Iterative Prototype Optimisation with Evolved Improvement Steps

Jiri Kubalik and Jan Faigl

Department of Cybernetics, CTU Prague,
Technicka 2, 166 27 Prague 6, Czech Republic
{kubalik, xfaigl}@labe.felk.cvut.cz

Abstract. Evolutionary algorithms have already been more or less successfully applied to a wide range of optimisation problems. Typically, they are used to evolve a population of complete candidate solutions to a given problem, which can be further refined by some problem-specific heuristic algorithm. In this paper, we introduce a new framework called *Iterative Prototype Optimisation with Evolved Improvement Steps*. This is a general optimisation framework, where an initial prototype solution is being improved iteration by iteration. In each iteration, a sequence of actions/operations, which improves the current prototype the most, is found by an evolutionary algorithm. The proposed algorithm has been tested on problems from two different optimisation problem domains - binary string optimisation and the traveling salesman problem. Results show that the concept can be used to solve hard problems of big size reliably achieving comparably good or better results than classical evolutionary algorithms and other selected methods.

1 Introduction

In the evolutionary optimisation framework, the evolutionary algorithms (EAs) are typically used to evolve a population of candidate solutions to a given problem. Each of the candidate solutions encodes a complete solution - a complete set of problem control parameters, a complete schedule in the case of scheduling problems, a complete tour for the traveling salesman problem, etc. This implies, that especially for large instances of the solved problem the EA searches enormous space of potential solutions. In this paper, a new approach is presented, where the EA does not handle the solved problem as a whole. Instead, the EA is employed within the iterative optimisation framework to evolve the best modification of the current solution prototype in each iteration. Thus, the load of searching for the best complete solution at once is cut into pieces, each of them representing a process of seeking the best transformation of the current solution prototype to the new possibly better one.

The structure of the paper is as follows. In section 2, the general outline of the algorithm of Iterative **P**rototype **O**ptimisation with **E**volved I**M**provement **S**teps (POEMS) is described. Section 3 describes the problems and the experimental set-up used for the proof-of-concept validation of POEMS. In section 4, POEMS is compared with other evolutionary approaches and other selected

P. Collet et al. (Eds.): EuroGP 2006, LNCS 3905, pp. 154–165, 2006.

methods. The paper ends with conclusions on effectiveness of POEMS, its advantages and disadvantages, and its further extensions.

2 POEMS

The main idea behind POEMS (Figure 1) is that some initial prototype solution is further improved in an iterative process, where the most suitable modification of the current prototype is sought for using an evolutionary algorithm (EA) in each iteration. The modifications are represented as a sequence of primitive actions/operations, defined specifically for the solved problem. Such a sequence can be considered a program and the employed EA a special case of a linear genetic programming. The evaluation of action sequences is based on how good/bad they modify the current prototype, which is an input parameter of EA. Sequences that do not change the prototype at all are penalized in order to avoid generating trivial solutions. After the EA finishes, it is checked whether the best evolved sequence improves the current prototype or not. If an improvement is found, then the sequence is applied to current prototype and the result becomes a new prototype. Otherwise the current prototype remains unchanged for the next iteration.

Representation. The EA evolves linear chromosomes of length *MaxGenes*, where each gene represents an instance of certain action chosen from a set of elementary actions defined for given problem. Each action is represented by a record, with an attribute *action_type* followed by parameters of the action. Besides actions that truly modify the prototype there is also a special type of action called *nop* (no operation). Any action with *action_type* = *nop* is interpreted as a void action with no effect on the prototype, regardless of the values of its parameters. A chromosome can contain one or more instances of the *nop* operation. This way a variable effective length of chromosomes is implemented.

Operators. The crossover operator generates a child chromosome so that each gene of the new chromosome is a copy of randomly chosen gene either from the first or the second parent. Both parents have the same probability of contributing its genes to the generated child, and each gene can be used only once. This can be considered a generalized uniform crossover, where any combination of parental genes can form a valid offspring. After the new chromosome has been

```
1    generate(Prototype)
2    repeat
3        BestSequence ← run_EA(Prototype)
4        if(apply_to(BestSequence, Prototype) is better than Prototype)
5            then Prototype ← apply_to(BestSequence, Prototype)
6    until(POEMS termination condition)
7    return Prototype
```

Fig. 1. An outline of POEMS algorithm

```
1   initialize(OldPop)
2   BestSequence ← best_of(OldPop)
3   repeat
4       NewPop ← BestSequence
5       repeat
6           Parents ← select(OldPop)
7           Children ← cross_over(Parents)
8           mutate(Children)
9           evaluate(Children)
10          NewPop ← Children
11      until(NewPop is completed)
12      BestSequence = best_of(NewPop)
13      switch(OldPop, NewPop)
14  until(EA termination condition)
15  return BestSequence
```

Fig. 2. An outline of a simple generational evolutionary algorithm

finished, it is checked for gene duplicates and left with just one copy of each non *nop* action. Each duplicate of some non *nop* action is converted into *nop* action, simply by setting *action_type* = *nop*. This means that the population genotype can contain many "inactive" action specifications. These can be activated again by changing their *action_type* from *nop* to some effective action. Action can be activated/inactivated by mutation operator, which can also change the parameters of the action.

Evolutionary model. The design and configuration of the EA can differ for each particular optimisation problem. Figure 2 shows a simple generational evolutionary algorithm (gEA) with tournament selection, and elitism preserving the best individual in the population. Figure 3 shows a mutation-based iterational EA (iEA) that iteratively modifies a population of individuals. In each iteration a chromosome is selected by tournament selection. Then the chromosome is mutated so that one action out of its active actions (i.e. genes with *action_type* other than *nop*) is selected and inactivated (*action_type* set to *nop*). If the fitness of the chromosome did not worsen after this change the modified chromosome is accepted and replaces other bad performing chromosome in the population.

In general, the EA is expected to be executed many times during the whole run of the POEMS. Thus, it must be configured to converge fast in order to get the output in short time. As the EA is evolving sequences of actions to improve the solution prototype, not the complete solution, the maximal length of chromosomes *MaxGenes* can be short compared with the size of the problem. For example *MaxGenes* would be much smaller than the size of the chromosome in case of binary string optimisation or much smaller than the number of cities in case of the TSP problem. The relaxed requirement on the expected EA output and the small size of evolved chromosomes enables to setup the EA so that it converges within a few generations. It is important to note, that POEMS does not perform prototype optimisation via improvement steps that are purely

```
1  initialize(Population)
2  repeat
3      Parents ← select(Population)
4      Children ← cross_over(Parents)
5      mutate(Children)
6      evaluate(Children)
7      Replacement = find_replacement(Population)
8      Population[Replacement] ← Child1
9      Replacement = find_replacement(Population)
10     Population[Replacement] ← Child2
11 until(EA termination condition)
12 BestSequence ← best_of(Population)
13 return BestSequence
```

Fig. 3. An outline of a mutation-based iterational evolutionary algorithm

local with respect to the current prototype. In fact, long phenotypical as well as genotypical distances between the prototype and its modification can be observed if the system possesses a sufficient explorative ability. The space of possible modifications of the current prototype is determined by the set of elementary actions and the maximum allowed length of evolved action sequences *MaxGenes* (as demonstrated in section 6). The less explorative actions are and the shorter sequences are allowed the more the system searches in a prototype neighborhood only and the more it is prone to get stuck in a local optimum, and vice versa.

3 Test Problems

The first set of test problems belongs to a binary string optimisation problem domain. It includes simple onemax, royal road, deceptive, hierarchical, multimodal and non-linear function optimisation problems.

OneMax. This is a simple problem, where the chromosome is assigned a value equal to the number of ones it contains. Thus the optimal sought string is of fitness equal to the size of the chromosome (that is 100, here). Note that this function is considered to be easy for GAs.

DF3. This is a representative of deceptive problems, i.e. problems that are intentionally designed to make a GA converge towards local deceptive optimum. The problem is composed of 25 copies of a 4-bit fully deceptive function DF3 taken from [8]. DF3 has a global optimum in the string 1111 with fitness 30 and a deceptive attractor 0000 with low fitness 10, which is surrounded, in the search space, by four strings of just one 1 with fitness values 28, 27, 26, and 25. The whole 100-bit long chromosome has the global optimum of value 750.

Rosenbrock. This problem uses as the basic building block the well-known Rosenbrock function of two parameters x and y, each of them coded on 12 bits. The function has high degree of dependency between variables, which makes it hard

to optimize using standard genetic algorithms. The sought minimum value of the function is 0.0 at the point [1.0, 1.0]. The problem consists of 4 copies of the function whose contributions are summed up in the final fitness value. Any solution of fitness less than or equal to 0.001 is considered to be an optimum.

F103. This test problem is based on function $F103(x, y)$ taken from [9]. It is a non-linear non-separable and highly multimodal function of two variables, where the parameters x and y are each coded on 10 bits. The global minimum is of value 0.0. Our problem consists of 5 copies of the function, where the fitness of the whole chromosome is given as the sum of the five function contributions. Again, any solution of fitness less than or equal to 0.001 is considered an optimum.

RR. This is a 16-bit version of the RR1 single-level royal road problem described in [2]. The problem is defined by enumerating the schemata, where each schema s_i has assigned its contribution coefficient c_i. The evaluation of an arbitrary chromosome is given as a sum of all contributions of those schemata that are covered by the chromosome. Only the combination of all ones on the bits pertinent to a given schema contributes to the fitness with the nonzero value, any other combination has value 0. Here, the problem is defined as a concatenation of six 16-bit long schemata, so the optimum solution is the string of all ones of the fitness 96.

H-IFF. A hierarchical-if-and-only-if function proposed in [6] is the representative of hierarchically decomposable problems. The hierarchical block structure of the function is a balanced binary tree. Leaf nodes, corresponding to single genes, contribute to the fitness by 1. Each inner node is interpreted as 1 if and only if its children are both 1's, and as 0 iff they are both 0's - in such cases the inner node contributes to the fitness by a positive value $2^{height(x)}$, where $height(x)$ is the distance from the node x to its antecedent leaves. Otherwise the node is interpreted as null and its contribution is 0. The function has two global optima - one consists of all 1's and the other one has all 0's. We have used the 128-bit version with global optima of value 1024.

TSP. The second set of test problems are instances of the well-known Traveling Salesman Problem. We have used datasets for 100, 200, 500, 1000, and 2000 cities, where the cities were generated randomly in the area of size 100 by 100.

4 Optimisation Techniques Used for Comparisons

On the binary string optimisation problems, we have compared the proposed POEMS algorithm with the following approaches :

- *Simple Genetic Algorithm* (SGA). This is a generational genetic algorithm, with tournament selection, 2-point crossover, a simple bit-swapping mutation operator, and an elitism, which preserves the best individual in the population. Population size 500 was used. The probability of crossover was 0.9. The mutation rate was set so that one bit of each chromosome is inverted.
- *Genetic Algorithm with Gene Based Adaptive Mutation Strategy* (GBAM). Uyar et al. [5] proposed this adaptive approach for adjusting mutation rates

for the gene locations based on the feedback obtained by observing the relative success or failure of the individuals in the population. There are two mutation rates for each locus - one for allele 1 and the other for allele 0. For each generation the mutation rates are updated for each locus so that the mutation rate for the better-performing allele decreases, and vice versa. This certainly speeds up the convergence so the strategy is implemented with a convergence control mechanism for escaping local optima.

– *A Genetic Algorithm with Limited Convergence* (GALCO). Kubalik et al. [4] proposed this approach for preserving population diversity. It is based on an idea that the population is explicitly prevented from becoming homogenous by simply imposing limits on its convergence. This is done by specifying the maximum difference between frequency of ones and zeros at any position of the chromosome calculated over the whole population. A steady-state evolutionary model and a special replacement operator are used to keep the desired distribution of ones and zeros during the whole run.

Note that all of these techniques are more or less modifications of the standard genetic algorithm. As such they rely on the proper spacial structure of the chromosome. In other words, they work well if the groups of dependent genes are spatially clustered within the chromosome. This is called a tight linkage. In the opposite case, when the linkage is loose (i.e. the dependent genes are far from each other), the genetic algorithm can not combine building blocks of two parental chromosomes properly, see [3]. In order to show the effect of the linkage on the performance of the algorithms, two series of experiments were carried out - one for the *tight linkage*, and the other for the *loose linkage* (this was implemented so that the sequence of genes within the chromosome was chosen by random for each experiment). Obviously, the evolutionary algorithm used in POEMS is linkage independent, so it was tested on the loose linkage only.

The following approaches have been compared with POEMS on the TSP:

– *Genetic Algorithm with E-R crossover* (ER). This is a steady-state genetic algorithm, with the edge-recombination crossover proposed by Whitley et al. [7]. Mutation operator exchanges positions of two randomly selected cities within a given path. First, two parental chromosomes are selected by tournament selection, and crossed over to generate its offspring. The new chromosome undergoes the mutation and is evaluated. Finally, the newly generated chromosome replaces a chosen poorly performing chromosome in the population if the new chromosome outperforms the replacement one.
– *Self Organising Maps* (SOM). The salesman's city tour is represented by a ring of neurons, where the neighboring neurons are connected. The general schema of SOM algorithm consists of two procedures: (1) a selection of winner neuron, where the closest neuron for each city is selected and (2) an adaptation of the winner neuron, where the neuron along with its several neighbors are moved towards the closest city. These two procedures are repeated until a stopping condition is satisfied. The algorithm implemented in this work is based on [1].

– *2-opt heuristic* (2-opt). This algorithm is based on simple local search heuristic called 2-opt, that was proposed by Flood and Croes in fifties. The algorithm starts with some feasible (random) solution. Than it searches for two edges $e_1 = (v_1, v_3), e_2 = (v_2, v_4)$ such that a recombination of the edges to $e_1 = (v_1, v_2), e_2 = (v_3, v_4)$ improves the current solution. The algorithm stops if no two edges for improvement recombination are found.

5 Experimental Setup

For the binary string optimisation problems the action sequences evolved in POEMS were composed of just one type of action called *invert*(*gene*). The action simply inverts specified *gene* within the prototype.

For the TSP problem a direct path representation of the tour was used. The prototype tour in POEMS was modified by action sequences composed of actions of the following types

– $move(city_1, city_2)$ moves $city_1$ right after $city_2$ in the tour,
– $invert(city_1, city_2)$ inverts a subtour between $city_1$ and $city_2$,
– $swap(city_1, city_2)$ swaps $city_1$ and $city_2$.

Both versions, POEMS with gEA and POEMS with iEA were tested on the binary string optimisation problems. Both of them used the same parameter setup as follows: chromosome length 10 (the maximal number of active actions in the action sequence, see Section 2), population size 200, number of fitness function evaluations 3000, tournament selection with $N = 3$, crossover and mutation operators as described in Section 2 with $P_c = 0.8$ and $P_m = 0.1$, respectively.

The other algorithms were used with the following common setting: population size 500, 2-point crossover with $P_c = 0.8$, tournament selection with $N = 3$. SGA used a mutation operator with the probability $P_m = 0.01$, GALCO does not use any explicit mutation operator, and GBAM was used with the mutation rate interval $(0.0001 - 0.2)$, the initial mutation rate 0.025, and the mutation adaptation step 0.001. All algorithms were running for 10^6 fitness evaluations.

The following configurations of POEMs and ER algorithms were used in experiments on TSP problem. The population size was set to the number of cities ($Cities$) and $2{\cdot}Cities$ for POEMS and ER, respectively. EA used in POEMS (line 3 in Figure 1) worked with chromosomes of length 10 and lasted for $1000{\cdot}PopSize$ fitness evaluations.

Two strategies for generating of the starting prototype were used - the random initialisation and the heuristic one. When generating a tour, a decision of what city should be visited from the current city is made by random in the random strategy whilst the heuristic strategy prefers the next city to be from the neighborhood of the current city. Similarly, the concept of the neighborhood was used with the ER crossover so that the operator prefers links to cities from the current city's neighborhood.

Both POEMs and ER were running for $Cities \cdot 1000$ fitness function evaluations. The tournament selection parameter and the neighborhood size were set

Table 1. Configurations of POEMs and ER used for TSP problem

PopSize	*Tournament*	*Cities*	*Neighborhood*
100	3	100	20
200	4	200	15
500	5	500	10
1000	6	1000	7
2000	7	2000	5
4000	8		

depending on the population size and the number of cities as shown in Table 1. The crossover operator was applied on the parents with the probability 0.9. If the parents did not undergo crossover they were mutated so that positions of two randomly chosen cities were swapped. The following statistics were calculated based on 50 runs of each experiment

- *Mean*. Mean *best-of-run* value calculated over the 50 independent runs.
- *StDev*. Standard deviation of the *best-of-run* values.
- *#Succ*. A number of runs, in which the optimum solution was found.
- *When*. The average number of fitness evaluations needed to get the optimum.
- *BestPath*. The shortest path out of 50 runs for each TSP experiment.

6 Results

Table 2 shows results obtained with POEMS-gEA and POEMS-iEA on binary string optimisation problems. It shows that POEMS-iEA is better than POEMS-gEA on OneMax, DF3, Rosenbrock, and F103 problem. POEMS-iEA achieves either better mean quality of the best-of-run solutions or finds the optimal solutions more often or is faster in converging to the optimal solution on those problems. This may be attributed to the fact that the iEA is designed to converge very fast so it might be able to come up with better action sequence than gEA in each iteration. On the other hand, the performance of both variants of POEMS is very poor for the RR and H-IFF problems. For the royal road problem this might be surprising as those problems are considered easy for genetic algorithms. The explanation of why POEMS does not work for these problems

Table 2. Performance of POEMS on binary string optimisation problems

	POEMS-gEA				POEMS-iEA			
problem	*Mean*	*StDev*	*#Succ*	*When*	*Mean*	*StDev*	*#Succ*	*When*
OneMax	100.0	0.0	50	20732	100.0	0.0	50	12168
DF3	749.1	3.7	47	617034	750	0.0	50	307820
Rosenbrock	0.36	0.52	7	656342	0.029	0.14	17	516988
F103	0.0127	0.0093	0	-	0.0063	0.0061	1	769400
RR	5.1	7.5	0	-	6.7	9.7	0	-
H-IFF	568.8	58.8	0	-	580.8	15.2	0	-

is that *it lacks the ability to combine several solution into a new one* as the standard genetic algorithms do. POEMS seeks the optimal solution via a process of iteratively modifying (mutating) the prototype solution. Thus, the algorithm can get stuck with such a prototype, for which it is very hard (or even impossible) to evolve any improving action sequence. In case of RR problem, for POEMS is very hard to find a sequence of single bit inversions such that it would discover a new 16-bit long building block of all ones without simultaneously damaging any of already existing blocks in the prototype. Similarly this applies for H-IFF problem, where POEMS optimizes the prototype up to some level, where any further improvement would require to invert a large block of genes.

When comparing POEMS with the other algorithms (see Table 3) on the binary string optimisation problems we can observe that if the chromosome

Table 3. Performance of SGA, GBAM and GALCO on binary string optimisation problems

		tight linkage				loose linkage			
	problem	*Mean*	*StDev*	*#Succ*	*When*	*Mean*	*StDev*	*#Succ*	*When*
SGA	OneMax	100.0	0.0	50	13448	100.0	0.0	50	13564
	DF3	750	0.0	50	595359	707.1	6.8	0	-
	Rosenbrock	0.063	0.084	4	724888	0.502	0.591	0	-
	F103	0.005	0.0035	2	680329	0.0197	0.0156	0	-
	RR	91.5	7.3	36	654756	88.8	9.7	35	706064
	H-IFF	710.4	87.8	1	295261	617.6	34.7	0	-
GBAM	OneMax	100.0	0.0	50	10935	100.0	0.0	50	10847
	DF3	750.0	0.0	50	558097	728.3	6.9	0	-
	Rosenbrock	1.22	0.46	0	-	2.11	0.8	0	-
	F103	0.32	0.11	0	-	0.47	0.17	0	-
	RR	96.0	0.0	50	43984	96.0	0.0	50	68378
	H-IFF	790.8	57.4	0	-	625.6	38.1	0	-
GALCO	OneMax	100.0	0.0	50	141190	100.0	0.0	50	142170
	DF3	750.0	0.0	50	108685	714.6	5.2	0	-
	Rosenbrock	0.0103	0.0137	1	936201	0.36	0.26	0	-
	F103	0.0008	0.0004	38	642325	0.025	0.013	0	-
	RR	94.7	4.4	23	453373	43.5	8.6	0	-
	H-IFF	1024.0	0.0	50	22620	574.4	46.9	0	-

Table 4. Performance of POEMS-gEA on TSP problem

	random			heuristic		
cities	*Mean*	*StDev*	*BestPath*	*Mean*	*StDev*	*BestPath*
100	831.9	15.7	803.8	818.6	14.5	786.1
200	1190.1	25.6	1156.3	1132.9	16.2	1098.1
500	2025.9	40.3	1975.6	1746.3	13.5	1718.0
1000	9970.0	290.0	9475.9	2523.0	15.0	2491.6
2000	34829.0	503.9	34281.5	3692.2	20.4	3655.1

representation with tight linkage is used the GALCO algorithm slightly outperforms POEMS on DF3 and F103 problems. However, the situation changes when solving problems with loose linkage. Then the POEMS performs considerably better than the other algorithms on DF3, Rosenbrock and F103 problems.

Tables 4 and 5 provide a comparison of the POEMS with ER, ER-heuristic, SOM, and 2-opt. We can observe that POEMS with random initialisation of the prototype works poorly on large TSP datasets. This is because starting from a very bad tour it would require many more iterations to find a good solution than allowed here. On the other hand, when the heuristic is used for generation of the initial prototype the POEMS outperform all the other approaches even on the large datasets of 2000 cities.

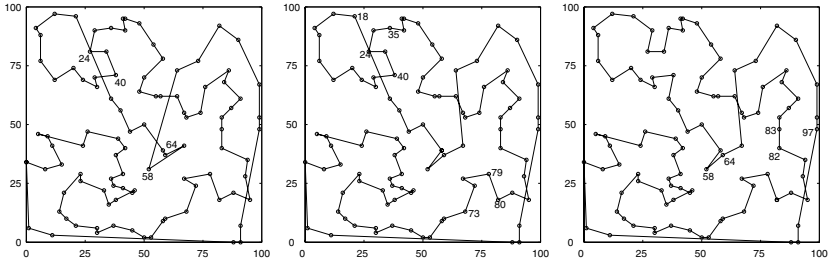
Table 5. Performance of ER, ER-heuristic, SOM, and 2-opt on TSP problem

	ER			ER-heuristic		
cities	*Mean*	*StDev*	*BestPath*	*Mean*	*StDev*	*BestPath*
100	1192.9	54.7	1115.6	935.1	27.1	884.1
200	2096.3	74.6	1973.6	1406.1	54.6	1289.1
500	4562.4	155.5	4424.0	2753.3	66.4	2666.6
1000	8256.5	223.9	7964.1	3799.6	91.4	3678.8
2000	16956.1	480.7	16402.4	5983.5	95.1	5875.8
	SOM			2-opt		
cities	*Mean*	*StDev*	*BestPath*	*Mean*	*StDev*	*BestPath*
100	830.4	13.0	811.7	853.5	18.2	797.3
200	1155.4	12.4	1124.7	1196.1	24.1	1149.2
500	1776.0	14.0	1751.2	1866.8	123.1	1772.9
1000	2533.0	12.0	2508.4	2650.1	160.0	2572.1
2000	3725.3	14.9	3695.6	3908.1	122.8	3789.2

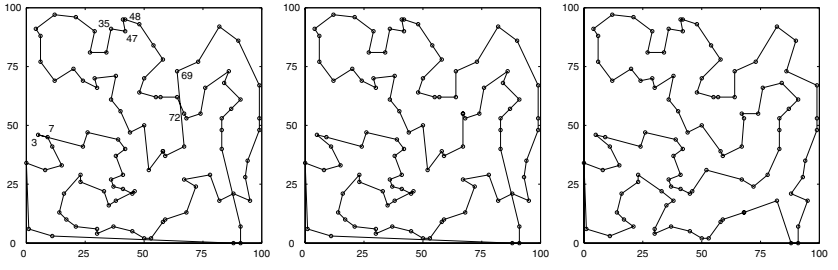
Table 6. Performance of POEMS-gEA using just one type of the elementary function

action type	*Mean*	*StDev*	*BestPath*
invert	2554.7	13.3	2526.2
move	2689.1	20.1	2653.6
swap	2824.8	31.5	2775.1

Results in Table 6 demonstrate how the selection of elementary functions affects the performance of the POEMS approach. The results were obtained for TSP problem with 1000 cities. It shows that if just one function out of the three functions *invert*, *move*, and *swap* is enabled the POEMS performs worse than if all the functions are allowed to be combined in the action sequences. A fragment of an execution of POEMS on TSP with 100 cities is shown in Figure 4.



a) initial tour b) after iteration 1 c) after iteration 2



d) after iteration 3 e) after iteration 4 f) final tour

iteration	prototype fitness	evolved action sequence	final fitness
1	965.134	(move 58 64), (invert 24 40)	952.550
2	952.550	(move 79 73), (invert 24 18), (invert 24 35), (move 79 80), (invert 24 40)	927.025
3	927.025	(invert 97 82), (invert 83 82), (move 58 64)	919.573
4	919.573	(invert 48 47), (invert 69 72), (invert 35 47), (swap 7 3)	904.033

Fig. 4. A fragment of an execution of POEMS on TSP with 100 cities. a) an initial tour prototype of length 965.134. b) the tour obtained after applying the action sequence evolved in iteration 1 on the current prototype. c) the tour obtained after iteration 2. d) the tour obtained after iteration 3. e) the tour obtained after iteration 4. f) the final tour of length 824.874.

7 Conclusions

In this paper, an algorithm called *Iterative Prototype Optimisation with Evolved Improvement Steps* (POEMS) is proposed. POEMS iteratively improves the prototype solution via evolving the best sequence of actions to be applied to the current prototype in each iteration.

The POEMS concept has been tested on the binary string optimisation problems and the traveling salesman problem and compared with other optimisation algorithms. The presented experiments show that the proposed approach

achieves competitive or better results than the compared algorithms. However, as a mutation-based optimisation approach it possesses a limited ability to identify and process building blocks of higher order.

On the other hand, this approach might be well suited for solving problems where the representation does not allow to design crossover operators that would effectively mix important building blocks of the parental solutions. In other words, these are the problems where the crossover performs just as a hypermutation.

Future research will focus on the analysis of the proposed algorithm behavior as well as on the identification of problems the algorithm is well suited for.

Acknowledgments. The research has been supported by the research program No. MSM6840770012 "Transdisciplinary Research in the Area of Biomedical Engineering II" of the CTU in Prague, sponsored by the Ministry of Education, Youth and Sports of the Czech Republic. Authors would like to thank Petr Pošík (CTU Prague) for many valuable comments improving the paper.

References

1. Faigl, J., Kulich, M., Přeučil, L.: Multiple traveling salesmen problem with hierarchy of cities in inspection task with limited visibility. In proceedings of the 5th Workshop on Self-Organizing Maps. Universit Paris-Sud, (2005) 91–98
2. Forrest, S. and Mitchell, M.: Relative building-block fitness and the Building Block Hypothesis. In Whitley, L. D. (Ed.), *Foundations of Genetic Algorithms 2*. San Mateo, CA: Morgan Kaufmann, (1993) 109–126
3. Goldberg, D.E.: *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Genetic Algorithms and Evolutionary Computation, Volume 7. Kluwer Academic Pub; ISBN: 1402070985, (2002)
4. Kubalík, J., Rothkrantz, L.J.M., Lažanský, J.: Genetic Algorithms with Limited Convergence. In Grana, M.; Duro, R.J.; Anjou, A.d.; Wang, P.P. (Eds.), *Information Processing with Evolutionary Algorithms: From Industrial Applications to Academic Speculations*, ISBN: 1-85233-866-0, (2005) 233–254
5. Uyar, A.S., Sariel, S., Eryigit, G.: "A Gene Based Adaptive Mutation Strategy for Genetic Algorithms", GECCO 2004: Genetic and Evolutionary Computation Conference, (2004) 271–281
6. Watson, R.A., Hornby, G.S. and Pollack, J.B.: Modeling Building-Block Interdependency. In *Fifth International Conference PPSN V*. Springer, (1998) 97–106
7. Whitley, D., Starkweather, T., D'Ann Fuquay: Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. ICGA 1989. (1989) 133–140
8. Whitley, D.: Fundamental Principles of Deception in Genetic Search. In: *Foundations of Genetic Algorithms*. G. Rawlins (ed.), Morgan Kaufmann, (1991) 221–241
9. Whitley, D., Mathias, K., Rana, S., Dzubera, J.: Evaluating Evolutionary Algorithms. *Artificial Intelligence*, Volume 85, (1996) 245–2761