INTERNATIONAL JOURNAL OF OPTIMIZATION IN CIVIL ENGINEERING
Int. J. Optim. Civil Eng., 2014; 4(3):321-339

COMPUTER CODES FOR COLLIDING BODIES OPTIMIZATION AND ITS ENHANCED VERSION

A. Kaveh[*,†] and M. Ilchi Ghazaan
Centre of Excellence for Fundamental Studies in Structural Engineering, Iran University of Science and Technology, Narmak, Tehran, P.O. Box 16846-13114, Iran

ABSTRACT

Colliding bodies optimization (CBO) is a new population-based stochastic optimization algorithm based on the governing laws of one dimensional collision between two bodies from the physics. Each agent is modeled as a body with a specified mass and velocity. A collision occurs between pairs of objects to find the global or near-global solutions. Enhanced colliding bodies optimization (ECBO) uses memory to save some best solutions and utilizes a mechanism to escape from local optima. The performances of the CBO and ECBO are shown through truss and frame design optimization problems. The codes of these methods are presented in MATLAB and C++.

Received: 10 May 2014; Accepted: 15 September 2014

KEY WORDS: colliding bodies optimization; enhanced colliding bodies optimization; structural optimization; MATLAB; C++

1. INTRODUCTION

Meta-heuristics are the recent generation of the optimization methods that are proposed to solve complex problems. The basic idea behind these stochastic search techniques is usually to simulate the natural phenomena. Genetic algorithm (GA) is inspired by Darwin's theory about biological evolutions [1] and [2]. Particle swarm optimization (PSO) simulates the social interaction behavior of birds flocking and fish schooling [3] and [4]. Ant colony optimization (ACO) imitates the way that ant colonies find the shortest route between the

[*]Corresponding author: Department of Civil Engineering, Iran University of Science and Technology, Narmak, Tehran, Iran
[†]E-mail address: alikaveh@iust.ac.ir (A. Kaveh)

food and their nest [5]. Harmony search (HS) algorithm was conceptualized using the musical process of searching for a perfect state of harmony [6]. Charged system search (CSS) uses the electric laws of physics and the Newtonian laws of mechanics to guide the Charged Particles [7].

As a newly developed type of meta-heuristic algorithm, colliding bodies optimization (CBO) is introduced and applied to structural problems by Kaveh and Mahdavi [8-10]. The CBO is multi-agent algorithm inspired by a collision between two objects in one-dimension. Each agent is modeled as a body with a specified mass and velocity. A collision occurs between pairs of objects and the new positions of the colliding bodies are updated based on the collision laws. The enhanced colliding bodies optimization (ECBO) is introduced by the authors [11] and it uses memory to save some historically best solution to improve the CBO performance without increasing the computational cost. In this method, some components of agents are also changed to jump out from local minimum.

The remainder of this paper is organized as follows: The CBO and ECBO algorithms are briefly presented in Section 2. In order to show the performance of these techniques on structural optimization, section 3 includes truss and frame examples. The last section concludes the paper.

Computer codes in Matlab and C++ are provided in the Appendix 1 and Appendix 2, respectively.

2. OPTIMIZATION ALGORITHMS

2.1 Colliding bodies optimization (CBO)

Colliding bodies optimization (CBO) is a new meta-heuristic search algorithm that is developed by Kaveh and Mahdavi [8]. In this technique, one object collides with other object and they move towards a minimum energy level. The CBO is simple in concept and does not depend on any internal parameter. Each colliding body (CB), X_i, has a specified mass defined as:

$$m_k = \frac{\dfrac{1}{fit(k)}}{\dfrac{1}{\sum_{i=1}^{n} \dfrac{1}{fit(i)}}}, \qquad k = 1,2,...,n \tag{1}$$

where *fit*(*i*) represents the objective function value of the *i*th CB and *n* is the number of colliding bodies.

In order to select pairs of objects for collision, CBs are sorted according to their mass in a decreasing order and they are divided into two equal groups: (i) stationary group, (ii) moving group (Fig. 1). Moving objects collide to stationary objects to improve their positions and push stationary objects towards better positions. The velocities of the stationary and moving bodies before collision (v_i) are calculated by

$$v_i = 0, \qquad i = 1,2,...,\frac{n}{2} \tag{2}$$

$$v_i = x_{i-\frac{n}{2}} - x_i, \qquad i = \frac{n}{2}+1, \frac{n}{2}+2,...,n \tag{3}$$

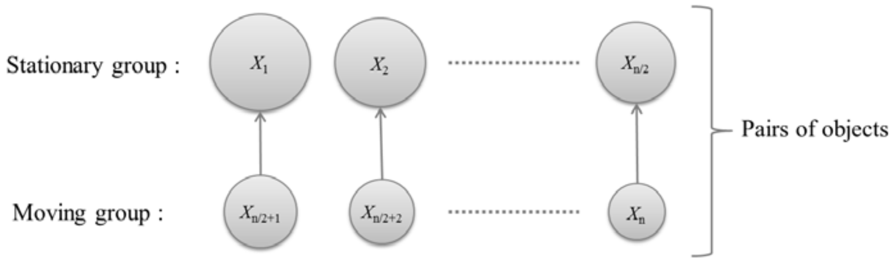

Figure 1. The pairs of CBs for collision [13]

The velocity of stationary and moving CBs after the collision (v'_i) are evaluated by

$$v_i^{'} = \frac{(m_{i+\frac{n}{2}} + \varepsilon m_{i+\frac{n}{2}})v_{i+\frac{n}{2}}}{m_i + m_{i+\frac{n}{2}}} \qquad i = 1,2,...,\frac{n}{2} \tag{4}$$

$$v_i^{'} = \frac{(m_i - \varepsilon m_{i-\frac{n}{2}})v_i}{m_i + m_{i-\frac{n}{2}}} \qquad i = \frac{n}{2}+1, \frac{n}{2}+2,...,n \tag{5}$$

$$\varepsilon = 1 - \frac{iter}{iter_{\max}} \tag{6}$$

where *iter* and $iter_{max}$ are the current iteration number and the total number of iteration for optimization process, respectively. ε is the coefficient of restitution (COR).

New positions of each group are updated by

$$x_i^{new} = x_i + rand \circ v_i^{'}, \qquad i = 1,2,...,\frac{n}{2} \tag{7}$$

$$x_i^{new} = x_{i-\frac{n}{2}} + rand \circ v_i^{'}, \qquad i = \frac{n}{2}+1, \frac{n}{2}+2,...,n \tag{8}$$

where x_i^{new}, x_i and v'_i are the new position, previous position and the velocity after the collision of the ith CB, respectively. rand is a random vector uniformly distributed in the range of [–1,1] and the sign ‘‘$\circ$’’ denotes an element-by-element multiplication.

The flowchart of CBO algorithm is depicted in Fig. 2. MATLAB and C++ codes for CBO are presented in Appendices 1 and 2.

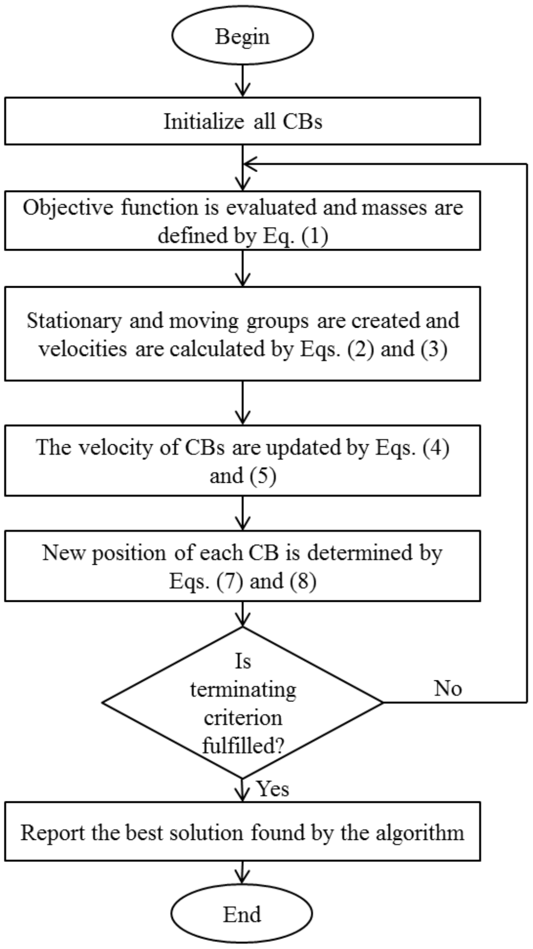
Begin
Initialize all CBs
Objective function is evaluated and masses are defined by Eq. (1)
Stationary and moving groups are created and velocities are calculated by Eqs. (2) and (3)
The velocity of CBs are updated by Eqs. (4) and (5)
New position of each CB is determined by Eqs. (7) and (8)
Is terminating criterion fulfilled?
No
Yes
Report the best solution found by the algorithm
End

Figure 2. Flowchart of the CBO algorithm

2.2 Enhanced colliding bodies optimization (ECBO)

In order to improve CBO to get faster and more reliable solutions, Enhanced Colliding Bodies Optimization (ECBO) was developed which uses memory to save a number of historically best CBs and also utilizes a mechanism to escape from local optima [11]. The flowchart of ECBO is shown in Fig. 3 and its codes in MATLAB and C++ are presented in Appendix 1 and 2. The steps of this technique are given as follows:

Level 1: Initialization

Step 1: The initial positions of all CBs are determined randomly in an m-dimensional search space.

$$x_i^0 = x_{\min} + random \circ (x_{\max} - x_{\min}), \qquad i = 1,2,...,n \tag{9}$$

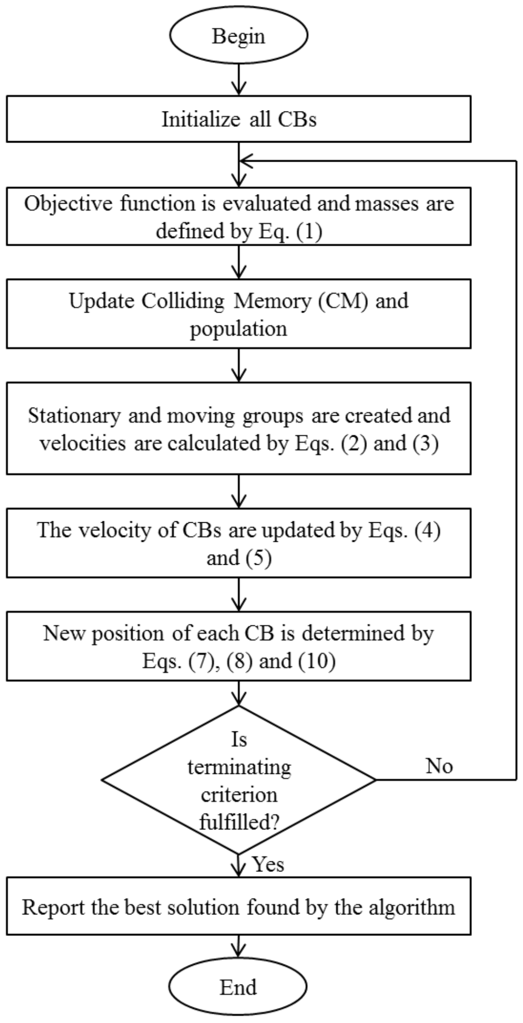
Begin
Initialize all CBs
Objective function is evaluated and masses are defined by Eq. (1)
Update Colliding Memory (CM) and population
Stationary and moving groups are created and velocities are calculated by Eqs. (2) and (3)
The velocity of CBs are updated by Eqs. (4) and (5)
New position of each CB is determined by Eqs. (7), (8) and (10)
Is terminating criterion fulfilled?
No
Yes
Report the best solution found by the algorithm
End

Figure 3. Flowchart of the ECBO algorithm [11]

where x_i^0 is the initial solution vector of the ith CB. Here, x_{min} and x_{max} are the bounds of design variables; random is a random vector which each component is in the interval [0, 1].

Level 2: Search

Step 1: The value of mass for each CB is evaluated according to Eq. (1).

Step 2: Colliding memory (CM) is utilized to save a number of historically best CB vectors and their related mass and objective function values. Solution vectors which are saved in CM are added to the population and the same number of current worst CBs are deleted. Finally, CBs are sorted according to their masses in a decreasing order.

Step 3: CBs are divided into two equal groups: (i) stationary group, (ii) moving group (Fig. 1).

Step 4: The velocities of stationary and moving bodies before collision are evaluated by Eqs. (2) and (3), respectively.

Step 5: The velocities of stationary and moving bodies after the collision are evaluated using Eqs. (4) and (5), respectively.

Step 6: The new position of each CB is calculated by Eqs. (7) and (8).

Step 7: A parameter like **Pro** within (0, 1) is introduced and it is specified whether a component of each CB must be changed or not. For each colliding body **Pro** is compared with rn_i (i=1,2,…,n) which is a random number uniformly distributed within (0, 1). If rn_i < **pro**, one dimension of the ith CB is selected randomly and its value is regenerated as follows:

$$x_{ij} = x_{j,\min} + random.(x_{j,\max} - x_{j,\min}) \tag{10}$$

where x_{ij} is the jth variable of the ith CB. $x_{j,min}$ and $x_{j,max}$ respectively, are the lower and upper bounds of the jth variable. In order to protect the structures of CBs, only one dimension is changed.

Level 3: Terminal condition check

Step 1: After the predefined maximum evaluation number, the optimization process is terminated.

3. NUMERICAL EXAMPLES

In this paper, the goal is to find optimum values for member cross-sectional areas that minimize the structural weight while satisfying some constraints. The minimum weight design problem can be formulated as:

$$\begin{array}{ll} \text{Find} & \{X\} = [x_1, x_2, .., x_{ng}] \\ \text{to minimize} & W(\{X\}) = \sum_{i=1}^{nm} \rho_i x_i L_i \\ \text{subjected to :} & \begin{cases} g_j(\{X\}) \le 0, \quad j = 1,2,...,n \\ x_{\text{i min}} \le x_{\text{i}} \le x_{i\,\text{max}} \end{cases} \end{array} \tag{11}$$

where $\{X\}$ is the vector containing the design variables; ng is the number of design variables; $W(\{X\})$ presents weight of the structure; nm is the number of elements of the structure; ρ_i and L_i denotes the material density and the length of the ith member, respectively. x_{imin} and x_{imax} are the lower and upper bounds of the design variable x_i, respectively. $g_j(\{X\})$ denotes design constraints; and n is the number of the constraints. The constraints are handled using the well-known penalty approach.

The performances of the standard CBO and ECBO are evaluated through two standard design optimization problems. The investigated instances consist of the 200-bar planar truss [12] and the 3-bay 15-story frame [13]. The population of 20 and 40 CBs are utilized for truss and frame problems, respectively. The predefined maximum evaluation number is considered as

20,000 analyses for two examples. To reduce statistical errors, each test is repeated 20 times.

3.1 A 200-bar planar truss

The 200-bar plane truss is shown in Fig. 4. The elastic modulus is 210 GPa and the material density is 7,860 kg/m^3 for all elements. Non-structural masses of 100 kg are attached to the nodes 1 to 5. The minimum admissible cross-sectional areas are 0.1 cm^2. Because of the symmetry, the bars are categorized into 29 groups. The first three natural frequencies of the structure are assumed as the constraints ($f_1 \geq 5$ Hz, $f_2 \geq 10$ Hz, $f_3 \geq 15$ Hz).

Table 1 illustrates the best solution vectors, the corresponding weights and mean weights of the CSS-BBBC [14], standard CBO and ECBO [12]. Table 2 represents the natural frequencies of the optimized structures. None of the frequency constraints are violated. The ECBO algorithm finds the best design among the other methods, which is 2158.08 kg. The best weights for CSS-BBBC and standard CBO are 2298.61 kg and 2161.15 kg, respectively.

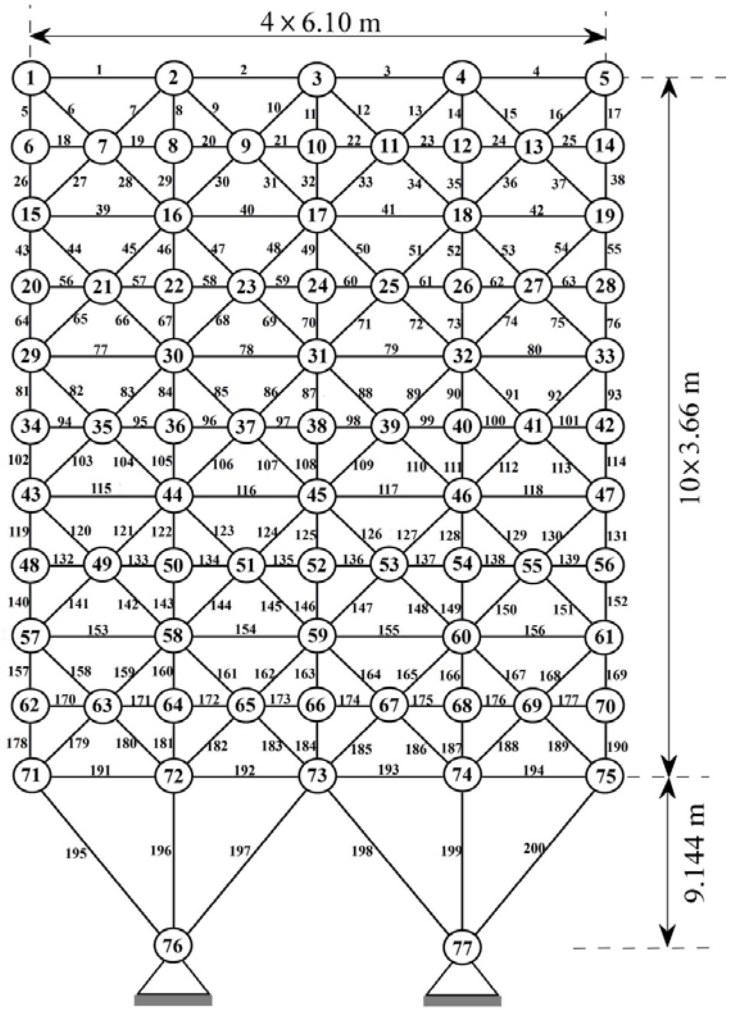

Figure 4. Schematic of the 200-bar planar truss [12]

Table 1: Optimal design obtained for the 200-bar planar truss

Element group	Members in the group	Areas (cm^2)		
		Kaveh and Zolghadr [14]	Present work	
			CBO	ECBO
1	1,2,3,4	0.2934	0.3059	0.2993
2	5,8,11,14,17	0.5561	0.4476	0.4497
3	19,20,21,22,23,24	0.2952	0.1000	0.1000
4	18,25,56,63,94,101,132,139,170,177	0.1970	0.1001	0.1
5	26,29,32,35,38	0.8340	0.4944	0.5137
6	6,7,9,10,12,13,15,16,27,28,30,31,33, 34,36,37	0.6455	0.8369	0.7914
7	39,40,41,42	0.1770	0.1001	0.1013
8	43,46,49,52,55	1.4796	1.5514	1.4129
9	57,58,59,60,61,62	0.4497	0.1000	0.1019
10	64,67,70,73,76	1.4556	1.5286	1.6460
11	44,45,47,48,50,51,53,54,65,66,68,69, 71,72,74,75	1.2238	1.1547	1.1532
12	77,78,79,80	0.2739	0.1000	0.1000
13	81,84,87,90,93	1.9174	2.9980	3.1850
14	95,96,97,98,99,100	0.1170	0.1017	0.1034
15	102,105,108,111,114	3.5535	3.2475	3.3126
16	82,83,85,86,88,89,91,92,103,104,106, 107,109,110,112,113	1.3360	1.5213	1.5920
17	115,116,117,118	0.6289	0.3996	0.2238
18	119,122,125,128,131	4.8335	4.7557	5.1227
19	133,134,135,136,137,138	0.6062	0.1002	0.1050
20	140,143,146,149,152	5.4393	5.1359	5.3707
21	120,121,123,124,126,127,129,130,141, 142,144,145,147,148,150,151	1.8435	2.1181	2.0645
22	153,154,155,156	0.8955	0.9200	0.5443
23	157,160,163,166,169	8.1759	7.3084	7.6497
24	171,172,173,174,175,176	0.3209	0.1185	0.1000
25	178,181,184,187,190	10.98	7.6901	7.6754
26	158,159,161,162,164,165,167,168,179, 180,182,183,185,186,188,189	2.9489	3.0895	2.7178
27	191,192,193,194	10.5243	10.6462	10.8141
28	195,197,198,200	20.4271	20.7190	21.6349
29	196,199	19.0983	11.7463	10.3520
Weight (kg)		2298.61	2161.15	2158.08
Mean weight (kg)		N/A	2447.52	2159.93

Table 2: Optimal design of the natural frequencies (Hz)

Frequency number	Natural frequencies (Hz)		
	Kaveh and Zolghadr [14]	Present work	
		CBO	ECBO
1	5.010	5.000	5.000
2	12.911	12.221	12.189
3	15.416	15.088	15.048
4	17.033	16.759	16.643
5	21.426	21.419	21.342

6	21.613	21.501	21.382

Fig. 5 depicts the best and average convergence history for the results of the standard CBO and ECBO. The standard CBO algorithm needs about 10,500 analyses to find the best solution while this number is about 14,700 analyses for the ECBO algorithm. It should be noted that the design found by ECBO at 10,500th analysis is lighter than that found by standard CBO at the same analysis.

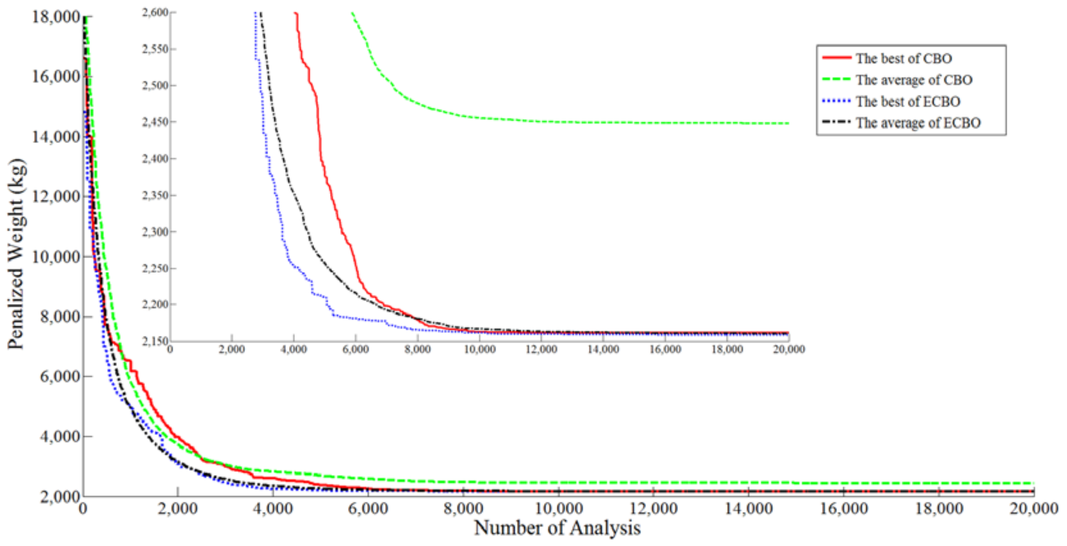

Figure 5. The convergence curve for the 200-bar planar truss [12]

3.2 A 3-bay 15-story frame

The configuration, applied loads and the numbering of member groups for this problem is shown in Fig. 6. The modulus of elasticity is 29,000 ksi (200 GPa) and the yield stress is 36 ksi (248.2 MPa) for all members. The effective length factors of the members are calculated as k_x≥0 for a sway-permitted frame and the out-of-plane effective length factor is specified as k_y=1.0. Each column is considered as non-braced along its length, and the non-braced length for each beam member is specified as one-fifth of the span length. The frame is designed following the LRFD specification and uses an inter-story drift displacement constraint [15]. Also, the sway of the top story is limited to 9.25 in (23.5 cm).

Table 3 shows the best solution vectors, the corresponding weights and the average weights for present algorithms and some other meta-heuristic algorithms [13]. ECBO has obtained the lightest design compared to other methods. The best weight of the ECBO algorithm is 86,986 lb while it is 95,850 lb for the HPSACO [16], 97,689 lb for the HBB-BC [17], 93,846 lb for the ICA [18], 92,723 lb for CSS [19] and 93,795 lb for the CBO. The CBO and ECBO algorithms get the optimal solution after 9,520 and 9,000 analyses, respectively. Convergence history of the present algorithms for the best and average optimum designs is depicted in Fig. 7. It can be seen that the convergence rate of the ECBO algorithm is higher than the CBO.

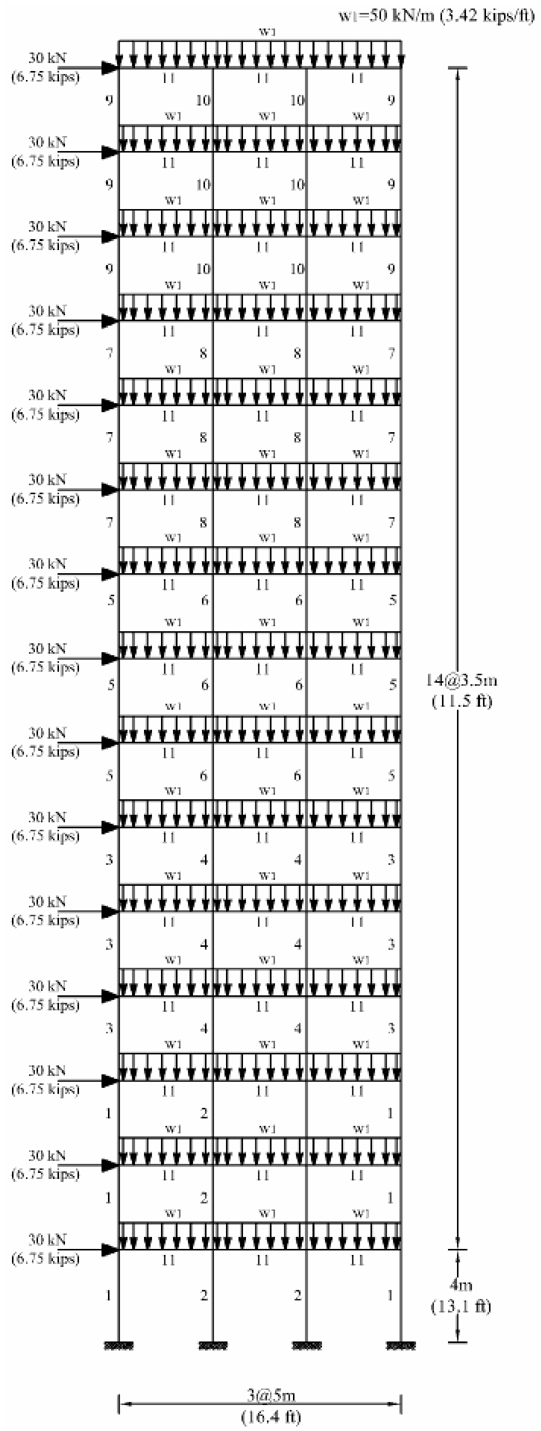

Figure 6. Schematic of the 3-bay 15-story frame [13]

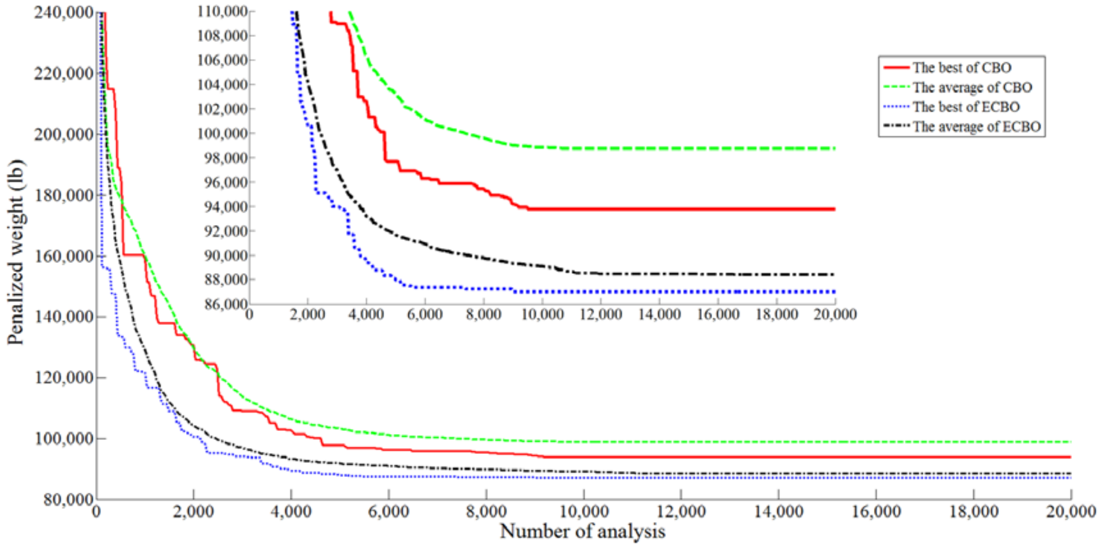

Figure 7. The convergence curve for the 3-bay 15-story frame [13]

Table 3: Optimal design obtained for the 3-bay 15-story frame

Element group	Optimal W-shaped sections					
	HPSACO [16]	HBB-BC [17]	ICA [18]	CSS [19]	Present work CBO	Present work ECBO
1	W21×111	W24×117	W24×117	W21×147	W24×104	W14×99
2	W18×158	W21×132	W21×147	W18×143	W40×167	W27×161
3	W10×88	W12×95	W27×84	W12×87	W27×84	W27×84
4	W30×116	W18×119	W27×114	W30×108	W27×114	W24×104
5	W21×83	W21×93	W14×74	W18×76	W21×68	W14×61
6	W24×103	W18×97	W18×86	W24×103	W30×90	W30×90
7	W21×55	W18×76	W12×96	W21×68	W8×48	W14×48
8	W27×114	W18×65	W24×68	W14×61	W21×68	W14×61
9	W10×33	W18×60	W10×39	W18×35	W14×34	W14×30
10	W18×46	W10×39	W12×40	W10×33	W8×35	W12×40
11	W21×44	W21×48	W21×44	W21×44	W21×50	W21×44
Weight (lb)	95,850	97,689	93,846	92,723	93,795	86,986
Mean weight (lb)	N/A	N/A	N/A	N/A	98,738	88,410

4. CONCLUSION

In the CBO, each solution vector is considered as a colliding body and the governing laws of collision from the physics is the base of this technique, where these laws determine the movement process of the CBs. The CBO has a simple formulation, and it requires no internal parameter tuning. Enhanced colliding bodies optimization (ECBO) uses memory to save a number of historically best CBs and also utilizes the random perturbation mechanism to update the positions. The introduction of memory can increase the convergence speed of ECBO with respect to CBO. Furthermore, changing some components of colliding bodies will help ECBO to escape from local minima.

REFERENCES

1. Holland JH. Adaptation in natural and artificial systems, Ann Arbor, *University of Michigan Press* 1975.
2. Goldberg DE. Genetic algorithms in search optimization and machine learning, Boston, *Addison-Wesley* 1989.
3. Eberhart RC, Kennedy J. A new optimizer using particle swarm theory, *In Proc. 6th Int. Symp. Micromachine Hum. Sci*, 1995, pp. 39-43.
4. Kennedy J, Eberhart RC. Particle swarm optimization, *In Proc. IEEE Int. Conf. Neural Netw*, 1995, pp. 1942-8.
5. Dorigo M, Maniezzo V, Colorni A. The ant system: optimization by a colony of cooperating agents, *IEEE Trans Syst Man Cybern* 1996; **B26**(1): 29-41.
6. Geem ZW, Kim J-H, Loganathan GV. A new heuristic optimization algorithm: harmony search, *Simulation* 2001; **76**(2): 60–8.
7. Kaveh A, Talatahari S. A novel heuristic optimization method: charged system search, *Acta Mech* 2010; **213**: 267-86.
8. Kaveh A, Mahdavai VR. Colliding bodies optimization: A novel meta-heuristic method, *Comput Struct* 2014; **139**: 18-27.
9. Kaveh A, Mahdavai VR. Colliding Bodies Optimization method for optimum design of truss structures with continuous variables, *Adv Eng Softw* 2014; **70**: 1-12.
10. Kaveh A, Mahdavai VR. Colliding Bodies Optimization method for optimum discrete design of truss structures, *Comput Struct* 2014; **139**: 43-53.
11. Kaveh A, Ilchi Ghazaan M. Enhanced colliding bodies optimization for design problems with continuous and discrete variables, *Adv Eng Softw* 2014; **77**: 66-75.
12. Kaveh A, Ilchi Ghazaan M. Enhanced colliding bodies algorithm for truss optimization with frequency constraints, accepted for publication in *J Comput Civil Eng, ASCE* 2014.
13. Kaveh A, Ilchi Ghazaan M. A comparative study of CBO and ECBO for optimal design of skeletal structures, Submitted for publication, 2014.
14. Kaveh A, Zolghadr A. Truss optimization with natural frequency constraints using a hybridized CSS–BBBC algorithm with trap recognition capability, *Comput Struct* 2012; **102–103**: 14–27.
15. American Institute of Steel Construction (AISC), Manual of steel construction: load and resistance factor design, Chicago, 2001.
16. Kaveh A, Talatahari S. Hybrid algorithm of harmony search, particle swarm and ant colony for structural design optimization, *Stud Comput Intel* 2009; **239**: 159–98.
17. Kaveh A, Talatahari S. A discrete Big Bang-Big Crunch algorithm for optimal design of skeletal structures, *Asian J. Civil Eng* 2010; **11**:103-22.
18. Kaveh A, Talatahari S. Optimum design of skeletal structure using imperialist competitive algorithm, *Comput Struct* 2010; **88**: 1220-29.
19. Kaveh A, Talatahari S. Charged system search for optimal design of planar frame structures, *Appl Soft Comput* 2012; **12**: 382–93.

APPENDIX 1: CBO AND ECBO IN MATLAB

The CBO code in MATLAB:

```
% Colliding Bodies Optimization - CBO

% clear memory
clear all

% Initializing variables
popSize=20;     % Size of the population
nVar=30;        % number of optimization variables
xMin=-500;      % lower bound of the variables
xMax=500;       % upper bound of the variables
maxIt=200;      % Maximum number of iteration

% Initializing Colliding Bodies (CB)
CB=xMin+rand(popSize,nVar).*(xMax-xMin); % random population


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start iteration
iter=0;                     % counter
Inf=1e100;                  % infinity
bestCost=Inf;               % initializing the best cost
agentCost=zeros(popSize,2); % array of agent costs

while iter < maxIt
    iter=iter+1;

    % Evaluating the population
    for e=1:popSize
        cost=eval(CB(e,:)); % evaluating objective function for each agent
        agentCost(e,1)=cost;
        agentCost(e,2)=e;
    end

    % Finding the best CB
    agentCost=sortrows(agentCost);
    if agentCost(1,1)<bestCost
        bestCost=agentCost(1,1);
        bestDesign=CB(agentCost(1,2),:); % the best design
    end

    % Evaluating the mass
    mass=zeros(popSize,1);
    for e=1:popSize
        mass(e,:)=1/(agentCost(e,1));
    end

    % Updating CB positions
    for e=1:popSize/2
        indexS=e;             % index of stationary bodies
        indexM=popSize/2+e;   % index of moving bodies
        COR=(1-(iter/maxIt)); % coefficient of restitution
        % velocity of moving bodies before collision
        velMb=(CB(agentCost(indexS,2),:)-CB(agentCost(indexM,2),:));
        % velocity of stationary bodies after collision
       velSa=((1+COR)*mass(indexM,1))/(mass(indexS,1)+mass(indexM,1))*velMb;
        % velocity of moving bodies after collision
         velMa=(mass(indexM,1)-COR*mass(indexS,1))/(mass(indexS,1)…
             +mass(indexM,1))*velMb;
        CB(agentCost(indexM,2),:)=CB(agentCost(indexS,2),:)…
                                 +2*(0.5-rand(1,nVar)).*velMa;
        CB(agentCost(indexS,2),:)=CB(agentCost(indexS,2),:)…
                                 +2*(0.5-rand(1,nVar)).*velSa;
    end

end% while

disp(bestCost)
disp(bestDesign)
```

The ECBO code in MATLAB:

```
% Enhanced Colliding Bodies Optimization - ECBO

% clear memory
clear all

% Initializing variables
popSize=20;     % Size of the population
nVar=30;        % number of optimization variables
cMs=2;          % Colliding memory size
pro=0.3;
xMin=-500;      % lower bound of the variables
xMax=500;       % upper bound of the variables
maxIt=200;      % Maximum number of iteration

% Initializing Colliding Bodies (CB)
CB=xMin+rand(popSize,nVar).*(xMax-xMin); % random population


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start iteration
iter=0;                     % counter
agentCost=zeros(popSize,2); % array of agent costs
Inf=1e100;                  % infinity
% Colliding memory; The first column contains CB costs and the remaining
columns include CB positions
cm=zeros(cMs,nVar+1);
tm=zeros(2*cMs,nVar+1);     % Temporary memory
for e=1:cMs
    cm(e,1)=Inf;
end

while iter < maxIt
    iter=iter+1;

    % Evaluating the population
    for e=1:popSize
        cost=eval(CB(e,:)); % evaluating objective function for each agent
        agentCost(e,1)=cost;
        agentCost(e,2)=e;
    end

    % Updating colliding memory
    agentCost=sortrows(agentCost);
    if iter>1
        for e=1:cMs
            agentCost(popSize-cMs+e,1)=cm(e,1);
            for ee=1:nVar
                CB(agentCost(popSize-cMs+e,2),ee)=cm(e,ee+1);
            end
        end
    end
    for e=1:cMs
        tm(e,1)=agentCost(e,1);
        tm(e+cMs,1)=cm(e,1);
        for ee=1:nVar
            tm(e,ee+1)=CB(agentCost(e,2),ee);
            tm(e+cMs,ee+1)=cm(e,ee+1);
        end
    end
    tm=sortrows(tm);
    for e=1:cMs
        cm(e,:)=tm(e,:);
    end
    agentCost=sortrows(agentCost);

    % Evaluating the mass
    mass=zeros(popSize,1);
    for e=1:popSize
        mass(e,:)=1/(agentCost(e,1));
    end

    % Updating CB positions
    for e=1:popSize/2
        indexS=e;       % index of stationary bodies
        indexM=popSize/2+e;   % index of moving bodies
        COR=(1-(iter/maxIt)); % coefficient of restitution
        % velocity of moving bodies before collision
        velMb=(CB(agentCost(indexS,2),:)-CB(agentCost(indexM,2),:));
        % velocity of stationary bodies after collision
       velSa=((1+COR)*mass(indexM,1))/(mass(indexS,1)+mass(indexM,1))*velMb;
        % velocity of moving bodies after collision
        velMa=(mass(indexM,1)-COR*mass(indexS,1))/(mass(indexS,1)…
              +mass(indexM,1))*velMb;
        CB(agentCost(indexM,2),:)=CB(agentCost(indexS,2),:)…
                                +2*(0.5-rand(1,nVar)).*velMa;
        CB(agentCost(indexS,2),:)=CB(agentCost(indexS,2),:)…
                               +2*(0.5-rand(1,nVar)).*velSa;
        if rand<pro
            tmp=ceil(rand*nVar);
            CB(agentCost(indexS,2),tmp)=xMin+rand*(xMax-xMin);
        end
        if rand<pro
            tmp=ceil(rand*nVar);
            CB(agentCost(indexM,2),tmp)=xMin+rand*(xMax-xMin);
        end
    end

end% while

disp(cm(1,:))
```

APPENDIX 2: CBO AND ECBO IN C++

The CBO code in C++:

```
#include "util.h"

long double eval (matrix CB) {
  //...
}

class CBO{
  private:
    #define POPSIZE 20
    #define NVAR 30
    #define XMIN -32
    #define XMAX 32
    #define MAXIT 10000
    #define inf 1e100
```

```
      matrix CB;

    public:
      CBO (){
        CB = matrix(POPSIZE, NVAR);
        CB.fill_rand(XMIN, XMAX);
      }
      long double run(){
        long double best_cost = inf;
        matrix best_design;
        matrix fit1(POPSIZE, 2);
        for (int it=0; it<MAXIT; it++){
          //evaluating the population
          for (int e=0; e<POPSIZE; e++){
          //evaluating objective function for each agent
            long double cost = eval(CB.getrow(e));
            //long double cost;
            fit1.a[e][0] = cost;
            fit1.a[e][1] = e;
          }
          //finding the best CB
          fit1.sort(0, fit1.get_n());
          if (fit1.a[0][0] < best_cost){
            best_cost = fit1.a[0][0];
            best_design = CB.getrow((int)fit1.a[0][1]); //the best design
          }
          //evaluating the mass
          matrix mass(POPSIZE, 1);
          for (int e=0; e<POPSIZE; e++)
            mass.a[e][0] = 1.0/fit1.a[e][0];
          //updating CB positions
          for (int e=0; e<POPSIZE/2; e++){
            int index_s = e;          //index of stationary bodies
            int index_m = POPSIZE/2 + e;   //index of moving bodies
          //coefficient of restitution
            long double cor = 1.0 - (long double)it / MAXIT;
          // velocity of moving bodies before colllision
            matrix vel_mb = CB.getrow(fit1.a[index_s][1]) …
                          - CB.getrow(fit1.a[index_m][1]);
          // velocity of stationary bodies after colllision
            matrix vel_sa = vel_mb * (((1+cor) * mass.a[index_m][0]) …
                          / (mass.a[index_s][0] + mass.a[index_m][0]));
          // velocity of moving bodies after colllision
            matrix vel_ma = vel_mb * ((mass.a[index_m][0]- …
                          cor*mass.a[index_s][0])/(mass.a[index_s][0] …
                          +mass.a[index_m][0]));
            matrix rand1 = matrix(1,NVAR); rand1.fill_rand(-0.5,0.5);
            matrix rand2 = matrix(1,NVAR); rand2.fill_rand(-0.5,0.5);
            CB.a[fit1.a[index_m][1]] = (CB.getrow(fit1.a[index_s][1]) …
                                    + ((rand1 * 2.0) ^ vel_ma)).a[0];
            CB.a[fit1.a[index_s][1]] = (CB.getrow(fit1.a[index_s][1]) …
                                    + ((rand2 * 2.0) ^ vel_sa)).a[0];
          }
        }
        return best_cost;
      }
  };
```

The ECBO code in C++:

```
  #include "util.h"

  long double eval (matrix CB) {
    //...
  }
```

```
class ECBO{
  private:
    #define POPSIZE 20
    #define NVAR 30
    #define CMS 2
    #define PRO 0.25
    #define XMIN -32
    #define XMAX 32
    #define MAXIT 10000
    #define inf 1e100

    matrix CB;

  public:
    ECBO (){
      CB = matrix(POPSIZE, NVAR);
      CB.fill_rand(XMIN, XMAX);
    }
    long double run(){
      long double best_cost = inf;
      matrix best_design;
      matrix agent_cost(POPSIZE, 2);
      matrix cm(CMS, NVAR+1);
      matrix tm(2*CMS, NVAR+1);
      for (int e=0; e<CMS; e++)
        cm.a[e][0] = inf;
      for (int it=0; it<MAXIT; it++){
        //evaluating the population
        for (int e=0; e<POPSIZE; e++){
        //evaluating objective function for each agent
          long double cost = eval(CB.getrow(e));
          agent_cost.a[e][0] = cost;
          agent_cost.a[e][1] = e;
        }
        //updating colliding memory
        agent_cost.sort(0, agent_cost.get_n());
        if (it > 1){
          for (int e=0; e<CMS; e++){
            agent_cost.a[POPSIZE-CMS+e][0] = cm.a[e][0];
            for (int ee=0; ee<NVAR; ee++)
              CB.a[agent_cost.a[POPSIZE-CMS+e][1]][ee] = cm.a[e][ee+1];
          }
        }
        for (int e=0; e<CMS; e++){
          tm.a[e][0] = agent_cost.a[e][0];
          tm.a[e+CMS][0] = cm.a[e][0];
          for (int ee=0; ee<NVAR; ee++){
            tm.a[e][ee+1] = CB.a[agent_cost.a[e][1]][ee];
            tm.a[e+CMS][ee+1] = cm.a[e][ee+1];
          }
        }
        tm.sort(0, tm.get_n());
        for (int e=0; e<CMS; e++)
          cm.a[e] = tm.a[e];
        agent_cost.sort(0, agent_cost.get_n());
        //evaluating the mass
        matrix mass(POPSIZE, 1);
        for (int e=0; e<POPSIZE; e++)
          mass.a[e][0] = 1.0/agent_cost.a[e][0];
        //updating CB positions
        for (int e=0; e<POPSIZE/2; e++){
          int index_s = e;          //index of stationary bodies
          int index_m = POPSIZE/2 + e;   //index of moving bodies
        //coefficient of restitution
          long double cor = 1.0 - (long double)it / MAXIT;
       // velocity of moving bodies before colllision
          matrix vel_mb = CB.getrow(agent_cost.a[index_s][1]) …
```

```
                        - CB.getrow(agent_cost.a[index_m][1]);
           % velocity of stationary bodies after collision
             matrix vel_sa = vel_mb * (((1+cor) * mass.a[index_m][0]) …
                          / (mass.a[index_s][0] + mass.a[index_m][0]));
           // velocity of moving bodies after colllision
             matrix vel_ma = vel_mb * ((mass.a[index_m][0]- …
                                  cor*mass.a[index_s][0])/(mass.a[index_s][0]
                                  +mass.a[index_m][0]));
             matrix rand1 = matrix(1,NVAR); rand1.fill_rand(-0.5,0.5);
             matrix rand2 = matrix(1,NVAR); rand2.fill_rand(-0.5,0.5);
             CB.a[agent_cost.a[index_m][1]] = …
                                (CB.getrow(agent_cost.a[index_s][1]) …
                                 + ((rand1 * 2.0) ^ vel_ma)).a[0];
             CB.a[agent_cost.a[index_s][1]] = …
                                       (CB.getrow(agent_cost.a[index_s][1]) …
                                         + ((rand2 * 2.0) ^ vel_sa)).a[0];
             assert (agent_cost.a[12].size() == 2);
             if (next_random(0.0,1.0) < PRO){
               int tmp = ceil(next_random(1e-10, 1.0) * NVAR) - 1;
               CB.a[agent_cost.a[index_s][1]][tmp] = next_random(XMIN, XMAX);
             }
             assert (agent_cost.a[12].size() == 2);
             if (next_random(0.0,1.0) < PRO){
               int tmp = ceil(next_random(1e-10, 0.1) * NVAR) - 1;
               CB.a[agent_cost.a[index_s][1]][tmp] = next_random(XMIN, XMAX);
             }
           }
         }
         return cm.a[0][0];
       }
   };
```

```
#include <bits/stdc++.h>
using namespace std;

long double next_random (long double lo, long double hi){
  #define MAXRANDOM 16000
  int r = rand() % MAXRANDOM;
  return lo + (r / ((long double)MAXRANDOM-1)) * (hi - lo);
}

class matrix{
  public:
    vector < vector<long double> > a;
    matrix () {}
    matrix (int n, int m){
      a = vector < vector<long double> > (n, vector<long double>(m, 0.0));
    }
    int get_n () { return a.size(); }
    int get_m () { return a[0].size(); }

    void fill_rand(long double lo, long double hi){
      for (int i=0; i<a.size(); i++)
        for (int j=0; j<a[i].size(); j++)
          a[i][j] = next_random(lo,hi);
    }
    matrix operator + (const long double &val) const{
      matrix ret = *this;
      for (int i=0; i<ret.a.size(); i++)
        for (int j=0; j<ret.a[i].size(); j++)
          ret.a[i][j]+= val;
      return ret;
    }
    matrix operator + (const matrix &operand) const{
      matrix ret = *this;
      for (int i=0; i<operand.a.size(); i++)
        for (int j=0; j<operand.a[i].size(); j++)
          ret.a[i][j]+= operand.a[i][j];
      return ret;
    }
    matrix operator - (const long double &val) const{
      return (*this) + (-val);
    }
    matrix operator - (const matrix &operand) const{
      matrix ret = *this;
      for (int i=0; i<operand.a.size(); i++)
        for (int j=0; j<operand.a[i].size(); j++)
          ret.a[i][j]-= operand.a[i][j];
      return ret;
    }
    matrix operator * (const long double &coeff) const{
      matrix ret = *this;
      for (int i=0; i<ret.a.size(); i++)
        for (int j=0; j<ret.a[i].size(); j++)
          ret.a[i][j]*= coeff;
      return ret;
    }
    matrix operator * (const matrix &operand) const{
      matrix ret(a.size(), a[0].size());
      for (int i=0; i<ret.a.size(); i++)
        for (int j=0; j<ret.a[i].size(); j++)
          for (int k=0; k<a[i].size(); k++)
            ret.a[i][j] = (ret.a[i][j] + a[i][k] * operand.a[k][j]);
      return ret;
    }
    matrix operator ^ (const matrix &operand) const{
      matrix ret = *this;
      for (int i=0; i<ret.a.size(); i++)
        for (int j=0; j<ret.a[i].size(); j++)
          ret.a[i][j]*= operand.a[i][j];
      return ret;
    }
    void sort (int i, int j){
      std::sort(a.begin()+i, a.begin()+j);
    }
    matrix getrow (int r){
      matrix ret(1, a[0].size());
      ret.a[0] = a[r];
      return ret;
    }
};
```