague, Czech Republic: Czech Technical University.

Kanungo, Tapas et al. (2002). "A Local Search Approximation Algorithm for K-means Clustering". In: *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*. SCG '02. Barcelona, Spain: ACM, pp. 10–18.

Krarup, Jakob and Peter Mark Pruzan (Jan. 1983). "The simple plant location problem: Survey and synthesis". In: *European Journal of Operational Research* 12.1, pp. 36–81.

Kuehn, Alfred A. and Michael J. Hamburger (1963). *A Heuristic Program for Locating Warehouses*. Pittsburgh, Pennsylvania: Carnegie Institute of Technology.

Lin, S. and B. W. Kernighan (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". In: *Operations Research* 21.2, pp. 498–516.

Pošík, Petr (2012). *Building an Aggregated Performance Index*.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College.

Yuen, Shiu Yin, Chi Kin Chow, and Xin Zhang (2013). "Which Algorithm Should I Choose at Any Point of the Search: An Evolutionary Portfolio Approach". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO '13. Amsterdam, The Netherlands: ACM, pp. 567–574.

Appendix A

API by strategy and parameter settings

A.1 Travelling salesman problem

	tsp-42	tsp-175	tsp-535	tsp-1032
constant-100	**5.85**	8.49	10.54	11.40
constant-200	5.86	7.97	9.88	10.83
constant-500	5.91	**7.72**	9.36	10.31
constant-1k	5.90	7.73	**9.19**	10.11
constant-2k	6.00	7.74	9.21	9.95
constant-5k	6.10	7.75	**9.19**	**9.86**
stagnation-100	**5.82**	7.75	9.18	**9.85**
stagnation-200	5.89	7.73	**9.17**	9.87
stagnation-500	5.91	7.75	9.19	**9.85**
stagnation-1k	5.99	**7.72**	9.19	9.86
stagnation-2k	6.08	7.77	9.22	**9.85**
stagnation-5k	6.23	7.75	9.18	9.86

Table 9 APIs of restarting strategies for instances of TSP.

	tsp-42	tsp-175	tsp-535	tsp-1032
multiea-kde-10	**7.63**	**9.02**	**10.14**	**10.85**
multiea-kde-20	8.00	9.51	10.75	11.33
multiea-kde-50	8.63	10.34	11.53	12.05
multiea-kde-100	9.32	10.94	12.12	12.65
multiea-kde-200	9.85	11.54	12.73	13.29
multiea-kde-500	10.58	12.36	13.63	14.13
multiea-avg-10	**7.44**	**8.77**	**9.87**	**10.44**
multiea-avg-20	7.96	9.20	10.28	10.82
multiea-avg-50	8.59	9.94	10.97	11.48
multiea-avg-100	9.06	10.52	11.56	12.04
multiea-avg-200	9.65	11.08	12.17	12.63
multiea-avg-500	10.34	11.89	12.99	13.46
multiea-med-10	**7.58**	**8.73**	**9.83**	**10.40**
multiea-med-20	7.97	9.21	10.25	10.80
multiea-med-50	8.68	9.87	10.91	11.41
multiea-med-100	9.21	10.47	11.46	11.93
multiea-med-200	9.72	11.08	12.07	12.53
multiea-med-500	10.45	11.87	12.91	13.34

Table 10 APIs of MultiEA strategies for instances of TSP.

	tsp-42	tsp-175	tsp-535	tsp-1032
metamax-k-10	**7.38**	**9.50**	**11.02**	**11.76**
metamax-k-20	7.62	9.75	11.30	12.10
metamax-k-50	7.92	10.02	11.59	12.44
metamax-k-100	8.20	10.18	11.78	12.64
metamax-k-200	8.59	10.30	11.89	12.80
metamax-k-500	9.18	10.60	12.02	12.92
metamax-inf-10	7.01	**9.61**	**11.26**	**11.99**
metamax-inf-20	7.02	9.61	11.32	12.09
metamax-inf-50	**7.00**	9.66	11.38	12.13
metamax-inf-100	7.05	9.71	11.42	12.19
metamax-inf-200	7.07	9.71	11.42	12.21
metamax-inf-500	7.07	9.68	11.44	12.22
metamax-10	7.00	**9.59**	**11.25**	**11.96**
metamax-20	**6.99**	**9.59**	11.32	12.08
metamax-50	7.02	9.64	11.34	12.14
metamax-100	7.01	9.64	11.40	12.17
metamax-200	**6.99**	9.68	11.40	12.17
metamax-500	7.06	9.63	11.40	12.20

Table 11 APIs of MetaMax strategies for instances of TSP.

	tsp-42	tsp-175	tsp-535	tsp-1032
eps-10-0.1	**6.68**	**8.02**	**9.30**	**10.01**
eps-10-0.3	6.75	8.24	9.52	10.22
eps-10-0.5	7.02	8.49	9.83	10.53
eps-10-0.7	7.27	8.92	10.30	11.03
eps-20-0.1	**6.99**	**8.15**	**9.35**	**10.04**
eps-20-0.3	7.10	8.33	9.58	10.28
eps-20-0.5	7.12	8.57	9.89	10.58
eps-20-0.7	7.45	9.00	10.32	11.03
eps-50-0.1	7.50	**8.43**	**9.46**	**10.13**
eps-50-0.3	**7.44**	8.56	9.66	10.35
eps-50-0.5	7.52	8.79	9.95	10.64
eps-50-0.7	7.74	9.15	10.39	11.10
eps-100-0.1	7.97	**8.68**	**9.59**	**10.23**
eps-100-0.3	**7.91**	8.80	9.79	10.43
eps-100-0.5	7.95	8.99	10.05	10.72
eps-100-0.7	8.11	9.32	10.45	11.16
eps-200-0.1	8.57	**9.00**	**9.78**	**10.37**
eps-200-0.3	**8.39**	9.11	9.94	10.56
eps-200-0.5	8.48	9.27	10.19	10.81
eps-200-0.7	8.53	9.55	10.58	11.23
eps-500-0.1	9.49	**9.52**	**10.11**	**10.63**
eps-500-0.3	9.26	9.60	10.26	10.80
eps-500-0.5	9.27	9.79	10.46	11.02
eps-500-0.7	**9.21**	9.95	10.80	11.40

Table 12 APIs of Epsilon-greedy strategies for instances of TSP.

A.2 K-means clustering

	km-5	km-10	km-18	km-29
constant-100	0.85	1.44	1.97	4.95
constant-200	**0.77**	**1.34**	1.98	4.80
constant-500	1.04	1.49	2.11	5.01
constant-1k	0.83	1.45	**1.94**	4.98
constant-2k	1.05	1.68	2.17	4.76
constant-5k	0.92	**1.34**	1.98	**4.75**
stagnation-100	0.93	1.50	2.11	**4.66**
stagnation-200	0.71	**1.28**	2.23	4.70
stagnation-500	0.99	1.59	2.02	5.15
stagnation-1k	**0.62**	1.47	2.08	5.03
stagnation-2k	1.09	1.38	2.18	5.15
stagnation-5k	1.16	1.35	**1.99**	4.71

Table 13 APIs of restarting strategies for instances of k-means.

	km-5	km-10	km-18	km-29
multiea-kde-10	**2.42**	3.51	5.10	8.85
multiea-kde-20	3.03	**3.43**	**4.93**	9.58
multiea-kde-50	3.93	4.09	5.77	9.35
multiea-kde-100	4.61	4.70	5.84	9.27
multiea-kde-200	5.30	5.38	6.37	**8.57**
multiea-kde-500	6.22	6.26	6.74	9.34
multiea-avg-10	**2.43**	**2.78**	5.25	9.39
multiea-avg-20	3.05	3.35	**5.11**	9.51
multiea-avg-50	3.93	4.09	5.30	**8.75**
multiea-avg-100	4.61	4.73	5.82	8.80
multiea-avg-200	5.30	5.38	6.02	8.84
multiea-avg-500	6.22	6.27	6.69	9.03
multiea-med-10	**2.40**	3.34	5.32	9.05
multiea-med-20	3.05	**3.23**	**5.26**	9.25
multiea-med-50	3.93	4.08	5.37	9.22
multiea-med-100	4.62	4.73	5.60	9.19
multiea-med-200	5.31	5.39	5.99	**8.97**
multiea-med-500	6.22	6.26	6.71	9.31

Table 14 APIs of MultiEA strategies for instances of k-means.

	km-5	km-10	km-18	km-29
metamax-k-10	**2.37**	3.55	5.41	9.56
metamax-k-20	3.01	**3.22**	**5.32**	9.12
metamax-k-50	3.92	3.95	5.61	9.37
metamax-k-100	4.61	4.62	5.80	9.15
metamax-k-200	5.30	5.31	6.07	**9.02**
metamax-k-500	6.21	6.22	6.60	9.57
metamax-inf-10	**0.76**	1.39	**1.79**	4.35
metamax-inf-20	0.91	**1.28**	1.90	4.39
metamax-inf-50	0.90	1.39	1.87	**4.27**
metamax-inf-100	0.86	1.43	1.90	4.30
metamax-inf-200	0.89	1.42	1.87	4.40
metamax-inf-500	0.92	1.35	1.85	4.28
metamax-10	0.91	**1.32**	1.93	**4.34**
metamax-20	0.96	1.56	1.94	4.46
metamax-50	1.05	1.38	**1.80**	4.35
metamax-100	0.90	1.41	1.89	4.45
metamax-200	0.97	1.43	1.87	4.47
metamax-500	**0.89**	1.39	1.86	4.40

Table 15 APIs of MetaMax strategies for instances of k-means.

	km-5	km-10	km-18	km-29
eps-10-0.1	2.59	3.60	5.16	10.26
eps-10-0.3	2.44	3.48	5.29	9.91
eps-10-0.5	2.40	3.65	5.49	10.03
eps-10-0.7	**2.38**	**3.39**	**5.10**	**9.47**
eps-20-0.1	3.09	3.52	**4.97**	**8.54**
eps-20-0.3	3.04	3.31	5.37	9.49
eps-20-0.5	**3.02**	**3.23**	5.23	9.16
eps-20-0.7	3.03	3.85	5.11	9.36
eps-50-0.1	3.95	4.24	5.58	9.10
eps-50-0.3	**3.92**	4.11	5.67	9.42
eps-50-0.5	3.93	**4.06**	**5.27**	**9.04**
eps-50-0.7	3.93	4.07	5.45	9.11
eps-100-0.1	4.62	4.81	**5.65**	**8.86**
eps-100-0.3	4.62	4.76	5.72	8.89
eps-100-0.5	**4.61**	4.71	5.77	9.02
eps-100-0.7	**4.61**	**4.67**	5.79	9.21
eps-200-0.1	5.31	5.44	6.14	9.14
eps-200-0.3	5.31	5.38	6.15	9.13
eps-200-0.5	**5.30**	**5.35**	**6.01**	9.54
eps-200-0.7	**5.30**	**5.35**	6.11	**9.01**
eps-500-0.1	6.23	6.27	6.84	9.31
eps-500-0.3	**6.22**	6.25	**6.66**	**9.14**
eps-500-0.5	**6.22**	6.24	6.69	9.20
eps-500-0.7	**6.22**	**6.23**	6.76	9.28

Table 16 APIs of Epsilon-greedy strategies for instances of k-means.

A.3 Single-swap clustering

	ss-5	ss-10	ss-18	ss-29
constant-100	**1.70**	2.71	2.93	4.17
constant-200	2.23	2.63	3.00	4.08
constant-500	2.11	2.52	3.20	4.00
constant-1k	1.77	**2.48**	2.85	3.94
constant-2k	1.90	2.68	3.05	**3.89**
constant-5k	2.05	2.63	**2.84**	4.08
stagnation-100	1.93	2.70	3.01	4.11
stagnation-200	2.01	2.84	2.99	4.09
stagnation-500	1.88	2.46	**2.69**	3.97
stagnation-1k	1.91	**2.36**	2.74	3.87
stagnation-2k	**1.83**	2.55	3.03	**3.85**
stagnation-5k	2.12	2.49	2.74	3.96

Table 17 APIs of restarting strategies for instances of single-swap.

	ss-5	ss-10	ss-18	ss-29
multiea-kde-10	**3.22**	**3.77**	**4.13**	**5.04**
multiea-kde-20	3.75	4.38	4.68	5.51
multiea-kde-50	4.58	5.07	5.41	6.24
multiea-kde-100	5.19	5.65	5.95	6.84
multiea-kde-200	5.80	6.24	6.60	7.38
multiea-kde-500	6.67	7.04	7.42	8.21
multiea-avg-10	**3.27**	**3.81**	**4.11**	**4.98**
multiea-avg-20	3.82	4.29	4.61	5.50
multiea-avg-50	4.57	5.03	5.39	6.16
multiea-avg-100	5.19	5.62	5.96	6.72
multiea-avg-200	5.79	6.20	6.53	7.32
multiea-avg-500	6.67	7.03	7.36	8.11
multiea-med-10	**3.25**	**3.66**	**3.91**	**4.66**
multiea-med-20	3.77	4.18	4.39	5.06
multiea-med-50	4.52	4.86	5.13	5.71
multiea-med-100	5.11	5.49	5.69	6.20
multiea-med-200	5.75	6.08	6.30	6.79
multiea-med-500	6.62	6.93	7.15	7.58

Table 18 APIs of MultiEA strategies for instances of single-swap

	ss-5	ss-10	ss-18	ss-29
metamax-k-10	**3.27**	**3.82**	**4.17**	**5.22**
metamax-k-20	3.81	4.28	4.63	5.56
metamax-k-50	4.54	4.90	5.28	6.08
metamax-k-100	5.11	5.45	5.69	6.31
metamax-k-200	5.72	5.98	6.22	6.70
metamax-k-500	6.57	6.74	6.91	7.27
metamax-inf-10	2.11	3.02	**3.40**	**4.82**
metamax-inf-20	2.25	**2.92**	3.45	4.86
metamax-inf-50	2.21	2.94	3.46	**4.82**
metamax-inf-100	2.27	2.93	3.47	4.90
metamax-inf-200	**2.09**	2.97	3.56	4.85
metamax-inf-500	2.11	2.96	3.49	4.83
metamax-10	**2.15**	**2.83**	3.42	4.78
metamax-20	2.21	2.97	3.46	**4.72**
metamax-50	2.36	3.04	**3.37**	**4.72**
metamax-100	2.19	3.04	3.43	4.74
metamax-200	2.30	3.00	3.43	4.79
metamax-500	2.22	2.84	3.45	4.79

Table 19 APIs of MetaMax strategies for instances of single-swap.

	ss-5	ss-10	ss-18	ss-29
eps-10-0.1	**3.15**	**3.53**	**3.77**	**4.47**
eps-10-0.3	**3.15**	**3.53**	3.88	4.66
eps-10-0.5	3.22	3.64	4.02	4.83
eps-10-0.7	3.31	3.78	4.13	5.16
eps-20-0.1	**3.68**	**3.94**	**4.23**	**4.78**
eps-20-0.3	**3.68**	4.03	4.30	4.91
eps-20-0.5	3.74	4.11	4.44	5.14
eps-20-0.7	3.79	4.27	4.52	5.33
eps-50-0.1	**4.41**	**4.60**	**4.82**	**5.20**
eps-50-0.3	4.44	4.68	4.86	5.33
eps-50-0.5	4.45	4.76	4.97	5.48
eps-50-0.7	4.49	4.85	5.07	5.76
eps-100-0.1	**5.03**	**5.17**	**5.32**	**5.64**
eps-100-0.3	5.04	5.25	5.40	5.72
eps-100-0.5	5.07	5.29	5.45	5.84
eps-100-0.7	5.12	5.36	5.54	6.06
eps-200-0.1	**5.66**	**5.77**	**5.87**	**6.12**
eps-200-0.3	**5.66**	5.81	5.92	6.18
eps-200-0.5	5.68	5.83	5.99	6.30
eps-200-0.7	5.73	5.91	6.06	6.45
eps-500-0.1	**6.53**	**6.60**	**6.66**	**6.80**
eps-500-0.3	**6.53**	6.61	6.69	6.85
eps-500-0.5	6.54	6.63	6.73	6.92
eps-500-0.7	6.55	6.68	6.81	7.07

Table 20 APIs of Epsilon-greedy strategies for instances of single-swap.

A.4 Warehouse location problem

	wlp-50	wlp-200	wlp-500
constant-100	6.99	16.09	17.85
constant-200	**6.86**	16.04	**17.73**
constant-500	7.02	16.01	17.84
constant-1k	7.22	**15.96**	17.97
constant-2k	7.05	16.25	17.85
constant-5k	7.45	16.33	18.06
stagnation-100	7.10	16.20	17.88
stagnation-200	**6.90**	**16.00**	**17.87**
stagnation-500	7.11	16.13	17.92
stagnation-1k	7.07	16.14	17.96
stagnation-2k	7.19	16.13	17.89
stagnation-5k	7.49	16.24	17.94

Table 21 APIs of restarting strategies for instances of WLP.

	wlp-50	wlp-200	wlp-500
multiea-kde-10	**8.46**	16.34	17.89
multiea-kde-20	8.57	**16.33**	17.93
multiea-kde-50	8.73	16.38	17.88
multiea-kde-100	9.15	16.59	17.98
multiea-kde-200	9.62	16.40	18.05
multiea-kde-500	9.99	16.44	**17.87**
multiea-avg-10	**8.00**	16.30	17.90
multiea-avg-20	**8.00**	**16.24**	17.99
multiea-avg-50	8.36	16.46	**17.87**
multiea-avg-100	8.85	16.49	18.01
multiea-avg-200	9.22	16.27	18.07
multiea-avg-500	9.77	16.70	17.97
multiea-med-10	**8.19**	16.46	17.93
multiea-med-20	**8.19**	**16.27**	17.93
multiea-med-50	8.34	16.39	**17.90**
multiea-med-100	8.47	16.56	17.98
multiea-med-200	8.91	16.36	17.94
multiea-med-500	9.38	16.47	18.07

Table 22 APIs of MultiEA strategies for instances of WLP.

	wlp-50	wlp-200	wlp-500
metamax-k-10	**8.60**	**16.43**	**18.01**
metamax-k-20	8.89	16.52	18.14
metamax-k-50	9.09	**16.43**	18.10
metamax-k-100	9.14	16.47	18.21
metamax-k-200	9.08	16.45	18.20
metamax-k-500	9.25	16.50	18.18
metamax-inf-10	9.05	16.44	18.15
metamax-inf-20	9.08	16.47	18.06
metamax-inf-50	8.93	16.49	18.09
metamax-inf-100	8.91	**16.40**	18.14
metamax-inf-200	8.95	16.55	18.14
metamax-inf-500	**8.84**	16.46	**18.03**
metamax-10	8.96	16.50	18.13
metamax-20	8.92	16.48	18.07
metamax-50	8.95	16.46	**18.05**
metamax-100	**8.89**	16.43	18.06
metamax-200	8.94	16.43	18.13
metamax-500	8.91	**16.42**	**18.05**

Table 23 APIs of MetaMax strategies for instances of WLP.

	wlp-50	wlp-200	wlp-500
eps-10-0.1	**7.54**	**16.12**	17.87
eps-10-0.3	7.76	**16.12**	**17.80**
eps-10-0.5	7.71	16.20	17.83
eps-10-0.7	8.14	16.31	17.95
eps-20-0.1	**7.55**	16.18	17.90
eps-20-0.3	7.71	**16.12**	**17.89**
eps-20-0.5	7.85	16.17	17.90
eps-20-0.7	8.18	16.21	17.98
eps-50-0.1	7.85	16.22	17.92
eps-50-0.3	**7.84**	**16.15**	17.93
eps-50-0.5	8.01	16.18	**17.84**
eps-50-0.7	8.27	16.26	17.97
eps-100-0.1	8.16	16.43	**17.92**
eps-100-0.3	**7.93**	**16.21**	18.00
eps-100-0.5	8.14	16.35	17.93
eps-100-0.7	8.40	16.35	17.98
eps-200-0.1	8.47	16.42	18.07
eps-200-0.3	**8.26**	16.49	**17.98**
eps-200-0.5	8.49	**16.27**	**17.98**
eps-200-0.7	8.63	16.49	18.11
eps-500-0.1	9.10	16.55	**17.97**
eps-500-0.3	8.99	16.66	18.02
eps-500-0.5	**8.71**	**16.51**	18.13
eps-500-0.7	8.93	16.70	18.13

Table 24 APIs of Epsilon-greedy strategies for instances of WLP.

Appendix B

Convergence curves

B.1 Travelling salesman problem

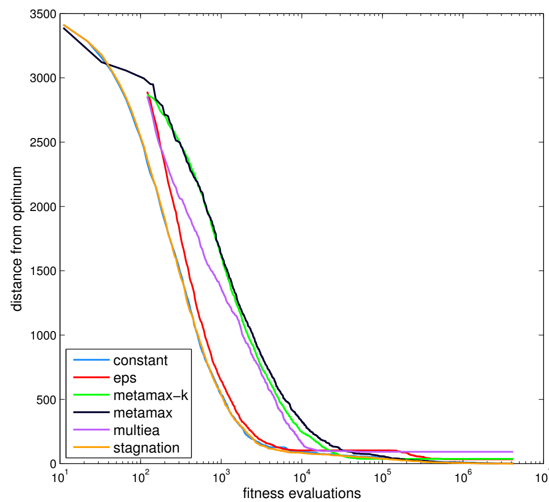

Figure 13 Convergence curves for the best strategies on tsp-42.

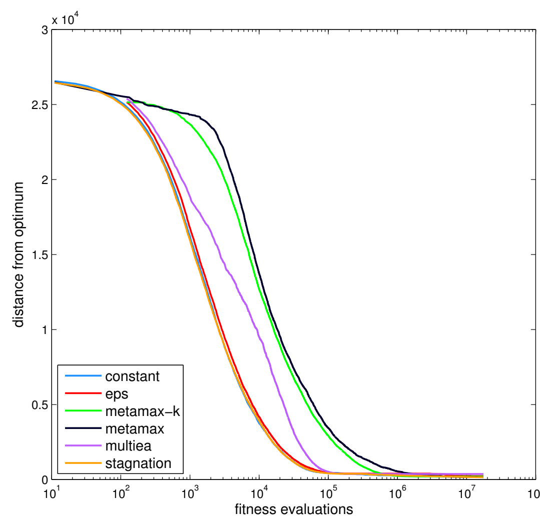

Figure 14 Convergence curves for the best strategies on tsp-175.

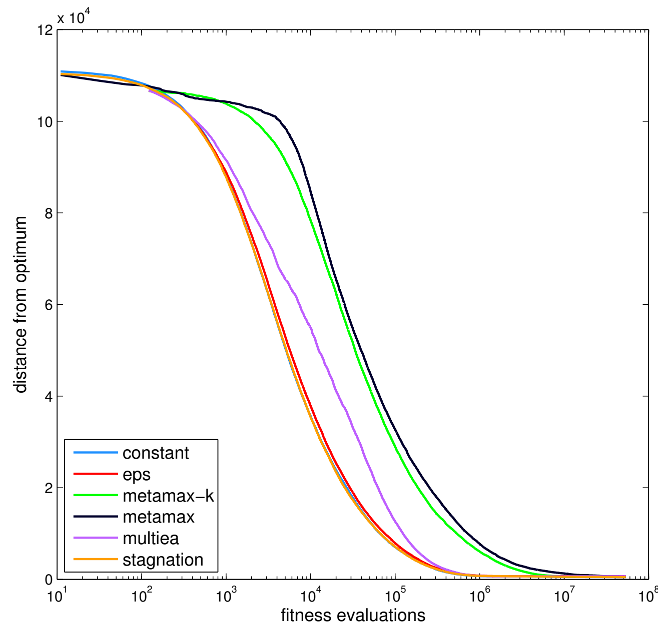

Figure 15 Convergence curves for the best strategies on tsp-535.

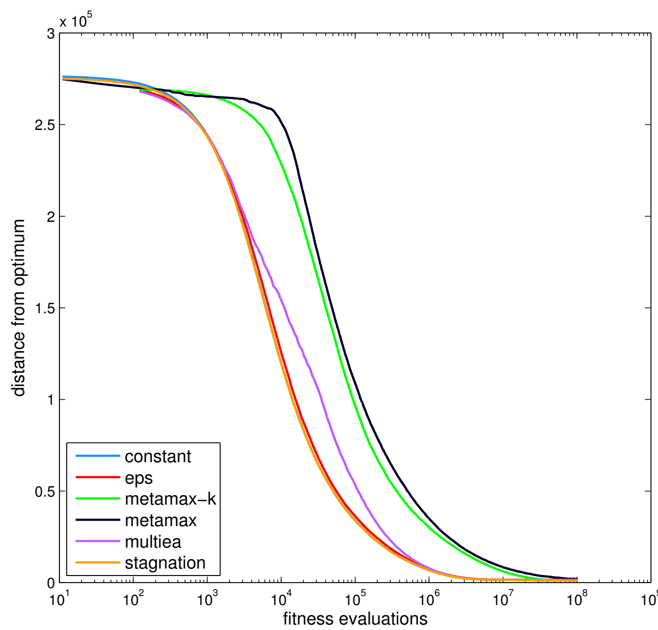

Figure 16 Convergence curves for the best strategies on tsp-1032.

B.2 K-means clustering

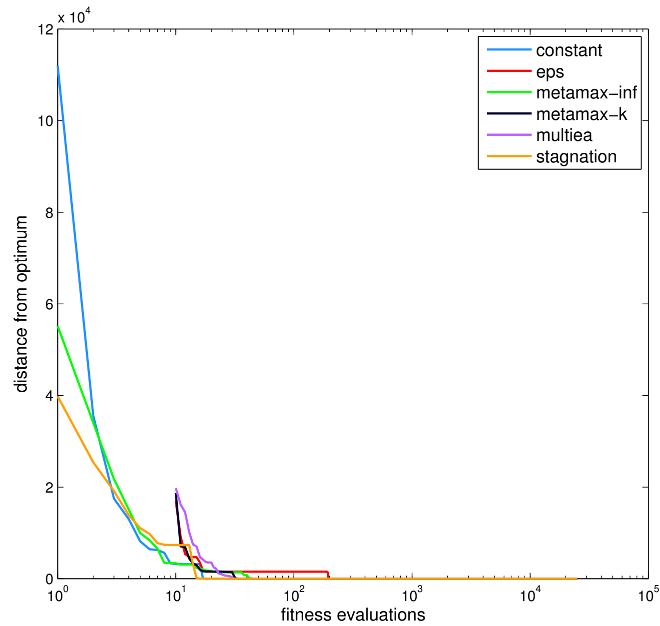

Figure 17 Convergence curves for the best strategies on km-5.

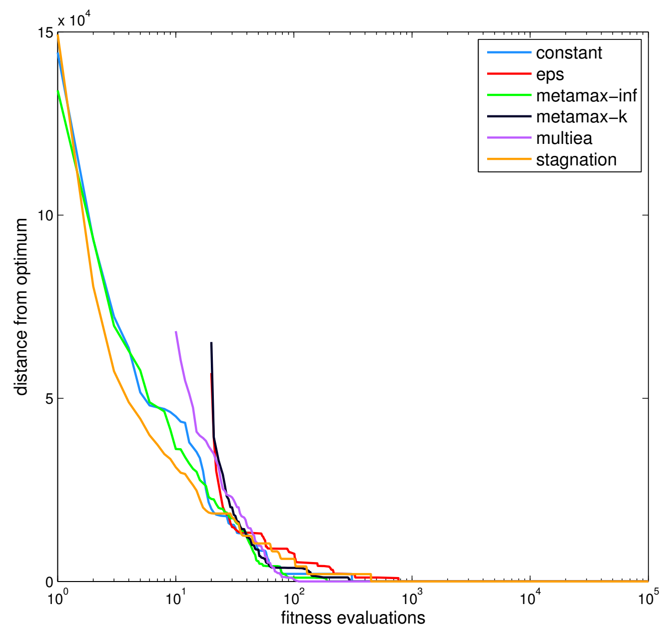

Figure 18 Convergence curves for the best strategies on km-10.

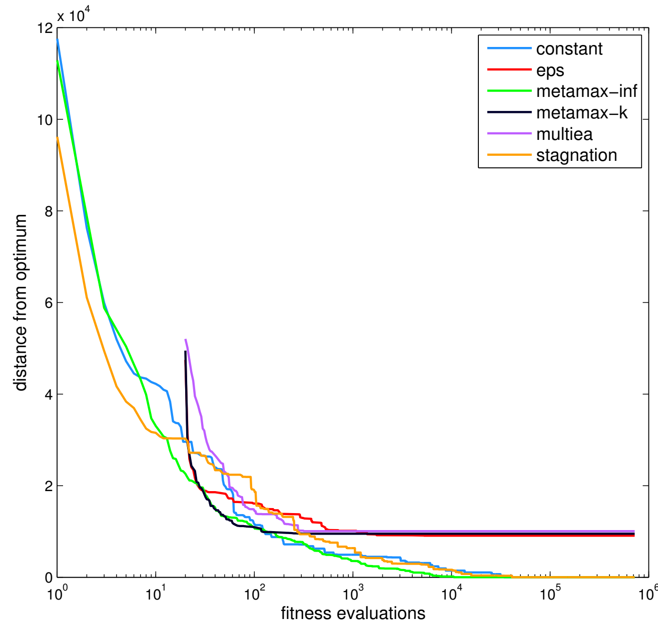

Figure 19 Convergence curves for the best strategies on km-18.

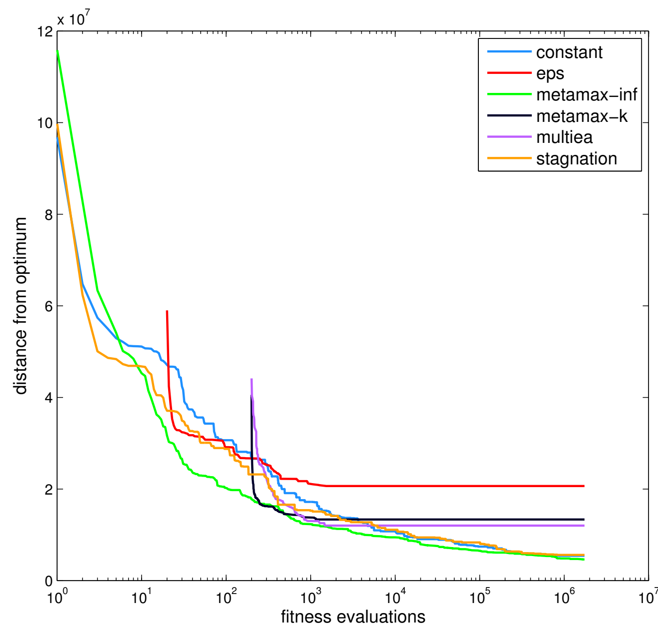

Figure 20 Convergence curves for the best strategies on km-29.

B.3 Single-swap clustering

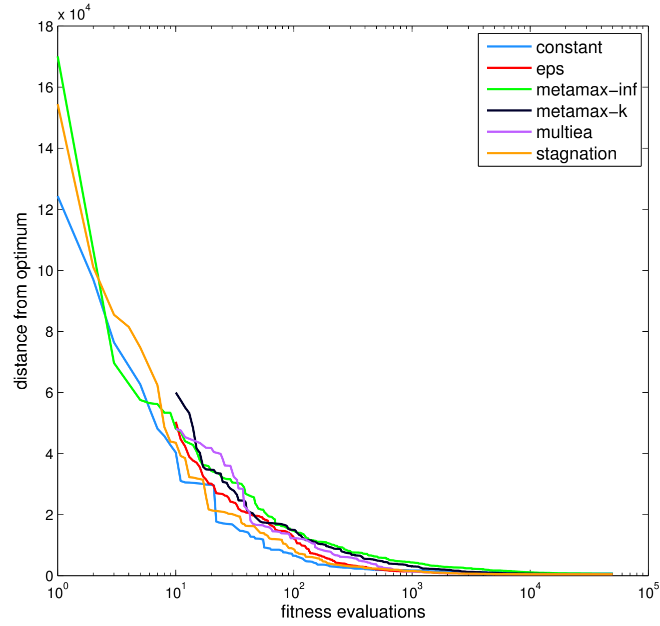

Figure 21 Convergence curves for the best strategies on ss-5.

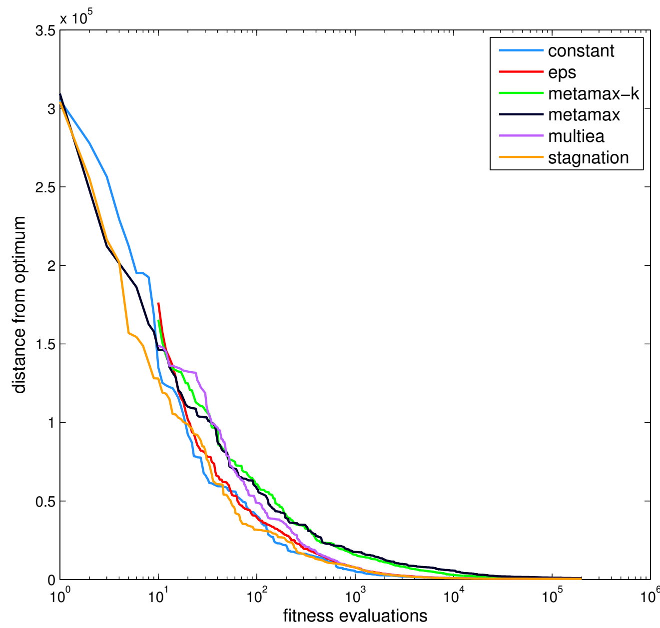

Figure 22 Convergence curves for the best strategies on ss-10.

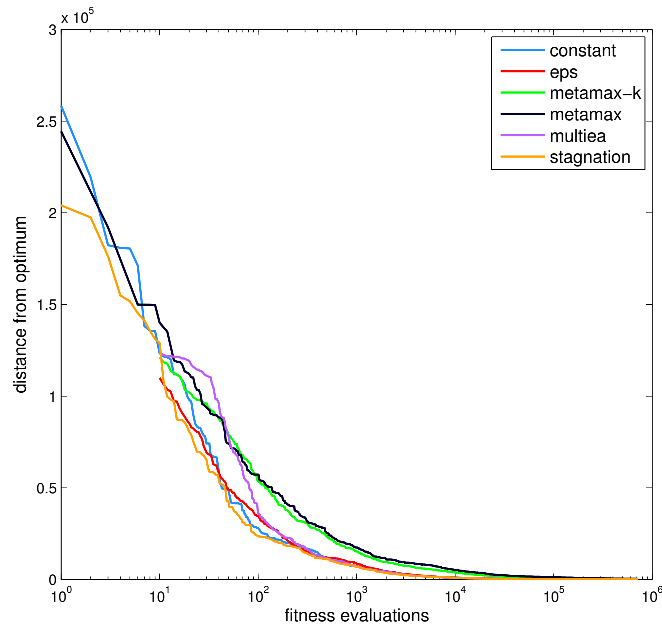

Figure 23 Convergence curves for the best strategies on ss-18.

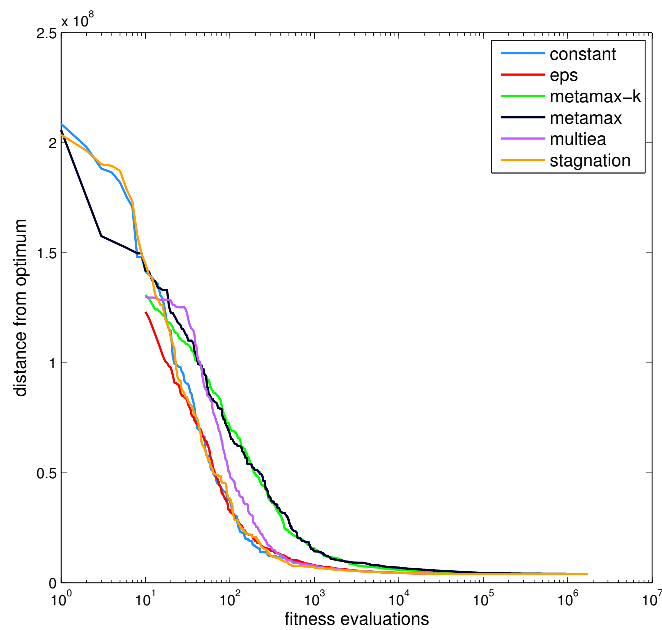

Figure 24 Convergence curves for the best strategies on ss-29.

B.4 Warehouse location problem

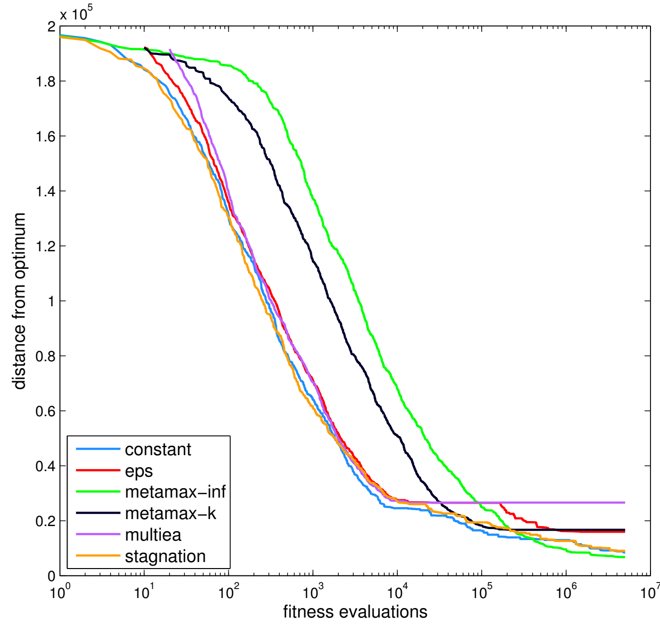

Figure 25 Convergence curves for the best strategies on wlp-50.

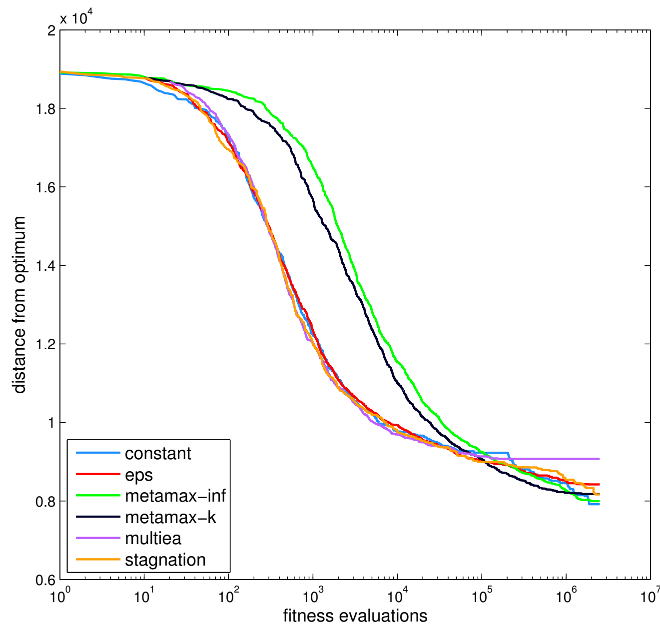

Figure 26 Convergence curves for the best strategies on wlp-200.

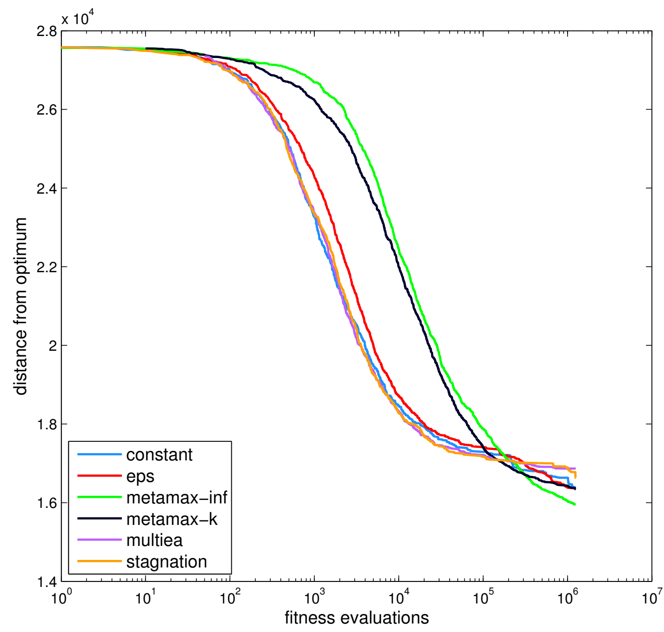

Figure 27 Convergence curves for the best strategies on wlp-500.