A Genetic Algorithm for the Index Selection Problem

Jozef Kratica[1], Ivana Ljubić[2] and Dušan Tošić[3]

[1] Institute of Mathematics, Serbian Academy of Sciences and Arts,
Kneza Mihajla 35/I, pp. 367, 11001 Belgrade, Yugoslavia
jkratica@mi.sanu.ac.yu
[2] Institute of Computer Graphics and Algorithms, Vienna University of Technology,
Favoritenstraße 9–11/186, 1040 Vienna, Austria
ljubic@ads.tuwien.ac.at
[3] Faculty of Mathematics, University of Belgrade,
Studentski trg 16, Belgrade, Yugoslavia
dtosic@matf.bg.ac.yu

Abstract. This paper considers the problem of minimizing the response time for a given database workload by a proper choice of indexes. This problem is NP-hard and known in the literature as the Index Selection Problem (ISP).
We propose a genetic algorithm (GA) for solving the ISP. Computational results of the GA on standard ISP instances are compared to branch-and-cut method and its initialisation heuristics and two state of the art MIP solvers: CPLEX and OSL. These results indicate good performance, reliability and efficiency of the proposed approach.

1 Introduction

The *Index Selection Problem* (ISP) is one of the crucial problems in the physical design of databases. The main goal is to choose a subset of given indexes to be created in a database, so that the response time for a given database workload is minimized.

The ISP represents a generalization of the well-known *Uncapacitated Facility Location Problem* - UFLP, which is known as NP-hard in a strong sense [1]. Some authors also consider the ISP problem as the multilevel (or multi-stage) uncapacitated facility location problem ([2, 3]). Suppose we are given a set of *indexes* (that can be built or not) and a set of *queries*. Each built index requires a *maintenance time*, while each query can be answered in *answer time*, which depends on the set of built indexes. The main goal is to minimize the overall execution time, defined as the sum of the maintenance times plus the sum of the answer times for all queries.

In the simple case, each query can be answered either without using any index, in a given answer time, or with using *one* built index, reducing answer time by a gain specified for every index usable for query. In this situation the problem can be formulated as an UFLP.

However, in most database management systems each query may use not only a single index, but a set of indexes ([4, 5]). This problem can not be formulated as a pure UFLP, but must be generalized and formulated as an ISP. Detailed description of the problem and analysis of all aspects of its influence on the managing of database systems is beyond the scope of this paper. Some important survey articles are [6, 7].

In this paper we propose a *genetic algorithm* (GA) for solving the ISP. In the following section, we will present the mathematical formulation of the problem and point out the previous work. Section 3 explains in detail all important aspects of the GA: encoding, evaluation of solutions and genetic operators. In Section 4, we will compare the results of the GA with branch-and-cut (BnC) algorithm proposed in [8, 9]. Results are also compared with those obtained using two state of the art MIP solvers: CPLEX and OSL. Finally, in Section 5, we will draw out some conclusions and propose ideas for the future work.

2 The Index Selection Problem

Let $N = \{1, 2, ..., n\}$ be the set of all indexes and $M = \{1, 2, ..., m\}$ the set of all queries for a database. Each index can be built or not - each built index requires a given maintenance time $f_j > 0$. In the ISP, not only single indexes, but also sets of indexes can be used for a query. Therefore, we are given a set of *configurations* $P = \{1, 2, ..., p\}$ - each configuration $k \in P$ is associated with some subset $N_k \subset N$ of indexes. A configuration is *active*, if all its indexes are built. When all indexes of a certain configuration $k \in P$ are built, during the execution of a query $i \in M$, we gain $g_{ik} \geq 0$ time. In practice, for the most pairs (i,k), $i \in M$, $k \in P$ the gains g_{ik} are equal zero. This can easily be explained with the fact that a certain configuration has an influence only on a limited number of queries from M. Our goal is to build the indexes, so that the total time needed to execute all the queries is minimized, i.e. that the total time of gains is maximized. Formally, the problem can be stated as:

$$max \left(\sum_{i \in M} \sum_{k \in P} g_{ik} \cdot x_{ik} \; - \; \sum_{j \in N} f_j \cdot y_j \right) \tag{1}$$

subject to

$$\sum_{k \in P} x_{ik} \leq 1, \qquad i \in M \tag{2}$$

$$x_{ik} \leq y_j, \qquad i \in M, j \in N, k \in P \tag{3}$$

$$x_{ik}, y_j \in \{0, 1\}, \quad i \in M, j \in N, k \in P \tag{4}$$

In this formulation, each solution is represented by the set of built indexes $S \subset N$, where y represents a characteristic vector of S, i.e.:

$$y_j = \begin{cases} 1, & \text{index } j \in S \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

For each query-configuration pair $(i, k) \in M \times P$, we introduce a variable:

$$x_{ik} = \begin{cases} 1, & \text{query } i \text{ uses configuration } k \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

Constraint (2) explains that each query $i \in M$ can use at most one configuration. Constraint (3) says that the queries can be served only by active configurations.

2.1 Previous Work

In [6], two methods for solving the ISP are proposed. One is a heuristics based on the solution of the suitably defined knapsack subproblem and on Lagrangean decomposition. This method is used on larger scale ISP instances, up to several thousand indexes and queries, and it produces solutions of a good quality. The second method is a branch-and-bound algorithm based on the linear programming relaxation of the model. The performance of the algorithm is enhanced by means of the preprocessing which reduces the size of the candidate index set. Optimal solutions on the ISP instances involving several hundred indexes and queries have been reported.

In [8], an improvement of the previous method by using branch-and-cut (BnC) algorithm is proposed. The problem is formulated as the Set Packing Problem that contains all clique inequalities. For initialisation of the root node in the branch-and-bound tree, a so called *LP-heuristics* is used: this is an adapted *greedy* heuristics for the UFLP, followed by a *2-interchange* heuristics. The greedy heuristics starts with an empty index set, and iteratively adds to the current set the index leading to the largest objective function increase. The 2-interchange procedure tries to improve the current solution by adding an index, removing an index, or interchanging two indexes. Analogous algorithms working on configurations instead of indexes are also applied, and the best solution is taken as a primal bound. Computational results on two different challenging types of instances (*class A* and *class B*) are reported. Although these instances contain only 50 (respectively 100) indexes and queries, presented results indicate that two new classes are more complicated than the instances tested in the previous paper with several thousand of indexes and queries.

A separation procedure for a suitably defined family of lifted odd-hole inequalities is embedded within the branch-and-cut method presented in [9]. These inequalities are obtained by applying *Chvátal-Gomory derivation* to the clique inequalities. As an initialisation heuristics, the LP-heuristics described above is used. The results of this approach compared to those obtained using CPLEX 3.0 MIP solver, show the effectiveness of the presented approach on the instances of *class A* and *class B*. However, the performance analysis indicates that the instances of *class B* are easily solvable by both exact algorithms, and that the instances of real interest are those of *class A*.

3 The Genetic Algorithm

We used traditional generational GA (containing N_{pop} individuals) with overlapping populations: N_{elit} is the number of elitist individuals that survive to the next generation. Since the evaluation of the objective value is a time-consuming operation, we store a certain amount of already calculated values in a cache-table of size N_{Cache}. Before the objective value for a certain individual is calculated, the table is checked [10, 11]. Least Recently Used caching strategy, which is simple but effective, is used for that purpose. Caching is implemented by a hash-queue data structure.

3.1 Encoding and Objective Function

We store only non-zero g_{ik} ($i \in M, k \in P$) values, since their number is small compared to the matrix dimension $|M| \cdot |P|$. For each query i we need to remember three variables: the number of g_{ik} variables that are greater than zero, the values itself stored in an array and their original indices. Thus, we enhance the evaluation of the objective function, since during the search the complete matrix need not to be considered. Figure 1 shows an example. A similar strategy has been used when the contents of the configurations are stored. Each solution S

	1	2	3	4	5	6	7	8
1	15	0	0	0	20	0	0	0
2	0	20	0	0	0	0	0	0
3	0	0	30	0	0	0	0	40

(a)

	1	2	3	4	5
1	2	15	1	20	5
2	1	20	2		
3	2	30	3	40	8

(b)

	1	2	3	4	5
1	2	20	5	15	1
2	1	20	2		
3	2	40	8	30	3

(c)

Fig. 1. An example how the values (g_{ik}) can be efficiently stored: (a) The rows represent the queries, while the columns represent possible configurations; (b) The matrix is reduced so that the first number which represents the number of non-zero values is followed by an array of values itself and their original indices. (c) The gains g_{ik} are finally sorted in the decreasing order.

is encoded using the binary vector y, given by (5). According to the values of indexes that are built, we determine the set of active configurations.

For each query i, we need to find the configuration k^* whose time gain is maximized, i.e.

$$k^* = \max_{k \in P} g_{ik}. \tag{7}$$

Finally, we need to sum all such gains over all queries, and to subtract the maintenance times of the built indexes.

The whole objective value's evaluation process can be enhanced significantly if the values g_{ik} are sorted in a decreasing order, for each query $i \in M$. That way, the first active configuration (if such exists) in the sorted array, represents the best one.

If the obtained value is negative, the building of the indexes does not pay off. In that case $y_j = 0$, for all $j \in N$ and there are no active configurations - the objective value is set to zero.

Time Complexity of the Objective Function The set of built indexes can be found in $O(n)$ time. Each configuration can contain n indexes in the worst case. Thus, all active configurations can be found in $O(np)$ time. For each query $i \in M$, finding the best configuration according to (7), can be done in $O(p)$ time. Hence, finding the best configurations for all queries needs $O(mp)$ time. However, this is only a theoretical upper bound, since the matrix (g_{ik}) is usually sparse. Finally, subtraction of the maintenance time for each built index $i \in S$ can be performed in $O(n)$ time. In total, evaluation of the objective function for each individual can be done in $O((n+m)p)$ time.

3.2 Genetic Operators and Other Aspects of the GA

In order to avoid premature convergence, multiple individuals are discarded from the population. To recombine two individuals, we use the uniform crossover described in [12]. The recombination is applied to a pair of selected individuals with probability p_{cro}.

Mutating every single gene may be unnecessarily low, since usually only several genes really mutate, while the majority of them remains the same. For a given mutation rate, the number of muted genes of one generation can be predicted. Using the *Central limit theorem*, by the Gaussian distribution, the mutation procedure is performed only on the number of randomly chosen genes. That way, a relatively small mutation probability does not create drawbacks on the performance of the simple mutation operator we used.

In our model, the mutation rate has been changed over the generations: We start with a mutation rate pm_{start}, and let it converge towards pm_{end}:

$$pm = pm_{end} + (pm_{start} - pm_{end}) \cdot 2^{\frac{-N_{gen}}{pm_{gen}}} \tag{8}$$

where N_{gen} represents the current generation, and pm_{gen} a mutation's bias-parameter. Note that for the large number of generations, the mutation rate converges towards pm_{end}, but the value itself can not be achieved.

As a selection method, we used the fine grained tournament selection, proposed in [13]. Instead of having an integer parameter N_{tour}, which represents the tournament size in the classic tournament selection, the new operator's input is a floating point parameter F_{tour} representing an average tournament size. The size of each of $N_{pop} - N_{elite}$ tournaments is chosen so that this value is on average as close as possible to F_{tour}. Such a selection scheme can significantly affect the overall performance of the GA, as it can be seen in [14].

4 Empirical Results

We have tested our GA on *class A* of randomly generated ISP instances, obtained by the generator of A. Caprara and J.J. Salazar Gonzales [9]. This generator intends to produce instances of the structure similar to real-world ISP instances, in the spirit of [6].

All instances are given with 50 queries, 50 indexes and 500 configurations. For each $j \in N$, $f_j = 100$, and for each $k \in P$, $|N_k|$ is chosen as a random integer from $\{1, \ldots, 5\}$. $|N_k|$ indexes inside of N_k are then randomly selected from N. For each $k \in P$, a set M_k is defined by randomly selecting 5 queries from M; The answer time gain g_{ik}, $\forall i \in M$, is set to zero if $i \in M \setminus M_k$, and to a random integer from $\{1, \ldots, t|N_k|\}$ otherwise (where the parameter t is given as an input).

The following setup was used for the GA, as it proved to be robust in preliminary tests: population size $N_{pop} = 150$; number of elitist individuals $N_{elit} = 100$; size of the cache $N_{Cache} = 5000$; average group size for the fine grained tournament selection $F_{tour} = 5.5$; crossover probability $p_{cro} = 0.85$, where 30% of information has been exchanged; mutation parameters $pm_{start} = 0.01$, $pm_{end} = 0.002$ and $pm_{gen} = 300$. Each run was terminated after 2 000 iterations. On all the instances we considered, this criterion allowed the GA to converge so that only minor improvements in the quality of final solutions can be expected when prolonging the runs. Thus, the main goal was to find high-quality solutions, and running times were considered only secondary.

Table 1 shows average results of our GA approach obtained on 40 instances of *class A*, running on an AMD Athlon K7/1.333GHz (Pentium IV class machine) with 256MB SDRAM memory. The GA was run 20 times on each instance.

Average time needed to detect the best value is given in $t[s]$ column, while $t_{tot}[s]$ represents the total time needed to execute all 2000 generations. On average, the best value has been reached after *gen* generations. In *sr* column, the success rate is given, i.e. in how many runs (out of 20) the optimal value has been reached.

The quality of a solution S obtained by the GA is evaluated as a percentage gap with respect to the optimal cost $cost(S^*)$:

$$gap(S) = \frac{cost(S) - cost(S^*)}{cost(S^*)} \cdot 100\% \,. \tag{9}$$

In the column *gap*, average over 20 runs is given. The values $cost(S^*)$ can be found in Table 2. Standard deviation of the average gap (σ) is also presented in the Table 1. There are additional four columns showing in how many runs the obtained gaps were less than 0.2%, 1%, 5% respectively, and in how many runs the gap was greater than 5%.

The last two columns are related to the caching: *eval* represents the average number of needed evaluations, while *cache*[%] displays savings (in percent) achieved by using the caching technique. On average, instead of making $100,000$ calls of the objective function, we took between 75% and 85% of the values from the cache-table.

Results of two MIP solvers (CPLEX and OSL) for the same set of instances are given in Table 2. OSL is tested also on AMD Athlon K7/1.333GHz machine, but CPLEX according to its license was run on Pentium III/800 MHz with 1GB memory. The columns describe: the instance name, the optimal solution, the number of CPLEX iterations, the number of branch-and-bound nodes and

Table 1. Empirical results of the GA on *class A* instances

Inst.	$t[s]$	$t_{tot}[s]$	gen	sr	err <			err >	gap	σ	$eval$	$cache[\%]$
					0.2%	1%	5%	5%				
i200.111	0.1	1.1	39	20	0	0	0	0	0.0	0.0	16844	83.2
i200.222	0.1	0.9	41	20	0	0	0	0	0.0	0.0	14684	85.3
i200.333	0.1	1.3	83	20	0	0	0	0	0.0	0.0	22449	77.6
i200.444	0.1	0.9	40	20	0	0	0	0	0.0	0.0	14485	85.5
i200.555	0.1	0.9	40	20	0	0	0	0	0.0	0.0	12448	87.6
i175.111	0.1	1.1	67	20	0	0	0	0	0.0	0.0	17151	82.9
i175.222	0.1	1.0	62	20	0	0	0	0	0.0	0.0	14941	85.1
i175.333	0.1	1.2	120	19	1	0	0	0	<0.1	<0.1	20447	79.6
i175.444	0.1	1.0	47	20	0	0	0	0	0.0	0.0	15747	84.3
i175.555	0.1	0.9	40	20	0	0	0	0	0.0	0.0	13355	86.7
i150.111	0.1	1.0	69	20	0	0	0	0	0.0	0.0	15659	84.3
i150.222	0.1	1.2	68	18	2	0	0	0	<0.1	<0.1	19950	80.1
i150.333	0.1	1.1	104	20	0	0	0	0	0.0	0.0	18381	81.6
i150.444	0.1	1.0	41	20	0	0	0	0	0.0	0.0	15448	84.6
i150.555	0.1	1.0	38	20	0	0	0	0	0.0	0.0	15244	84.8
i125.111	0.1	1.0	50	19	0	1	0	0	<0.1	<0.1	15883	84.1
i125.222	0.2	1.3	219	18	2	0	0	0	<0.1	<0.1	22142	77.9
i125.333	0.1	1.1	97	20	0	0	0	0	0.0	0.0	17742	82.3
i125.444	0.1	1.0	44	20	0	0	0	0	0.0	0.0	14433	85.6
i125.555	0.1	1.1	46	20	0	0	0	0	0.0	0.0	18594	81.4
i100.111	0.3	1.2	528	14	6	0	0	0	<0.1	<0.1	20335	79.7
i100.222	0.2	1.4	280	6	0	14	0	0	0.2	0.1	25388	74.6
i100.333	0.1	1.1	73	20	0	0	0	0	0.0	0.0	16624	83.4
i100.444	0.1	0.9	40	20	0	0	0	0	0.0	0.0	13368	86.6
i100.555	0.1	1.2	150	19	0	1	0	0	<0.1	<0.1	20469	79.5
i075.111	0.1	1.1	139	17	0	3	0	0	0.1	0.3	17299	82.7
i075.222	0.3	1.2	294	16	0	4	0	0	0.1	0.2	19688	80.3
i075.333	0.3	1.3	306	18	2	0	0	0	<0.1	<0.1	23088	76.9
i075.444	0.1	0.9	48	20	0	0	0	0	0.0	0.0	13982	86.0
i075.555	0.1	1.3	138	20	0	0	0	0	0.0	0.0	21611	78.4
i050.111	0.2	1.1	256	12	8	0	0	0	0.1	0.1	19693	80.3
i050.222	0.2	1.1	189	6	0	0	14	0	1.1	1.2	19667	80.3
i050.333	0.2	1.1	180	9	7	4	0	0	0.2	0.3	18180	81.8
i050.444	0.3	1.2	429	8	0	7	5	0	0.5	0.5	21174	78.8
i050.555	0.2	1.1	195	9	0	0	11	0	0.7	0.7	17980	82.0
i025.111	0.3	1.1	360	8	0	0	12	0	1.3	1.6	21381	78.6
i025.222	0.3	1.1	517	11	0	0	8	1	2.0	3.1	21263	78.7
i025.333	0.3	1.0	423	2	0	5	7	6	2.9	2.6	19434	80.6
i025.444	0.2	1.0	193	7	0	0	1	12	5.1	4.1	17808	82.2
i025.555	0.4	1.1	623	2	0	5	13	0	0.8	0.5	21784	78.2

Table 2. Results obtained by two MIP solvers: CPLEX and OSL

Instance	$cost(S^*)$	CPLEX			OSL	
		Iterations	Nodes	$t[s]$	Iterations	$t[s]$
i200.111	41102	190	0	0.5	3866	22
i200.222	40412	319	4	7.6	5364	56
i200.333	41034	383	13	8.2	5834	70
i200.444	41030	177	0	0.4	4316	24
i200.555	40706	191	0	0.4	3845	20
i175.111	35452	220	0	0.7	5233	54
i175.222	34803	937	26	17.4	6445	71
i175.333	35355	1477	66	18.4	18565	232
i175.444	35352	221	0	0.5	4509	24
i175.555	35097	211	0	0.5	4097	22
i150.111	29781	348	2	10.6	5577	64
i150.222	29199	3829	172	26.5	26104	281
i150.333	29712	4027	220	22.1	45093	546
i150.444	29703	604	7	18.3	6056	72
i150.555	29458	531	8	13.0	6751	78
i125.111	24121	1191	39	18.4	8375	105
i125.222	23612	20619	959	63.3	222424	2349
i125.333	24088	17458	1038	41.1	137939	1555
i125.444	24054	2325	69	24.1	10920	127
i125.555	23839	2338	101	29.2	11763	127
i100.111	18479	12752	465	82.7	62798	636
i100.222	18138	40775	2001	190.5	1792061	18071
i100.333	18486	57212	3346	187.4	2202795	28493
i100.444	18419	7863	314	72.3	51660	496
i100.555	18255	21598	994	79.9	212854	2037
i075.111	12956	60188	2376	288.5	768490	7157
i075.222	12698	83844	4100	349.2	-	-
i075.333	12899	117720	5714	486.9	-	-
i075.444	12777	149620	4602	495.1	2153357	18380
i075.555	12761	147597	7658	584.5	2245747	19196
i050.111	7475	1138414	36611	4151.2	-	-
i050.222	7400	1005745	40367	3719.7	-	-
i050.333	7423	2383662	124310	9900.6	-	-
i050.444	7312	2309766	98514	9783.8	-	-
i050.555	7384	1764340	61630	7664.2	-	-
i025.111	2356	70114754	1039922	249709.9	-	-
i025.222	2347	19846560	282383	66569.9	-	-
i025.333	2339	131290943	3945602	441895.3	-	-
i025.444	2278	90164493	1596756	294388.1	-	-
i025.555	2289	71976083	1178812	170476.5	-	-

Table 3. Comparing results obtained by LP-heuristics, BnC and GA

Instance	BnC (HP9000/720)			GA (AMD K5)	
	root-$t[s]$	root-$gap[\%]$	$t_{tot}[s]$	$t[s]$	$t_{tot}[s]$
i200.111	1.1	0	1.1	1.5	22.9
i200.222	1.0	0	1.0	1.6	21.2
i200.333	0.9	0	0.9	2.3	28.0
i200.444	1.0	0	1.0	1.3	20.1
i200.555	1.4	0	1.4	1.5	18.7
i175.111	1.2	0	1.2	2.0	22.5
i175.222	1.3	0	1.3	1.6	21.4
i175.333	1.1	0	1.1	3.8	25.4
i175.444	1.1	0	1.1	1.6	21.0
i175.555	2.0	0	2.0	1.3	19.1
i150.111	2.0	0	2.1	1.6	20.6
i150.222	2.5	0	2.5	1.9	24.1
i150.333	1.3	0	1.3	3.3	24.4
i150.444	2.1	0	2.1	1.6	21.3
i150.555	3.3	0	3.3	1.4	20.7
i125.111	4.7	0	4.8	1.5	20.5
i125.222	5.2	0	5.2	2.0	25.0
i125.333	1.6	0	1.6	2.8	23.2
i125.444	4.3	0	4.3	1.5	20.3
i125.555	7.4	0	7.4	2.0	24.1
i100.111	32.6	0	33.3	8.1	24.9
i100.222	25.4	0	25.4	6.9	27.8
i100.333	4.8	0	4.8	2.4	22.4
i100.444	31.6	0.2	37.6	1.5	18.8
i100.555	26.6	0.3	45.2	5.2	26.8
i075.111	83.8	1.9	391.2	3.6	22.3
i075.222	78.9	1.8	773.4	9.7	26.0
i075.333	47.5	0.5	75.9	6.1	27.4
i075.444	92.1	2.3	815.2	1.9	20.3
i075.555	79.1	1.9	457.2	3.4	26.0
i050.111	139.6	7.4	7553.5	4.5	23.8
i050.222	152.4	7.1	6583.4	7.6	25.6
i050.333	134.2	5.2	1692.0	7.7	24.9
i050.444	192.5	8.0	9284.9	5.8	24.5
i050.555	157.4	8.1	10566.7	2.8	23.4
i025.111	565.3	37.7	> 50000	7.0	23.2
i025.222	595.5	38.4	> 50000	6.3	23.3
i025.333	585.5	36.0	> 50000	7.0	22.7
i025.444	691.3	35.8	> 50000	7.6	21.4
i025.555	584.6	34.9	> 50000	5.2	22.2

CPLEX's running time in seconds; OSL's number of iterations and its running time.

Results of the branch-and-cut algorithm described in [8, 9] directly compared to the GA are given in Table 3. It is clear that AMD Athlon K7/1.333GHz is a more powerful machine than the one used in [9] (HP 9000/720 at 80 MHz, 59 Specs, 58 MIPS, 18 Mflops), but the exact speedup factor is unknown. This factor is unfortunately not constant, and greatly depends on instance's characteristics. For a fair comparison of the performance, the GA with the same set of parameters is tested also on AMD K5/100MHz machine with 64MB memory (Pentium I class machine). The performance on this machine is very similar to that of HP 9000/720.

The first two columns of Table 3 represent the running time of the LP-heuristics (described in Sect. 2.1) (root-$t[s]$) and its gap (root-$gap[\%]$), respectively. These values are followed by BnC's total time needed to obtain optimal solution, GA's average time needed to detect the best value, and GA's total time needed to execute all 2000 generations.

For the instances of type i200, i175, i150 and i125, solutions obtained by the LP-heuristics are already optimal, and no branching has been performed. For some of these instances is the BnC even faster than our GA. On the other side, the instances of type i100, i075, i050 and i025 seem to be quite challenging for the exact algorithms, since BnC's running time grows up exponentially. Even the initialisation time is at least four times greater than GA's average running time. The quality of initial solutions is also very bad (for the group i025, gaps are even greater than 30%).

In contrast to BnC's exponential nature, the GA performs very efficiently, keeping a satisfying quality of the obtained solutions (in the worst case, the success rate was 10%, and the total running time was less than 2 seconds).

In 50% of the instances, the GA had a success rate of 100%, while in 70% of them, the average gap was not greater than 0.1%. In only one out of 40 tested instances, the average gap was greater than 3% (i025.444).

For extended results obtained on the *class B* of instances, see `http://www.geocities.com/jkratica/instances/`. These instances appear to be easily solvable for BnC, CPLEX and OSL (optimal solutions are usually achieved within few seconds).

5 Conclusions

We proposed a genetic algorithm for the Index Selection Problem based on binary encoding, efficient data structures (for the evaluation of the objective function), on the uniform crossover, and simple mutation. The algorithm is tested on the class of challenging instances known from the literature. Obtained results indicate its efficiency and reliability.

The algorithm fits well into the parallel implementation and different island models, described in [15], should be tested. Incorporation of some problem-dependent variation operators, or a local-search procedures, could possibly make the algorithm more powerful for the instances of a larger size.

Acknowledgements

We thank to A. Caprara and J.J. Salazar Gonzales, the authors of [8, 9], for providing us the generator of test-instances for the ISP.

Ivana Ljubić is supported by the Doctoral Scholarship Programme of the Austrian Academy of Sciences (DOC).

References

1. J. Krarup and P.M. Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12:36–81, 1983.
2. D. W. Tcha and B.-Y. Lee. A branch-and-bound algorithm for the multilevel uncapacitated facility location problem. *European Jornal of Operational Research*, 18(1):35–43, 1984.
3. Y. A. Kochetov and E. N. Goncharov. Probabilistic tabu search algorithm for the multi-stage uncapacitated facility location problem. In B. Fleischmann, R. Lasch, U. Derigs, W. Domschke, and U. Rieder, editors, *Operations Research Proceedings 2000*, pages 65–70. Springer, 2000.
4. E. Barcucci, R. Pinzani, and R. Sprugnoli. Optimal selection of secondary indexes. *IEEE Transactions on Software Engineering*, 16, 1990.
5. M.Y.L. Ip, L.V. Saxton, and V.V. Raghavan. On the selection of an optimal set of indexes. *IEEE Transactions on Software Engineering*, pages 135–143, 1983.
6. A. Caprara, M. Fischetti, and D. Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), 1995.
7. S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13:91–128, 1988.
8. A. Caprara and J.J. Salazar Gonzalez. A branch-and-cut algorithm for a generalization of the uncapacitated facility location. *Problem Trabajos de Operativa - TOP*, 4(1):135–163, 1996.
9. A. Caprara and J.J Salazar Gonzalez. Separating lifted odd-hole inequalities to solve the index selection problem. *Discrete Applied Mathematics*, 92:111–134, 1999.
10. J. Kratica. Improving performances of the genetic algorithm by caching. *Computers and Artificial Intelligence*, 18(3):271–283, 1999.
11. J. Kratica. *Parallelization of Genetic Algorithms for Solving Some NP-complete Problems (in Serbian)*. PhD thesis, Faculty of Mathematics, Belgrade, 2000.
12. G. Syswerda. Uniform crossover in genetic algorithms. In *3th International Conference on Genetic ALgorithms, ICGA'89*, pages 2–9, Internet, 1989. Morgan Kaufmann, San Mateo, Calif.
13. V. Filipović. Proposition for improvement tournament selection operator in genetic algorithms (in serbian). Master's thesis, Faculty of Mathematics, Belgrade, 1998.
14. V. Filipović, J. Kratica, D. Tošić, and I. Ljubić. Fine grained tournament selection for the simple plant location problem. In *5th Online World Conference on Soft Computing Methods in Industrial Applications, WSC5*, pages 152–158, Internet, 2000. ISBN: 951-22-5205-8.
15. Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Boston, 2000.