ON SOLVING THE MAXIMUM BETWEENNESS PROBLEM USING GENETIC ALGORITHMS

Aleksandar Savić

Abstract. In this paper a genetic algorithm (GA) is applied on Maximum Betweenness 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 $f : A \rightarrow [1..|A|]$
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 $f : \{a_1, a_2, \ldots, a_n\} \rightarrow [1..|A|]$. This implies that it is enough to find a $1:1$ function $f : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$. 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 stochastic 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 \( \text{mod}(n - i) \). If the gene has the value \( q \) in position \( m \) then the \( m \)-th element of the permutation will be obtained as the \( (q + 1) \)-th element from the line of unused elements. Obtaining the \( n \)-th 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 \( N_{\text{elite}} \), are automatically passed to the next generation. All non-elite individuals, their number being \( N_{\text{nnel}} = N_{\text{pop}} - N_{\text{elite}} \), 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 \( f_{\text{ind}} \) 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 \( \text{obj}_{\text{ind}} \) of all individuals into the interval \([0,1]\), so that the best suitable individual \( \text{ind}_{\text{max}} \) gets the value 1 and the worst \( \text{ind}_{\text{min}} \) gets 0. Explicitly, \( f_{\text{ind}} = \frac{\text{obj}_{\text{ind}} - \text{obj}_{\text{ind}_{\text{min}}}}{\text{obj}_{\text{ind}_{\text{min}}} - \text{obj}_{\text{ind}_{\text{max}}}} \). 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: \( f_1 \geq f_2 \geq \cdots \geq f_{N_{\text{pop}}} \), where \( N_{\text{pop}} \) 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 \( N_{\text{rv}} \). 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 \( N_{\text{rv}} \). 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 \( N_{\text{nnel}} \), so that exactly \( N_{\text{nnel}} \) 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 oper-
ator, Fine-grained tournament selection – FGTS [4]. Here, the tournament size is
a real parameter $F_{\text{tour}}$, which represents the preferable average tournament size.
Now, there are two types of tournaments. One is held $k_1$ times, with tournament
size $\lfloor F_{\text{tour}} \rfloor$, and the other is held $k_2$ times, with tournament size $\lceil F_{\text{tour}} \rceil$. Hence

$$F_{\text{tour}} \approx \frac{k_1 \cdot \lfloor F_{\text{tour}} \rfloor + k_2 \cdot \lceil F_{\text{tour}} \rceil}{N_{\text{nuel}}}.$$ 

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

FGTS performs best with the value of $F_{\text{tour}}$ 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 $\lfloor N_{\text{nuel}}/2 \rfloor$ pairs.
The intention of the GA algorithm through its crossover operator is that two
selected individuals exchange genes and produce offspring which will have im-
proved 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 mod-
ification 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 times greater than the mutation rate on unfrozen genes.

The initialization of GA is random. This guarantees 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 cache 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 reflect 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 \( \{1, \ldots, n\} \), 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 \( \{1, \ldots, n\} \) and
On solving the maximum betweenness problem . . .

after that to find which permutations satisfies a maximum number of betweenness constraints \((p(x) < p(y) < p(z) \lor p(x) > p(y) > p(z))\) 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 Enumeration results on MBP instances

<table>
<thead>
<tr>
<th>Instance name</th>
<th>(t_{tot})</th>
<th>(sol)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbp_10_100</td>
<td>6.359</td>
<td>50.000000</td>
</tr>
<tr>
<td>mbp_10_20</td>
<td>2.234</td>
<td>16.000000</td>
</tr>
<tr>
<td>mbp_10_50</td>
<td>3.671</td>
<td>29.000000</td>
</tr>
<tr>
<td>mbp_11_100</td>
<td>72.625</td>
<td>55.000000</td>
</tr>
<tr>
<td>mbp_11_20</td>
<td>26.203</td>
<td>14.000000</td>
</tr>
<tr>
<td>mbp_11_50</td>
<td>41.171</td>
<td>33.000000</td>
</tr>
<tr>
<td>mbp_12_100</td>
<td>861.000</td>
<td>56.000000</td>
</tr>
<tr>
<td>mbp_12_20</td>
<td>331.156</td>
<td>17.000000</td>
</tr>
<tr>
<td>mbp_12_50</td>
<td>520.125</td>
<td>33.000000</td>
</tr>
</tbody>
</table>

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 \(\sigma\) of the average gap. A percentage gap \(agap\) is defined as \(agap = \frac{1}{20} \sum_{i=1}^{20} gap_i\), where
306  

Aleksandar Savić

\[
gap_i = 100 \times \frac{GA_i - GA_{best}}{GA_{best}}
\]

and \( GA_i \) represents the GA solution obtained in the \( i \)th run, while \( \sigma \) is the standard deviation of \( \gap_i, \ i = 1, 2, \ldots, 20 \), obtained by the 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

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

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

<table>
<thead>
<tr>
<th>Instance name</th>
<th>$t_{tot}$ (sec)</th>
<th>$t_{agap}$ (sec)</th>
<th>$\sigma$</th>
<th>eval</th>
<th>cache (%)</th>
<th>GA sol</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbp_15,200</td>
<td>0.289</td>
<td>2.048</td>
<td>2.458</td>
<td>13900.3</td>
<td>52.8</td>
<td>105.000000</td>
</tr>
<tr>
<td>mbp_15,30</td>
<td>0.217</td>
<td>0.047</td>
<td>9.000</td>
<td>3.866</td>
<td>13338.8</td>
<td>52.9</td>
</tr>
<tr>
<td>mbp_15,70</td>
<td>0.231</td>
<td>4.457</td>
<td>2.053</td>
<td>12962.0</td>
<td>54.0</td>
<td>46.000000</td>
</tr>
<tr>
<td>mbp_20,100</td>
<td>0.398</td>
<td>4.462</td>
<td>2.636</td>
<td>22190.9</td>
<td>36.3</td>
<td>65.000000</td>
</tr>
<tr>
<td>mbp_20,200</td>
<td>0.400</td>
<td>1.239</td>
<td>0.926</td>
<td>19494.4</td>
<td>36.4</td>
<td>113.000000</td>
</tr>
<tr>
<td>mbp_20,40</td>
<td>0.340</td>
<td>10.278</td>
<td>5.635</td>
<td>20126.8</td>
<td>35.9</td>
<td>36.000000</td>
</tr>
<tr>
<td>mbp_30,150</td>
<td>0.627</td>
<td>6.225</td>
<td>2.741</td>
<td>25676.3</td>
<td>31.6</td>
<td>102.000000</td>
</tr>
<tr>
<td>mbp_30,300</td>
<td>0.749</td>
<td>4.798</td>
<td>2.351</td>
<td>26433.8</td>
<td>31.8</td>
<td>173.000000</td>
</tr>
<tr>
<td>mbp_30,60</td>
<td>0.538</td>
<td>6.373</td>
<td>4.214</td>
<td>23549.3</td>
<td>30.9</td>
<td>51.000000</td>
</tr>
<tr>
<td>mbp_50,100</td>
<td>1.147</td>
<td>7.143</td>
<td>4.124</td>
<td>31936.7</td>
<td>18.4</td>
<td>84.000000</td>
</tr>
<tr>
<td>mbp_50,1000</td>
<td>2.169</td>
<td>4.187</td>
<td>2.140</td>
<td>37727.3</td>
<td>19.2</td>
<td>504.000000</td>
</tr>
<tr>
<td>mbp_50,200</td>
<td>1.385</td>
<td>5.643</td>
<td>3.268</td>
<td>36507.5</td>
<td>18.5</td>
<td>140.000000</td>
</tr>
<tr>
<td>mbp_50,400</td>
<td>1.535</td>
<td>4.104</td>
<td>2.622</td>
<td>35736.3</td>
<td>18.8</td>
<td>240.000000</td>
</tr>
</tbody>
</table>

(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


---

*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*