Serdica J. Computing **3** (2009), 299–308

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

ON SOLVING THE MAXIMUM BETWEENNESS PROBLEM USING GENETIC ALGORITHMS

Aleksandar Savić

Abstract. In this paper a genetic algorithm (GA) is applied on Maximum Betweennes Problem (MBP). The maximum of the objective function is obtained by finding a permutation which satisfies a maximal number of betweenness constraints. Every permutation considered is genetically coded with an integer representation. Standard operators are used in the GA. Instances in the experimental results are randomly generated. For smaller dimensions, optimal solutions of MBP are obtained by total enumeration. For those instances, the GA reached all optimal solutions except one. The GA also obtained results for larger instances of up to 50 elements and 1000 triples. The running time of execution and finding optimal results is quite short.

1. Introduction. The problem of maximum betweenness is well-known and can be found in most compendiums of NP problems. The problem can be summarised put as follows: Let there be a finite set A and a collection C of triples (x, y, z) of distinct elements from A. Find a $1:1$ function $f : A \to [1..|A|]$

ACM Computing Classification System (1998): G.1.6, I.2.8.
Key words: Evolutionary approach. Genetic algorithms. Betweenness Problem.

such that the number of triples that satisfies either $f(x) < f(y) < f(z)$ or $f(x) > f(y) > f(z)$ is maximal. In the late seventies Opatrny [14] showed that finding n totally ordered elements that satisfy m such betweenness constraints is NP-complete. Furthermore, the problem is MAX SNP complete, and for every $a > 47/48$ finding a total order that satisfies at least a of the m constraints is NP-hard (even if all the constraints are satisfiable) [1]. The origin of this problem is in molecular biology and correct mapping of chromosomes which can be seen in [2]. It is easy to see that the lower bound of the objective function must be equal or greater than $\frac{1}{3}$ of the constraints, because even if we order the elements randomly, at least one-third of the betweenness constraints will be satisfied. Until now, all considerations of this problem were purely theoretical [5, 6, 8]. Because of all this, it was of interest to try an application of some metaheuristic on this problem, especially in view that, in general, the problem becomes NP-hard.

2. Mathematical model. In MBP the given set A is finite. Because of this, every element of A can be denoted with $\{a_i\}$ $i = 1, \ldots, n$, where $n = |A|$. Thus in the finite set A a linear ordering is introduced so that now it is known which element is the first, the second and so on, until the last. Thus the MBP problem is to find a $1:1$ function $f : \{a_1, a_2, \ldots, a_n\} \to [1..|A|]$. This implies that it is enough to find a $1:1$ function $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Any $1:1$ function from a finite set to the same finite set is actually a permutation of that finite set. In conclusion, it is enough to find a permutation p of the set $1, 2, \ldots, n$ which has a maximal number of satisfied betweenness constraints. In this paper finding such number is considered by using the genetic algorithm (GA).

3. Proposed GA method. In recent times metaheuristics have started to play a major role in solving problems whose solving couldn't even be considered a decade ago because of the computational time and resources. Now with metaheuristics researchers can handle even the hardest NP-hard problems. Among the more well-known metaheuristics are genetic algorithms (GA). As their name suggests, they have roots in biology and real genetics. GA emulate nature in the manner that they are stohastic methods for finding solutions of various problems similar to nature's finding answers to environmental problems. As nature, GA works with individuals which constitute a population. And also, as in nature, GA works to find individuals better suited to cope with the problem. These better individuals are allowed to transfer their good qualities to the next generations of individuals. This is done through genetic operators of crossover and mutation. Deciding which individuals will pass their good qualities in GA is done by evaluating a fitness function. The betterment of individuals is iteratively repeated until the optimum of fitness function is achieved or some other stop criterion. The field of applications of GAs in recent years has grown vast and can't be covered in this paper's scope. Some descriptions can be found in [13]. Extensive computational experience on various optimization problems shows that GA often produces high-quality solutions in a reasonable time. Some recent applications are: hub location [10], metric dimension of graphs [11], traveling purchase [7], multiprocessor open shop [12], order acceptance [15] and airline crew-pairing and rostering [16].

The encoding of individuals in this paper is integer. The genetic code in this case has $n-1$ elements. Since the i-th element of the gene should be between 0 and $n-i-1$ it is obtained by division by $mod(n-i)$. If the gene has the value q in position m then the mth element of the permutation will be obtained as the $(q+1)$-th element from the line of unused elements. Obtaining the nth element of the permutation is unequivocal because there is only one element left unused.

For example, let $n=4$, then gene 120 generates permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$. The number 1 in the gene denotes that the first element of permutation should be the second element of 1234, which is 2. The number 2 in the gene denotes that the second element of the permutation should be the third element of 134, which is 4. The number 0 in the gene means that the third element in the permutation should be the first element from 13, which is 1. The last element is unique, that is the only element which is left over and that is 3. Similarly gene 301 generates permutation $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$.

The advantage of this method is that all individuals are feasible. Crossover and mutation operators will again give a feasible individual because it will again be a permutation. The disadvantage is that with the crossover operator in the next generation from two numbers in parent permutations it is not certain that both numbers will figure in descendant permutation.

Let us consider two genes from the previous example. They are 120 and 301. In this paper one point crossover operator is used which means that the first numbers in the genes will not be exchanged. As a result of the crossover operator from parent genes 120 and 301 the descendant genes are 101 and 320. This implies that from permutations $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$ and $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$, two descendant permutations are obtained, which are $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$ and $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}$. As can be seen in the second position the descendant permutations have numbers 1 and 3 though their parents have numbers 4 and 1.

The most important parts of the proposed GA are: Crossover and Mutation, Selection and Fitness Function. In the following text their function and use will be explained in detail.

In the population, as in nature, we differentiate elite and non-elite individuals. Their treatment with GA is accordingly also different. All elite individuals, their number being N_{elite}, are automatically passed to the next generation. All non-elite individuals, their number being $N_{nnel} = N_{pop} - N_{elite}$, are subject to genetic operators. This differentiation reduces computational time because evaluation of objective function in elite individuals does not change from generation to generation and needs to be calculated only once, in the first generation.

The fitness function f_{ind} is responsible for deciding which individual is fit to continue in evolution and which is not. Its values are computed by scaling the objective values obj_{ind} of all individuals into the interval [0,1], so that the best suitable individual $ind_{\max}$ gets the value 1 and the worst $ind_{\min}$ gets 0. Explicitly, $f_{ind} = \dfrac{obj_{ind_{\min}} - obj_{ind}}{obj_{ind_{\min}} - obj_{ind_{\max}}}$. After evaluating the objective function of all individuals in the generation and accordingly calculating the values of the fitness function, the individuals are arranged in non-increasing order by their best fitness: $f_1 \geq f_2 \geq \cdots \geq f_{N_{pop}}$, where N_{pop} is the number of individuals in the population.

After obtaining the values of the fitness function, two cases arise which must be considered. First, the number of individuals with the same objective function but different genetic code must be limited by some constant N_{rv}. This is done so that other genetic material with different potential can stay in the genetic pool. To obtain this, the fitness of all individuals with the same value of the objective function but different genetic material will be set to 0 except the first N_{rv}. Also, to avoid unwanted domination, individuals with the same genetic code in population must be avoided, and their fitness is set to 0 for all individuals, except the first one.

After this we decide by some criterion which individuals are elite and which are not. Non-elite individuals are now subject to selection for continuation of their good qualities.

Selection of non-elite individuals is done through tournaments. A number of individuals set a priori to participate in the tournament is randomly chosen. This a priori number is called tournament size. The purpose of the tournament is to allow its winner to pass in to the next generation. The winner of the tournament is the individual with the highest value of the objective function. To ensure that populations do not shrink in size, the number of tournaments is equal to the number of non-elite individuals N_{nnel}, so that exactly N_{nnel} parents can be chosen for crossover. It is allowed for the same individual from the current generation to participate in more than one tournament.

In the standard tournament selection, the tournament size is integer, which can hinder the efficiency of the algorithm. To avoid this problem, the implementation of the selection is improved with a tournament selection operator, Fine-grained tournament selection – FGTS [4]. Here, the tournament size is a real parameter F_{tour}, which represents the preferable average tournament size. Now, there are two types of tournaments. One is held k_1 times, with tournament size $\lfloor F_{tour} \rfloor$, and the other is held k_2 times, with tournament size $\lceil F_{tour} \rceil$. Hence $F_{tour} \approx \dfrac{k_1 \cdot \lfloor F_{tour} \rfloor + k_2 \cdot \lceil F_{tour} \rceil}{N_{nnel}}$.

An adequate ratio of the number of elite and the number of non-elite individuals is necessary to achieve the most satisfactory results of GA. For example, for $N_{pop} = 150$ the adequate proportion is $N_{elite} = 100$ and $N_{nnel} = 50$. The parameters k_1 and k_2 in FGTS which determine the tournament size are 30 and 20, respectively. Also, for $N_{pop} = 150$ the adequate maximum of individuals with the same fitness is $N_{rv} = 40$.

FGTS performs best with the value of F_{tour} set to 5.4, as can be seen in [4]. The same value is used in this work because the numerical and statistical data presented here are exceptional. For detailed information about FGTS see [4].

After selection of non-elite individuals, these individuals can be processed by the crossover operator. They are now randomly paired in $\lfloor N_{nnel}/2 \rfloor$ pairs. The intention of the GA algorithm through its crossover operator is that two selected individuals exchange genes and produce offspring which will have improved qualities and especially fitness function than the parent individuals. In every crossover two offsprings are produced which will replace their parents in the next generation. In this paper a standard one-point crossover operator is used. These standard operators exchange whole genes between the genetic codes of the parents to produce offspring. The probability of realization of the crossover operator is 85%. This means that approximately 85% of the pairs of individuals will exchange genes.

For the purposes of this paper in the genetic algorithm simple mutation operator is used. The direct application of this operator changes a randomly selected gene with some mutation rate. Also, in the operator is installed a modification for dealing with problems of frozen genes. Frozen genes are those genes that are in a certain position in all individuals in the population. They can't be changed with selection or a crossover operator because they are widespread through all population. The existence of frozen genes reduces the search space and increases the possibility of premature convergence. For example, if there are q frozen genes in the population, then the search space will be $q!$ times smaller.

The modification of the simple mutation operator in GA is that the mutation rate is increased only on frozen genes. To implement this for each generation it must be determined which genes are frozen. Then the mutation rate for these genes is increased. For the proposed GA in this paper the increased mutation rate for frozen genes is 2.5 timesgreater than the mutation rate on unfrozen genes.

The initialization of GA is random. This guaranties the most heterogeneous and diversified genetic pool of population in the first generation.

Finally, for improved performance of the proposed GA a caching technique is implemented. The main idea is to avoid evaluation of the objective function for individuals with the same genetic code. The values of the objective function that are already computed are stored by the least recently used (LRU) caching technique into the hash-queue data structure. Now, if a new individual has the same genetic code as some old one, the value of his objective function is not computed, but found in cache memory. This results in significant time savings. The limitation for the already calculated values of objective function in this implementation is 5000. If the cache memory is full then we remove the least recently used cash memory block. Detailed information about caching GA can be found in [9].

4. Experimental results. All computations were executed on a 2.8 GHz PC computer with 2 Gb RAM under Windows. The genetic algorithm was coded in the C programming language. For experimental testings, in this paper, randomly generated instances were used. The names of instances reflects the dimension of the problem, for example mbp-11-100 indicates that set A has 11 elements and that collection C has 100 triples from set A. These instances include different numbers of elements in set A (N = 10, 11, 12, 15, 20, 30, 50) and different numbers of triples in C (ranging from 20 to 1000).

Instances are generated on the following principles. An apriori number of elements of A (n) and number of triples (nt) in collection C are selected. After this, a variation of 3 elements from set $\{1, \dots, n\}$, where elements could not reoccur, was generated randomly. After generating a triple, it was checked if that triple was already in the collection of generated triples C'. If it wasn't, the triple would be included in C' and the process would be continued until the number of generated triples was equal to the a priori selected number nt and $C' = C$.

In order to check GA, smaller instances were solved by Total Enumeration. To do this, it was necessary to generate all permutations of the set $\{1, \dots, n\}$ and after that to find which permutations satisfies a maximum number of betweenness constraints $(p(x) < p(y) < p(z) \vee p(x) > p(y) > p(z))$ for triples (x, y, z) from collection C. For any specific permutation it wasn't time-consuming to check how many betweenness constraints were satisfied but the number of permutations grows rapidly with the increase of n. Already for $n = 12$ the number of permutations is $12! = 479001600$, so, Total Enumeration solved only those instances where the number of elements of the set A was smaller or equal to 12. The results obtained by Total Enumeration are given in Table 1. The first column contains running times and second contains optimal solutions.

Table 1: Total Enumeraton results on MBP instances

Instance name	t_{tot}	*sol*
mbp_10_100	6.359	50.000000
mbp_10_20	2.234	16.000000
mbp_10_50	3.671	29.000000
mbp_11_100	72.625	55.000000
mbp_11_20	26.203	14.000000
mbp_11_50	41.171	33.000000
mbp_12_100	861.000	56.000000
mbp_12_20	331.156	17.000000
mbp_12_50	520.125	33.000000

The finishing criterion of GA is the maximal number of generations $N_{gen} = 1000$. The algorithm also stops if the best individual or the best objective value remains unchanged through $N_{rep} = 500$ successive generations. Since the results of GA are nondeterministic, the GA was run 20 times on each instance.

Table 2 summarizes the GA results on these instances. In the first column the names of the instances are given. The instance's name carries information about the number of elements in set A and the number of triples M in collection C. For example, the instance mbp_11_100 is an instance which has $N = 11$ elements in set A and $M = 100$ triples in collection C.

The best GA values GA_{sol} are given in the last column. The mark *opt* is given if an optimal solution is reached and there is no difference between that solution and solution obtained by Total Enumeration.

The average times needed to detect the best GA values are given in column t. The solution quality in all 20 executions is evaluated as a percentage gap named *agap*, with respect to the optimal solution sol_{opt}, with standard deviation σ of the average gap. A percentage gap *agap* is defined as $agap = \frac{1}{20}\sum_{i=1}^{20} gap_i$, where

$gap_i = 100 * \frac{GA_i - GA_{best}}{GA_{best}}$ and GA_i represents the GA solution obtained in the ith run, while σ is the standard deviation of gap_i, $i = 1, 2, \ldots, 20$, obtained by formula $\sigma = \sqrt{\frac{1}{20} \sum_{i=1}^{20} (gap_i - agap)^2}$. The next two columns are related to the caching: *eval* represents the average number of evaluations, while *cache* displays savings (in percent) achieved by using caching technique.

Table 2: GA results on MBP smaller instances

Instance name	t_{tot}	t	*agap*	σ	*eval*	*cache*	*GA*	*opt*
	(opt)	(sec)		(sec)		(%)	*sol*	
mbp_10_100	0.652	0.160	2.417	1.018	12302.1	59.9	50.000000	*opt*
mbp_10_20	0.194	0.011	1.563	2.777	10264.4	61.7	16.000000	*opt*
mbp_10_50	0.195	0.007	0.172	0.771	10444.1	60.5	29.000000	*opt*
mbp_11_100	0.243	0.042	2.636	2.669	12056.5	59.8	55.000000	*opt*
mbp_11_20	0.200	0.008	2.500	3.495	12022.4	54.1	14.000000	*opt*
mbp_11_50	0.214	0.018	2.273	1.346	11305.5	58.5	33.000000	*opt*
mbp_12_100	0.246	0.043	3.125	3.219	11377.8	62.1	56.000000	*opt*
mbp_12_20	0.197	0.014	2.353	2.957	11008.5	59.0	17.000000	*opt*
mbp_12_50	0.228	0.032	3.030	1.390	10994.8	62.5	33.000000	

In Table 3 are given results obtained by GA on larger instances. Because Total Enumeration could not reach solutions for these larger instances there is no *opt* remark next to the results. Other notations are the same as in Table 2.

As can be seen in Tables 2 and 3, the running time of GA on all instances is really small. The average execution on the largest instance is smaller than 2.5 seconds.

Comparing Tables 1 and 2 it can be seen that results of GA algorithms reached *opt* solution in all instances except one. In all other instances GA algorithm reached the optimum obtained by Total Enumeration. For all instances the working time of Total Enumeration is much longer than the running time of GA. The running time of GA in these smaller instances is shorter than half a second.

5. Conclusions.

The GA metaheuristic for solving MBP is presented in this paper. An integer representation of the necessary permutations was used. Fine-grained tournament selection refinement was applied in the selection process. One-point crossover and simple mutation with frozen genes were used. Computational performance of GA was improved by caching. For almost all instances

Table 3: GA results on larger MBP instances

Instance name	t_{tot} *(opt)*	t (sec)	*agap*	σ (sec)	*eval*	*cache* (%)	*GA sol*
mbp_15_200	0.289	0.047	2.048	2.458	13900.3	52.8	105.000000
mbp_15_30	0.217	0.025	9.000	3.866	13338.8	52.9	25.000000
mbp_15_70	0.231	0.026	4.457	2.053	12962.0	54.0	46.000000
mbp_20_100	0.398	0.118	4.462	2.636	22190.9	36.3	65.000000
mbp_20_200	0.400	0.079	1.239	0.926	19494.4	36.4	113.000000
mbp_20_40	0.340	0.073	10.278	5.635	20126.8	35.9	36.000000
mbp_30_150	0.627	0.220	6.225	2.741	25676.3	31.6	102.000000
mbp_30_300	0.749	0.307	4.798	2.351	26433.8	31.8	173.000000
mbp_30_60	0.538	0.145	6.373	4.214	23549.3	30.9	51.000000
mbp_50_100	1.147	0.458	7.143	4.124	31936.7	18.4	84.000000
mbp_50_1000	2.169	1.391	4.187	2.140	37277.3	19.2	504.000000
mbp_50_200	1.385	0.766	5.643	3.268	36507.5	18.5	140.000000
mbp_50_400	1.535	0.793	4.104	2.622	35736.3	18.8	240.000000

(all but one) GA calculates solutions that match the optimal ones obtained by Total Enumeration. GA results were obtained in very short running time.

Based on the results, GA has the potential to be a useful metaheuristic for solving other similar problems. The parallelization of the GA and its hybridization with exact methods are also promising directions of future work.

REFERENCES

[1] Chor B., M. Sudan. A geometric approach to betweenness. *SIAM J. Discrete math.*, **11** (1998), No. 4, 511–523.

[2] Christof T., M. Junger, J. Kececioglu, P. Mutzel, G. Reinelt. A branch-and-cut approach to physical mapping with end-probes In: Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB-97), 1997, 84–92.

[3] Djurić B., J. Kratica, D. Tošić, V. Filipović. Solving the maximally balanced connected partition problem in graphs by using genetic algorithm. *Computing and Informatics*, **27** (2008), No. 3, 341–354.

[4] Filipović V. Fine-grained Tournament Selection Operator in Genetic Algorithms. *Computing and Informatics*, **22** (2003), 143–161.

[5] Goerdt A. On random betweenness constraints II. Submitted for publication 2009.

[6] Goerdt A., A. Lanka. On random betweenness constraints. Submitted for publication 2009.

[7] Goldbarg M. C., L. B. Bagi, E. F. G. Goldbarg. Transgenetic algorithm for the Traveling Purchaser Problem. *European Journal of Operational Research*, **199** (2009), 36–45.

[8] Guttmann W., M. Maucher. Variations on an ordering theme with constraints. In: Proceedings Fourth IFIP International Conference on Theoretical Computer Science TCS, 2006, 77–90.

[9] Kratica J. Improving Performances of the Genetic Algorithm by Caching. *Computers and Artificial Intelligence*, **18** (1999), 271–283.

[10] Kratica J., Z. Stanimirović , D. Tošić , V. Filipović. Two genetic algorithms for solving the uncapacitated single allocation p-hub median problem. *European Journal of Operational Research*, **182** (2007), No. 1, 15–28.

[11] Kratica J., V. Kovačević-Vujčić, M. Čangalović. Computing strong metric dimension of some special classes of graphs by genetic algorithms. *Yugoslav Journal of Operations Research*, **18** (2008), 143–151.

[12] Marie E. M. A genetic algorithm for the proportionate multiprocessor open shop. *Computers & Operations Research*, **36** (2009), Issue 9, 2601–2618.

[13] Mitchell M. Introduction to genetic algorithms. MIT Press, Cambridge, Massachusetts, 1999.

[14] Opatrny J. Total ordering problem, *SIAM J. Comput.*, **8** (1979), 111–114.

[15] Rom W.O., S.A. Slotnick. Order acceptance using genetic algorithms. *Computers & Operations Research*, **36** (2009), Issue 6, 1758–1767.

[16] Souai N., J. Teghem. Genetic algorithm based approach for the integrated airline crew-pairing and rostering problem. *European Journal of Operational Research*, **199** (2009), Issue 3, 674–683.

Faculty of Mathematics
University of Belgrade
Studentski trg 16/IV
11 000 Belgrade, Serbia
e-mail: `aleks3rd@gmail.com`

Received June 9, 2009
Final Accepted July 20, 2009