Solving Magic Cube

LI YICHENG s1684930

XIA ZHIYUN s1731424

LIACS

l.y.c.liyicheng@gmail.com xia4zhi3yun@gmail.com

Abstract

First step is to use annealing simulation to solve magic cube/square problem. In every temperature, several different mutation modes were conducted. The parameters which affects the convergence velocity can be the amount of mutation in every temperature, the temperature change function, the probablity to change among different mutation modes and the method to decide what element in the candidate vector should be changed. The next step is to use genetic algorithm to solve this problem, however in the solution in this paper, the geno type representation is the same with the pheno type representation and the crossover is not actually happening between two parents but it is kind of self-crossover. The reason for doing this is that I do not want to waste much computation in geno repair which is caused by duplicated integers. The $\mu + \lambda$ *strategy is used in the ga algorithm that for the next generation we choose the best candidates both from the group formed by the parents and the children.And the breif result is, when using GA to solve both magic square, with in 10000 evaluate budget we can always have the oppotunity to find fitness below 1 and when it comes to the magic cube, we have the oppotunity to get fitness under 1000 within 10000 evaluate budget. This indicates GA's fast convergence to solve the problem. And using SA the result is always stable which can ensure a better solution to be find.*

I. General Algorithm

I.I. SA

```
set initial temperature T
generate initial solution s
f = evaluate(s)
fopt = f
while eval_count < eval_budget
  for 1 : k  /*k is the amount of trial
      per temperature*/
    generate a new solution s' /*according
        to mutation function*/
    f' = evaluate(s')
    if f' < f
      fopt = f /*update f*/
      s = s'   /*update s*/
    else
      p = exp((f - f')/T)
      if p > randon probablity
        s = s'   /*update s*/
      fi
    fi
  end for
  update T
end while
```

Important factors that influence the convergence velocity are first, how the temperature changes, second how to generate the new candidate and third, the amount of trials in each temperature. In the process of generating the new candidate, operation selection probablity is the key elements that influence the velocity.

I.II. GA

```
generate first generation with population
    mu
evaluate the parents
while not terminated
  for k = 1 to lambda( the amount of the
      children)
    select one parent pi according to
        their fitness
    if pc > rand()
      apply the self-crossover to pi then
          generate offspring_k
    fi
    if pm > rand()
      apply the  self-mutation to
          offspring_k
    fi
  end for
  evaluate the lambda offspring
```

```
    select the best mu candidates among both
        the offspring and the parents
    parents = mu best candidates
end while
```

The self-crossover operator and self-mutation operator are mentiioned below and both of them are used in sa and ga optimizers.

II. Mutation

II.I. Mutation Algorithm Used in SA

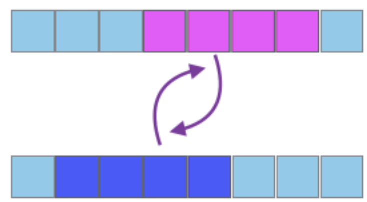
Algorithm for candidate mutation, take magic cube for example

```
reshape candidate into n^2 strings //can
    be n*n pillars or rows or colomns
evaluate each string
each string is assigned with a probability
select strings that will be mutated
if strings amount > 6
    select 6 indexes according to the
        indexes of the strings
    operation between rows
    operation between colomns
    operation between pillars
fi
```

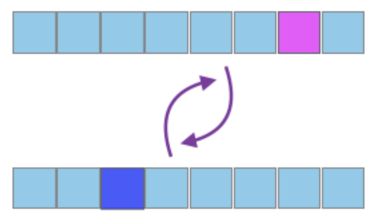
Unlike most of the candidate vector, the candidate vector for a magic square or a magic cube consists of integers without repeation. Therefore, instead of numerical mutation for every element in the vector, swap operation are mostly done to mutate the candidate vector.
Because of the special feature of the candidate, the first step is to reshape the candidate. For a magic square with order n, candidate vector has length $n*n$, and the candidate is reshaped into n strings with length n. For a magic cube with order n, candidate vector has length $n*n*n$, and there are three directions to reshape the magic cube candidate. We can view the candidate vector as $n*n$ pillars with length n or $n*n$ rows or $n*n$ colomns. *In the following report, the string means one of the colomn or row or pillar and the mutation is operated among a string or between two strings.*

II.II. Mutation Modes

Generally two kind of mutations are created, one is point to point swap and the other is string to string mutation, which are demonstrated in the following figures. In a swap operation between a string and another string, first the length of the string is randomly generated and then the position where the two substring would be extracted from the two strings are also generated randomly. And at last the two substrings are swapped and two new strings are created. In a swap mmutation



(a) *string to string swap*



(b) *point to point swap*

Figure 1: *two mutation demonstration*

The string to string swap is used in GA as the self-crossover operator and the point to point swap is used in GA as the self-mutation operator. Therefore, unlike the mutation algorithm used in SA, there are not very complicated mutation algorithms in GA so the following are all about the mutation algorihtm in SA.

II.III. Choosing Strings

Instead of generating random numbers to choose two strings in the cube or in the square, the first step in the algorithm is to evaluate all the strings. In my algorithm, because the magic cube is isotropic and to decrease the complexity of the operation, I only view the magic cube into *nXn* rows and evaluate each row by comparing them with the standard sum. Then, we generate a table to rate these strings, if the sum of the string is far diffferent from the standard sum, then this string has a high probability to be chosen. This can be clearly shown in the following piece of code.

```
P_row=ones(1,n^2);
for i = 1:n
    for j=1:n
        if abs(sum(s(j,:,i)))==std
            P_row(j+(i-1)*n)=0.125;
        else if abs(sum(s(j,:,i))-std)<
            std*0.001
            P_row(j+(i-1)*n)=0.5;
        else if abs(sum(s(j,:,i))-std) <
            std*0.005
            P_row(j+(i-1)*n)=0.7;
        else if abs(sum(s(j,:,i))-std) <
            std*0.01
            P_row(j+(i-1)*n)=0.85;
        else if abs(sum(s(j,:,i))-std) <
            std*0.015
            P_row(j+(i-1)*n)=0.95;
        fi fi fi fi fi
    end for
end for
for i=1:n^2
    if P_row(i) > rand(1)
        index(k+1) = i;
        k= k+1;
    fi
end for
```

Next step is pick 4 string indexes randomly, we first decode this index into (row,col,pillar), for instance, take 9 order magic cube for example, I view the magic cube as 81 rows, and if I got a string indexed by 56, then 56 is 6*9 + 2, which means the string is located at the 7th layer and is the second string of that layer. Now, use symbol to summarise this part into code. We have an index i, then we generate two other numbers, first is ceil(i/n) and the second is mod(i,n). Now we use these two numbers to choose other strings to mutate.
The whole process is illustrated in the following graph. (Take order 4 for instance)

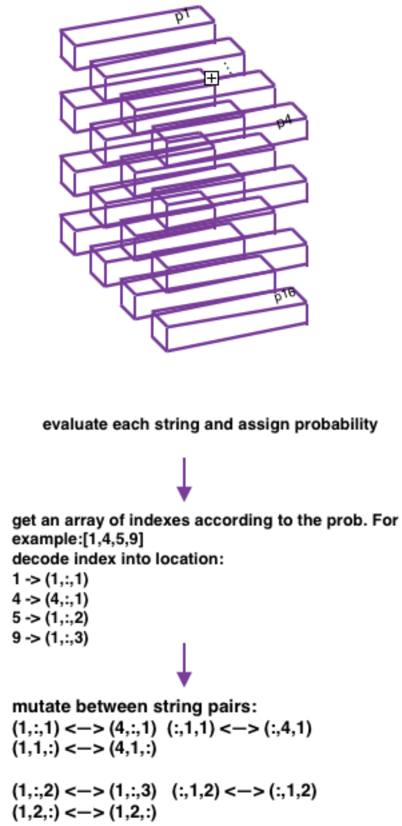


Figure 2: *Choosing Strings Algorithm Example*

III. Parameters in SA

III.I. Evaluation Times Per Temperature

To avoid random I did three experiments every time and from the result we can conclude that the convergence velocity is getting better and better with the decrease of k.

k	fopt		
	exp1	exp2	exp3
125	248.2	181.2	259.1
100	160.0	187.9	202.2
90	132.5	208.1	180.0
80	146.2	175.9	121.7
70	92.7	147.1	126.6
60	102.5	127.1	118.7
50	120.6	93.3	36.3
40	106.8	89.5	103.2
30	71.3	90.8	61.0
20	44.3	39.3	42.6
10	28.6	24.9	24.8
5	11.5	29.7	13.3

Table 1: *How evaluation times per temperature affects the velocity. (12 X 12 magic square, evaluation budget: 30000)*

III.II. Selection Between Two Mutation Methods

A variable pm is created to determine which kind of operation mutation would be selected every time.

```
if pm > rand(1)
   point to point swap operation
else
   string to string swap operation
fi
```

pm is continually decreasing during iterations, and to find out the best operation selection method, I change the the function which influence the pm and calculate the total times of point to point operation and string to string operation seperately and get the following data.

group	p2p weight(%)	fopt	avg
group1	1.38	56497	60971
	1.20	69182	
	1.41	57236	
group2	3.99	75597	63457
	5.19	59455	
	5.22	55321	
group3	14.73	48366	49340
	14.34	48309	
	15.93	51345	
group4	25.34	41490	47630
	26.27	49563	
	24.36	51839	
group5	35.36	36224	43971
	33.86	48402	
	35.09	47287	
group6	43.58	29547	31205
	45.71	35343	
	44.54	28725	
group7	55.16	19605	26063
	53.81	35499	
	53.63	23085	
group8	64.85	24137	22998
	65.42	26794	
	65.24	18064	
group9	74.21	22535	20228
	74.90	19179	
	75.61	18971	
group10	84.64	12620	11807
	85.24	574	
	84.16	22227	
group11	94.33	16569	20667
	93.82	28230	
	93.43	17203	

Table 2: *How different focus on two kind of mutations respectively affects the cnovergency velocity. budget: 40000)*

III.III. Probablity Table

The parameters like 0.001, 0.005 etc. are also a factor that can influence the performance of the method, and consider the situation that this algorithm can be used in different situations, istead of using specific integer like 1 or 2, I take the standard sum into consideration. The reason is obvious, if n is 3 then the standard sum is 56, and difference like 5 or 6 can not be stand, however when n is 9 and the standard sum is 3285, the difference like 5 or 6 is perfect in this case. After several tests I finally chose the above probability table. Attention to that even if the sum is equal to std, the string index still would be chosen because we do not want local optimal.

difference (%)	probability
0	0.125
0.1	0.5
0.5	0.7
1	0.85
1.5	0.95
>1.5	1

Table 3: *Example of probability table*
left: difference to the std. sum
right: probability to be chosen

Few experiment were done to adjust the parameters in this part since this can be subjective and if we define this table to be too complex that would affect the speed of the total processing.

IV. Parameters in GA

IV.I. Population

The first group of parameters that can be tuned are the parents amount μ and the offspring amount λ. The following table demonstrated how this two parameters affect the convergency.

Exp No.	μ	λ	fopt(10^4)
1	5	15	5.3121
2		20	3.7240
3		25	3.4086
4	10	30	5.8163
5		40	4.3294
6		50	3.2191
7	15	45	6.3917
8		60	5.2165
9		75	4.3925

Table 4: *GA parameter μ λ tuning, test based on magic cube pc = 0.8 pm = 0.2*

From the table we can observe that the amount of parents does not have a significant impact on the convergency however more offspring may lead to better result.

IV.II. Pm and Pc

Exp No.	p_c	p_m	fopt(10^4)
1	0.8	0.2	3.4056
2		0.4	1.9977
3		0.6	1.7689
4	0.7	0.2	2.5285
5		0.4	1.6253
6		0.6	0.85744
7	0.6	0.2	2.0687
8		0.4	1.1712
9		0.6	0.57004

Table 5: *GA parameter $\mu = 5$ $\lambda = 20$ p_c p_m tuning, test based on magic cube, eval_budget = 10000*

Surprisingly, it seems this problem and my algorithm is more favorable to p_m and the higher p_m and the less p_c may help get better solution.

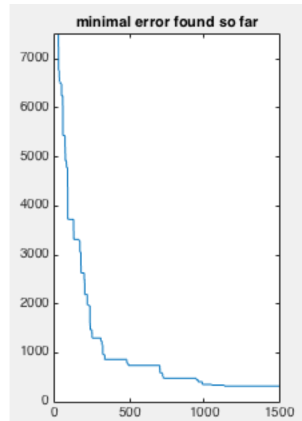
V. Result

According to the above discussion when using SA:
for solving magic square, we determine
k = 5 $P_{P2P} = 91\%$

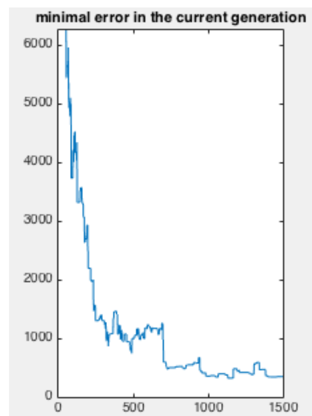
for solving magic cube, we determine
k = 15 $P_{P2P} = 83.2\%$
when using GA:
for solving both magic square and magic cube:
parents population μ: 5
offspring population λ: 25
p_c: 0.5
p_m: 0.8

V.I. Graph showing the process SA

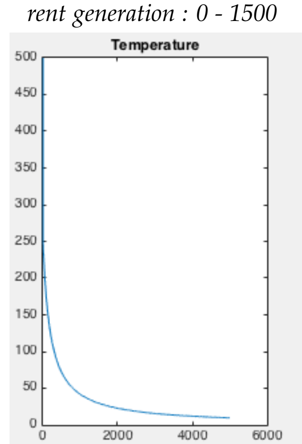
1. $MagicSquareeval_budget$: 5000



(a) *minimal error found so far : 0 - 1500*



(b) *minimal error found in the current generation : 0 - 1500*



(c) *temperature curve*

Figure 3: *solving magic square, evaluate budget 3000, graph showing process*

V.II. Data

The following is the result of several experiments respectively on magic square and magic cube.

Magic square:

Exp No.	fopt	avg
1	7.9231	
2	11.3077	
3	17.6538	
4	13.5769	
5	14.8769	
6	17.9231	16.6224
7	22.3077	
8	14.7308	
9	21.1923	
10	22.3546	
11	19.7692	

Table 6: *Magic square evaluate budget : 50000*

Consider the fopt is the MSE to the standard sum, thus a difference around 4 to the standard sum can be obtained within 50000 iterations.

Magic cube:

Exp No.	fopt 10^4	avg 10^4
1	3.0899	
2	3.5891	
3	3.4587	
4	3.0885	
5	3.3267	3.3115
6	3.5223	
7	3.3424	
8	3.0797	
9	3.3062	

Table 7: *Magic cube evaluate budget : 50000*

Standard sum for 9X9X9 cube is 3285, and within 50000 iteration we can get MSE around 33000 which means each row or coloum or pillar has a difference around 181. which account for 5.5% of the standard sum. And according to the probability table still all the string has the probability to be mutated. Because of the time consuming feature of my algorithm , I only did one greedy iteration (evaluate budget 150000), and got the fopt:825.2458 which means in this method it would not stuck and if giving enough iteration times it would find better solution and this can be seen from the below figure:

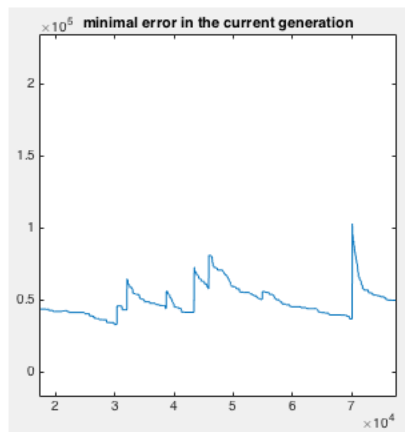


Figure 4: *large iteration times*

This is the advantage of simulating annealing.
From the result data in next section, when using GA to solve both magic square, with in 10000 evaluate budget we can always have the oppotunity to find fitness below 1 which indicates the fast convergence to solve the problem. And using SA the result is always stable which can ensure a better solution to be find.

VI. Results getting from 'run_comparison' script

Experiments Parameter setting: eval_budget = 15000 and runs_per_optimizer = 20;

2 optimizers detected: 'yicheng_li_sa' 'yicheng_li_ga'

Test problem 1/3 (square)

Optimizer 1/2 (yicheng_li_sa) on (square):

Executing run 1/20: fopt=32.115385, elapsed=4.390704
Executing run 2/20: fopt=18.923077, elapsed=4.640982
Executing run 3/20: fopt=49.615385, elapsed=4.559392
Executing run 4/20: fopt=32.423077, elapsed=4.421757
Executing run 5/20: fopt=31.576923, elapsed=4.645568
Executing run 6/20: fopt=45.384615, elapsed=4.499583
Executing run 7/20: fopt=26.461538, elapsed=4.537232
Executing run 8/20: fopt=41.423077, elapsed=4.451911
Executing run 9/20: fopt=40.576923, elapsed=4.430722
Executing run 10/20: fopt=38.423077, elapsed=4.583076
Executing run 11/20: fopt=52.500000, elapsed=4.567396
Executing run 12/20: fopt=53.230769, elapsed=4.424814
Executing run 13/20: fopt=53.500000, elapsed=4.440896
Executing run 14/20: fopt=29.615385, elapsed=4.540079
Executing run 15/20: fopt=40.884615, elapsed=4.671816
Executing run 16/20: fopt=22.423077, elapsed=4.676301
Executing run 17/20: fopt=45.076923, elapsed=4.411212
Executing run 18/20: fopt=70.923077, elapsed=4.443157
Executing run 19/20: fopt=35.269231, elapsed=4.427852
Executing run 20/20: fopt=39.461538, elapsed=4.489961
median: fopt=40.019231, elapsed=4.494772

Optimizer 2/2 (yicheng_li_ga) on (square):

Executing run 1/20: fopt=12.692308, elapsed=16.049244
Executing run 2/20: fopt=234.269231, elapsed=16.701510
Executing run 3/20: fopt=181.269231, elapsed=16.246582
Executing run 4/20: fopt=327.807692, elapsed=16.133161
Executing run 5/20: fopt=0.269231, elapsed=16.449347
Executing run 6/20: fopt=100.500000, elapsed=16.123489
Executing run 7/20: fopt=36.500000, elapsed=16.245289
Executing run 8/20: fopt=29.115385, elapsed=16.207332
Executing run 9/20: fopt=0.538462, elapsed=17.402994
Executing run 10/20: fopt=13.807692, elapsed=16.291342
Executing run 11/20: fopt=385.076923, elapsed=16.485633
Executing run 12/20: fopt=69.692308, elapsed=16.247654
Executing run 13/20: fopt=12.076923, elapsed=16.402910
Executing run 14/20: fopt=226.846154, elapsed=16.873349
Executing run 15/20: fopt=140.846154, elapsed=16.414276
Executing run 16/20: fopt=54.576923, elapsed=16.477947
Executing run 17/20: fopt=152.346154, elapsed=16.580532
Executing run 18/20: fopt=4.000000, elapsed=16.271088
Executing run 19/20: fopt=652.615385, elapsed=16.228104
Executing run 20/20: fopt=34.307692, elapsed=16.128122
median: fopt=62.134615, elapsed=16.281215

Test problem 2/3 (semi_cube)

Optimizer 1/2 (yicheng_li_sa) on (semi_cube):
Executing run 1/20: fopt=66398.259109, elapsed=43.669020
Executing run 2/20: fopt=59785.238866, elapsed=43.186991
Executing run 3/20: fopt=68513.360324, elapsed=43.596977
Executing run 4/20: fopt=72946.388664, elapsed=43.162552
Executing run 5/20: fopt=70111.983806, elapsed=43.116435
Executing run 6/20: fopt=62047.364372, elapsed=42.373546
Executing run 7/20: fopt=73191.858300, elapsed=43.866968
Executing run 8/20: fopt=67861.651822, elapsed=43.165127
Executing run 9/20: fopt=66051.364372, elapsed=42.396968
Executing run 10/20: fopt=71591.186235, elapsed=42.338696
Executing run 11/20: fopt=64363.631579, elapsed=43.380357
Executing run 12/20: fopt=69538.591093, elapsed=42.286594
Executing run 13/20: fopt=65990.906883, elapsed=41.426754
Executing run 14/20: fopt=62574.635628, elapsed=41.230004
Executing run 15/20: fopt=68493.364372, elapsed=40.988935
Executing run 16/20: fopt=67806.919028, elapsed=41.058615
Executing run 17/20: fopt=59260.558704, elapsed=40.889856
Executing run 18/20: fopt=65387.174089, elapsed=41.641407
Executing run 19/20: fopt=72470.105263, elapsed=42.335984
Executing run 20/20: fopt=65815.582996, elapsed=41.906787
median: fopt=67102.589069, elapsed=42.356121

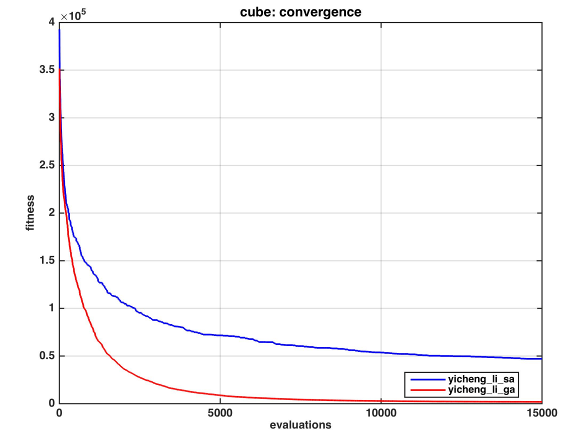
Optimizer 2/2 (yicheng_li_ga) on (semi_cube):
Executing run 1/20: fopt=22115.708502, elapsed=54.307458
Executing run 2/20: fopt=23212.246964, elapsed=51.076113
Executing run 3/20: fopt=27683.562753, elapsed=49.332312
Executing run 4/20: fopt=21942.024291, elapsed=49.920450
Executing run 5/20: fopt=23845.773279, elapsed=49.360149
Executing run 6/20: fopt=22411.842105, elapsed=49.875902
Executing run 7/20: fopt=21551.311741, elapsed=50.792367
Executing run 8/20: fopt=22117.659919, elapsed=49.289030
Executing run 9/20: fopt=22319.866397, elapsed=48.926406
Executing run 10/20: fopt=24002.020243, elapsed=49.550061
Executing run 11/20: fopt=21513.068826, elapsed=52.737895
Executing run 12/20: fopt=24175.502024, elapsed=49.216489
Executing run 13/20: fopt=22623.024291, elapsed=49.614862
Executing run 14/20: fopt=22157.854251, elapsed=49.272821
Executing run 15/20: fopt=22981.497976, elapsed=49.272224
Executing run 16/20: fopt=22513.311741, elapsed=49.208321
Executing run 17/20: fopt=21738.838057, elapsed=49.334266
Executing run 18/20: fopt=22727.157895, elapsed=49.437864
Executing run 19/20: fopt=23074.368421, elapsed=49.356353
Executing run 20/20: fopt=30362.198381, elapsed=49.224042
median: fopt=22568.168016, elapsed=49.358251

Test problem 3/3 (cube)
Optimizer 1/2 (yicheng_li_sa) on (cube):

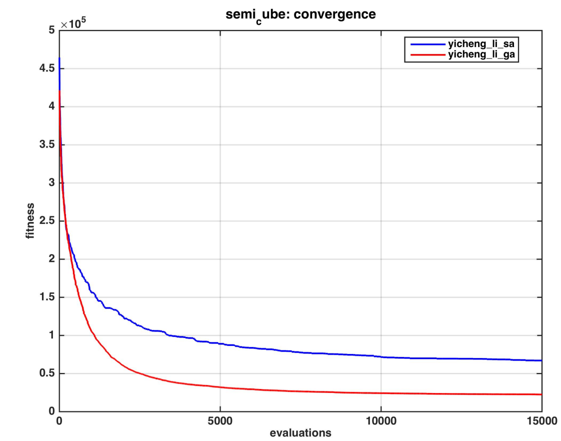
```
    Executing run 1/20: fopt=47740.691030, elapsed=67.312952
    Executing run 2/20: fopt=45958.906977, elapsed=69.278117
    Executing run 3/20: fopt=43300.242525, elapsed=66.919737
    Executing run 4/20: fopt=47333.458472, elapsed=67.561949
    Executing run 5/20: fopt=56117.245847, elapsed=65.719146
    Executing run 6/20: fopt=42931.803987, elapsed=65.201896
    Executing run 7/20: fopt=45262.980066, elapsed=66.234694
    Executing run 8/20: fopt=45558.295681, elapsed=65.602269
    Executing run 9/20: fopt=41948.607973, elapsed=65.478536
    Executing run 10/20: fopt=48787.116279, elapsed=65.563407
    Executing run 11/20: fopt=54215.093023, elapsed=65.608924
    Executing run 12/20: fopt=48521.093023, elapsed=65.120018
    Executing run 13/20: fopt=41486.255814, elapsed=65.936200
    Executing run 14/20: fopt=52916.601329, elapsed=69.422604
    Executing run 15/20: fopt=44117.132890, elapsed=67.285852
    Executing run 16/20: fopt=58695.940199, elapsed=66.524537
    Executing run 17/20: fopt=49490.514950, elapsed=66.488506
    Executing run 18/20: fopt=48344.740864, elapsed=65.883580
    Executing run 19/20: fopt=46789.641196, elapsed=67.744704
    Executing run 20/20: fopt=46812.448505, elapsed=74.032091
    median: fopt=47072.953488, elapsed=66.361600

Optimizer 2/2 (yicheng_li_ga) on (cube):
    Executing run 1/20: fopt=3901.146179, elapsed=221.841903
    Executing run 2/20: fopt=2078.438538, elapsed=214.497306
    Executing run 3/20: fopt=1901.249169, elapsed=215.799144
    Executing run 4/20: fopt=956.591362, elapsed=201.839940
    Executing run 5/20: fopt=3156.634551, elapsed=169.997891
    Executing run 6/20: fopt=2548.318937, elapsed=189.629769
    Executing run 7/20: fopt=1055.564784, elapsed=213.708614
    Executing run 8/20: fopt=847.299003, elapsed=211.438195
    Executing run 9/20: fopt=5960.883721, elapsed=213.241669
    Executing run 10/20: fopt=1530.823920, elapsed=214.076225
    Executing run 11/20: fopt=1875.591362, elapsed=225.177268
    Executing run 12/20: fopt=1556.647841, elapsed=216.731758
    Executing run 13/20: fopt=2604.727575, elapsed=169.905097
    Executing run 14/20: fopt=6309.740864, elapsed=188.613399
    Executing run 15/20: fopt=947.162791, elapsed=213.347190
    Executing run 16/20: fopt=2562.578073, elapsed=215.478943
    Executing run 17/20: fopt=1085.322259, elapsed=196.248160
    Executing run 18/20: fopt=3308.933555, elapsed=167.055251
    Executing run 19/20: fopt=1215.222591, elapsed=200.919613
    Executing run 20/20: fopt=1006.126246, elapsed=200.456605
    median: fopt=1888.420266, elapsed=212.339932
```

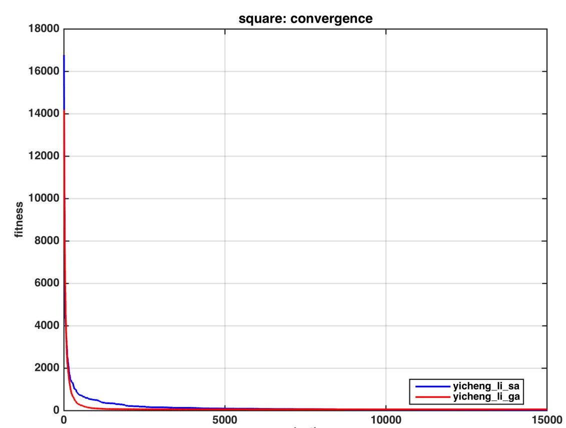
And we get the following figure showing the different algorithms' behaviors in different problems:



(a) *cube convergency*



(b) *semicube convergency*



(c) *square convergence*

Figure 5: *solving magic square, evaluate budget 3000, graph showing process*

From the data above and the figure below we can see that both SA and GA has a fast speed at the beginning but in the long run it could not behave better and may stopped at some local optimum however, GA can behave better in the long run and when using GA to solve both magic square, with in 10000 evaluate budget we can always have the oppotunity to find fitness below 1 and when it comes to the magic cube, we have the oppotunity to get fitness under 1000 within 10000 evaluate budget. This indicates GA's fast convergence to solve the problem. And using SA the result is always stable which can ensure a better solution to be find.