RESEARCH ARTICLE OPEN ACCESS

N-Queen Problem Revisited– A Variant of Cooperative Particle Swarm Optimization-based Solution

Fazlul Hasan Siddiqui [1], Md. Mostafijur Rahman [2], Md. Baker Hossen [3]
Md. Asrafullah [4]
Department of Computer Science and Engineering
Dhaka University of Engineering & Technology
Gazipur - Bangladesh

ABSTRACT
The n-queen problem is an interesting problem in computer science because of its exponential complexity and real world applications such as VLSI testing, traffic control, constraint satisfaction problem, load balancing in a multiprocessor computer, etc. A lot of researches have tried to solve n-queen, an NP-Hard problem, within reasonable time frame using various techniques. This problem is often treated as a combinatorial optimization problem. Particle swarm optimization (PSO), a stochastic optimization technique, can be useful to solve combinatorial optimization problems. Cooperative Particle Swarm Optimization (CPSO), an instance of Particle Swarm Optimization, has been used before to solve the n-queen problem for a larger dimension of n. In this paper, we develop a variant of CPSO technique, CPSO2, to solve the n-queen problem for a very large dimension of n with much more efficiency than the original PSO, CPSO, and other meta-heuristic methods such as genetic algorithm, simulated annealing, etc.
Keywords :— N-queen problem, particle swarm optimization, cooperative particle swarm optimization.

I. INTRODUCTION

The n-queen problem is about placing n chess queens on an n*n chess board so that no two queens attack each other. Two queens attack each other if they are in the same row or column or diagonal. A queen can be represented by Q_{ij}, where 'i' and 'j' represent the row and column numbers of the queen. Queens which have the same value of 'i' are in the same row, and which have the same value of 'j' are in same column. Here, 'i+j' represents the first diagonal number, and 'i-j' represents the second diagonal number. Queens which have the same value of 'i+j' are in the same first diagonal and which have the same value of 'i-j' are in the same second diagonal. An n-queen problem can also be represented as n-tuples (Q_1, Q_2, Q_3, …., Q_n), where Q_i represents the row position of the queen in the ith column [2].

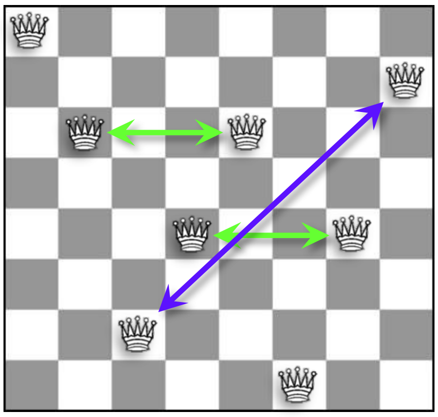
Fig. 1 Attacks between queens in an 8*8 chessboard.

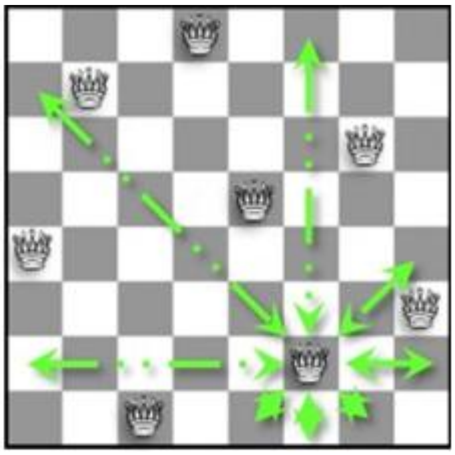
Fig. 2 A solution of 8 queen in an 8*8 chessboard.

The chess player Max Bezzel first introduced the 8-queen puzzle [3] in 1848 which was published in the German newspaper Berlin Schachzeitung. It was republished by Franz Nauck in 1850. From then this n-queen problem has been used in variety of real world applications, such as very large scale integration (VLSI) testing, traffic control, constraint satisfaction problems, permutation problems, computer science and industrial applications, load balancing in a multiprocessor computer, computer task scheduling, computer resource management, parallel memory storage scheme, optical parallel processing, and data compression [4]. These applications often require larger dimensions of n.

The value of n in an n-queen problem may vary from 1 to infinity. Search space of n-queen problem is exponential with the size of n and among that search space there are a few number of solutions [2]. For n=5 the search space is

120 and number of solution is only 10. For n=10 the search space become 3628800 and among this large search space the number of solution is only 724. Due to this exponential growth of search space solving n-queen problem for larger dimensions with brute-force technique within reasonable time frame is difficult, if not impossible. Backtracking is a fundamental strategy for solving this problem. In this strategy one starts with placing the first queen Q_1 in the first column, followed by placing the second queen Q_2 in a column so that it does not create any column or diagonal conflict with Q_1. This process incrementally continues placing the kth queen Q_k in some column so that it does not create any column or diagonal conflict with any of the queens $Q_1 \ldots Q_{k-1}$. If Q_k does not find any non-conflicting position then we backtrack by placing the previous queen in some other non-conflicting position, and repeat the procedure. Backtracking and other conventional algorithms such as greedy approach or dynamic programming fails to find a solution within reasonable time frame. For this reason, various heuristics or meta heuristic-based methods such as hill climbing, simulated annealing, genetic algorithm, tabu search [17], backtracking with local search, and different optimization techniques i.e. particle swarm optimization [18] and cooperative particle swarm optimization [1] are applied to solve n-queen problem.

For many real world applications, such as very large scale integration testing, traffic control, etc., finding any solution quickly is very important. In this paper, we develop a variant of cooperative particle swarm optimization, CPSO2, in order to find a solution of an n-queen problem efficiently.

The rest of this paper is organized as follows. We present an overview of particle swarm optimization in Section II. Section III gives an overview of cooperative particle swarm optimization. Details of our developed method (CPSO2) for solving n-queen problem is presented in Section IV. In Section V, we narrate performance analysis of our approach. Finally, Section VI describes the conclusion and future work.

II. PARTICLE SWARM OPTIMIZATION

Particle Swarm Optimization (PSO) [18] is a stochastic optimization technique that was developed by Eberhart and Kennedy (1995), motivated by the social behavior of bird flocking or fish schooling.

PSO works on a swarm that consists of a set of particles where particles are candidate solutions. Each particle searches for optimum value. In each iteration, each particle changes its position by a velocity and also updates the velocity. Velocity gives a random number and is generated by the equation (a). In every iteration, each particle updates its own best position achieved so far called *pbest* and also remembers the global best position of any particle achieved so far named *gbest*. After finding the two best values, the particle updates its velocity and position with the following equations (a) and *(b)*.

$$v = v * w + c1 * rand() * (pbest - present) + c2 * rand() * (gbest - present) \quad \ldots\ldots\ldots\ldots\ldots\ldots (a)$$

$$present = present + v \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (b)$$

Here, v is the particle velocity, *present* is the position of the current particle (solution). *pbest* and *gbest* are defined as stated before. The $rand$ function generates a random number between 0 and 1. *c1*, *c2* are learning factors. Usually *c1 = c2 = 2*. The velocity equation is updated by an inertia weight that is used for faster convergence.

Wang and Yang [5] proposed a different approach called Swarm Refinement PSO (SRPSO) to solve n-queen problem. SRPSO removes row and column conflict initially, and does not count any reversed attack for a collision. SRPSO performs better in both generations and execution time. Wang and Yang [5] experimentally shows that the swarm refinement technique is better than randomly produced particles.

III. COOPERATIVE PARTICLE SWARM OPTIMIZATION

Cooperative Particle Swarm Optimization (CPSO) was introduced by Van den Bergh and Engelbrecht [6]. In this method each n dimensional particle is divided into n 1-dimesional sub-particles. Each sub-particle is optimized aiming at optimizing whole particle. Positions of each sub-particle are updated according to the calculated velocity of each sub-particle. In the velocity equation personal best position of each sub-particle and global best position among all particles is used. Fitness of a sub-particle corresponds to the number of conflicts of the sub-particle with other sub-particles in a particle. Fitness of a particle is the summation of fitness values of all its constituent sub-particles.

Amooshahi et al. [1] solved n-queen problem by applying Cooperative Particle Swarm Optimization. They consider the best experience for each sub-particle as best local experience ($Pbesk_{spk}$) and best position for each particle among all particles as best global experience (Gbest). The velocity of each sub-particle is updated with the standard PSO equation, as shown below.

$$Vspk(t+1) = Vspk(t) * w(t) + c1 * rand(Pbestspk - Xspk(t)) + c2 * rand(Gbest - Xspk(t))$$

The location of Kth particle is updated according to the following equation.

$$Xspk(t+1) = Vspk(t+1) + Xspk(t)$$

This update is done whenever the fitness of evaluation function for that sub-particle would be optimized with the new velocity, according to the following equation.

$$Fit_{subparticlek} = \sum_{i=1}^{n} collisions_{ik}$$

By this method, the best local experience of each sub-particle is used to update context vector, without considering the overall position of that particle. Finding best global points according to overall fitness position of particles helps to keep the convergence of the algorithm.

IV. UPDATED CPSO (CPSO2) FOR SOLVING N QUEEN PROBLEM

Like CPSO our approach CPSO2 starts with an initial swarm which is a set of randomly generated particles. Each particle contains information of n-queens on an n*n chessboard. Then we divide each particle into n sub-particles. Each sub-particle includes information of single queen on an n*n chessboard. In each iteration, new positions of sub-particles are calculated by adding a velocity with the current positions of the sub-particles to reduce their conflicts with other sub-particles. The velocity of a sub-particle is a random integer value. We place a sub-particle in a new position if the new position is better than the current position for that sub-particle. The position of a sub-particle is evaluated by its fitness value, which corresponds to the number of conflicts of that sub-particle with other sub-particles in the corresponding particle. The sub-particle which has no conflicts with other sub-particles has zero fitness value. The fitness of a particle is the summation of fitness values of all the constituent sub-particles in that particle. A solution is found when conflicts of each sub-particle in a particle becomes zero.

In CPSO [1] an initial particle may have row, column, and diagonal conflicts but we eliminate all row and column conflicts during the generation of initial particles. This can simply be done by placing each of the queens Q_1, Q_2, Q_3, … Q_n in different rows, where Q_i represents the row position of the queen residing in the ith column. Therefore, in our approach an initial particle may contain only diagonal conflict, i.e., an initial particle does not contain any row or column conflict.

We calculate new positions only for sub-particles having conflicts with other sub-particles, meaning that we do not change the positions of sub-particles that have no conflict because they have already found their optimal positions. To calculate new position of a sub-particle we use an updated CPSO velocity equation. In our velocity equation, current position of a sub-particle is subtracted from the global best position found so far by all sub-particles named 'gBest', rather than the global best position of the corresponding particle achieved so far named 'Gbest'. For n-queen problem, position of a particle corresponds to the positions of n-queens in that particle, and position of a sub-particle corresponds to the position of a single queen. So, subtracting the position of a sub-particle from the position of a particle is not useful, for which we use gBest instead of Gbest.

To overcome from local minima, i.e. to escape from a stuck situation where any move of a queen increases the overall conflicts, if a sub-particle does not find better position within 'updateConstant' number of consecutive iteration then in the immediate next iteration we must update its position without considering whether the new position is better or not. The value of 'updateConstant' is equal to n if n<=1000, otherwise it is 1000. Such approach of deciding the value of 'updateConstant' is chosen from the experimental observation that there is a very high possibility of getting better position of a sub-particle within n number of iteration, but for larger value of n it takes more time. We use an update factor '$\text{updateFactor}_{spk}$' for each sub-particle, which represents the number of consecutive iterations the sub-particle does not find a better position. So, the initial value of update factor for a sub-particle is one and is incremented by one if the position of that sub-particle is not updated during an iteration. When position of a sub-particle is updated again its update factor is reset to one.

We also multiply the velocity of a sub-particle by its update factor to find long variations between velocities of a sub-particle in consecutive iterations while its position is not updated. This is because, long variations between velocities can give us a new position that is far from previously generated position and may be better than the current position. We use 10^7 as maximum velocity, named 'Vmax', of a sub-particle. Because very large value of a velocity creates computational complexity and takes more memory space. So if velocity of a sub-particle exceeds maximum velocity, then we define the velocity as follows.

Velocity = Velocity mod Vmax

The velocity equation of our approach is:

$$Vspk = (Vspk * W + C1 * rand1 * (pBest_{spk} - Xspk) + C2 * rand2 * (gBest - Xspk)) * updateFactor_{spk} \quad \text{……(c)}$$

Here, Vspk is the velocity of a sub-particle; 'W' is the inertia weight, C1 and C2 are constants; rand1 and rand2 are random numbers between 0 and 1, Xspk is the current position of the sub-particle, pBestspk is the best position of the sub-particle ever achieved, gBest is the global best position of sub-particles achieved so far, and updateFactorspk is the update factor of each subparticle.

```
1.  Randomly generate a set of initial particles
2.  Calculate gBest for each particle and $pBest_{spk}$ for
    each sub-particle
3.  for each particle p
4.  for each sub-particle spk
5.      If (fitness_Xspk > 0)
6.      {
7.              If ( fitness_Xspk < $fitness\_pBest_{spk}$)
8.                      $pBest_{spk} = X_{spk}$
9.              If (fitness_Xspk < fitness_gBest )
10.                     gBest = Xspk
11.             Vspk = ( Vspk * W + C1 * rand1 *
                  ($pBest_{spk}$ - Xspk ) + C2 * rand2 *
                  (gBest - Xspk ) ) * $updateFactor_{spk}$
12.             Vspk = Vspk mod n
13.             Xnew = (Vspk + Xspk) mod n
14.             If ( fitness_Xnew < fitness_Xspk OR

        $updateFactor_{spk}$ >updateConstant)
15.             {
16.                     Xspk = Xnew
17.                     fitness_Xspk = fitness_Xnew
18.             }
19.     }
20.     End for
21.   If (fitness_p = 0)
22.       return solution
23. End for
```

Fig. 3 Pseudo code of CPSO2

In the pseudocode shown in Figure 3,

- fitness_Xspk represents the fitness value of a sub-particle in the current position.
- fitness_Xnew represents the fitness value of a sub-particle in the new position.
- fitness_pBestspk represents the fitness value of a sub-particle in the local best position of that sub-particle achieved so far.
- fitness_gBest represents the fitness value of a sub-particle in the global best position of sub-particles achieved so far. fitness_p represents the fitness value of a particle p that is calculated as follows-

$$fitness\_p = \sum_{for\ all\ spk \in p} fitness_{spk}$$

In our pseudo code in Figure 3, line 1 generates initial particles. Line 2 calculates initial global and local best positions of sub-particles. Line 5 checks whether the sub-particle needs to get update. Lines 7-10 update local and global best position of the sub-particle. Lines 11-12 calculate the velocity of the sub-particle. Line 13 calculates the new position for the sub-particle. Lines 14-19 update the position and fitness value of the sub-particle. Line 21 evaluates the fitness of the particle and check whether it is zero, in which case the solution is found, otherwise the same process repeats in the next iteration.

The complexity of evaluating each sub-particle's fitness function is O(1), and that of each particle's fitness function is O(n), where n is the number of queens. Time complexity of our algorithm is O(n*n). The space complexity of our algorithm is O(36nM), where n is the number of queens and M is the swarm size.

V. Experimental analysis

In our experiment, we use Windows 7 and GNU GCC compiler in a CPU of Core i3 Intel 2.0 GHz. For each value of n we run our program for 10 times, and take the average of that as the final output.

Number of fitness function evaluation refers to the number of times it is checked for whether the solution is found or not. To analysis the performance of our algorithm we have compared the number of fitness function evaluation of our algorithm for different values of n against that of Simulated Annealing, Genetic Algorithm [2], PSO [8], and CPSO [1]. The result of this comparison is shown in Table 1.

TABLE 1
A COMPARISON AMONG OUR METHOD (CPSO2) AND OTHER META-HEURISTIC METHOD IN TERMS OF NUMBER OF FITNESS CALCULATION

n	SA	GA	PSO	CPSO	CPSO2
8	492.8	400	-	225.8	**196.5**
10	947.8	490	-	540.45	**297.3**
20	-	-	5669.7	2451.6	**871.4**
30	2159.9	9190	-	2020.5	**1887.7**
50	2848.6	1759230	14991.4	2764.2	**2621**
100	7872.7	887770	36199.4	5063.6	**4461.2**
200	2178.2	2287960	934399	9184.5	**7190.8**
300	2466.2	2774820	-	14559.6	**20673.1**
500	5669.7	8940640	-	23799.6	**18269.1**
1000	126401.7	-	-	47299.8	**34875.6**
2000	314373	-	-	95235.9	**79885.8**
3000	-	-	-	-	**144536**

4000	-	-	-	-	**280785**
5000	-	-	-	-	**320025**

Table-1 shows that number of fitness function evaluations of our method (CPSO2) for all values of n is lower than the other methods, meaning that CPSO2 is found more efficient than others.

We also compare the amount of time needed to find the first solution for different values of n between our algorithm CPSO2 and CPSO [1]. Here we measure the time in seconds. The result of this comparison is shown in Table 2. CPSO is quicker in finding the first solution than CPSO2 for smaller values of n, which is possibly because of the computational complexity in CPSO2 for multiplying the velocity of a sub-particle by its update factor in order to find long variations between velocities of a sub-particle in consecutive iterations while its position is not updated. However, finding that long variations helps CPSO2 to outperform CPSO for larger dimensions of chess board.

It is clear from Table 1 and Table 2 that our algorithm (CPSO2) gives much better result for larger values of n, mostly when n is equal or larger than 500.

TABLE 2
A COMPARISON BETWEEN CPSO2 AND CPSO [1] IN TERMS OF TIME (IN SECONDS) NEEDED TO FIND THE FIRST SOLUTION.

N	CPSO	CPSO2
8	**0.00047**	0.0109
10	**0.00133**	0.0203
20	**0.00765**	0.0968
50	**0.0734**	0.6194
100	**0.4331**	2.0421
200	**3.2190**	6.46
500	43.4925	**42.3183**
600	77.971	**56.9105**
700	122.9388	**75.56**
800	164.383	**138.422**
1000	400.234	**168.814**
2000	1993.876	**813.593**
3000	11520.856	**1664.08**

4000	-	**7184.5**
5000	-	**8993.75**

TABLE 3
A COMPARISON OF EXECUTION TIME (IN SECONDS) TO FIND THE FIRST SOLUTION FOR DIFFERENT NUMBER OF PARTICLES

N	NOP=2	NOP=3	NOP=5	NOP=10	NOP=15
10	**0.0203**	0.0224	0.026	0.0358	0.0376
20	**0.0968**	0.1333	0.176	0.1762	0.287
50	**0.6194**	0.8768	1.034	1.794	1.7893
100	**2.04**	2.92	3.099	5.1029	6.4189
200	**6.46**	9.88	11.87	21.25	27.150
500	**42.31**	52.11	75.08	126.73	132.49
700	**75.56**	76.85	116.5	156.8	198.5
1000	**168.81**	263.02	273.83	411.869	599.46

TABLE 4
A COMPARISON OF TOTAL NUMBER OF EVALUATIONS OF FITNESS FUNCTION TO FIND THE FIRST SOLUTION FOR DIFFERENT NUMBER OF PARTICLES.

N	NOP=2	NOP=3	NOP=5	NOP=10	NOP=15
10	**297.3**	376.6	428.5	430.7	483.5
20	**871.4**	1225.1	1650	1217.4	2559
50	**2621**	3751.3	4265.5	5784.4	6974.9
100	**4461.2**	6485.4	6187.7	8465.1	12980.2
200	**7190.8**	10843.7	12973	27237.3	28565.4
500	**18269.1**	22322.2	30527.3	53264.4	54816.1
700	**22322.5**	22685.7	35672.3	43871.2	49876.2
1000	**34875.6**	55498.5	56566.3	59690	120694

It is clear from Table 3 and Table 4 that using 2 particles gives better performance than using 3, 5, 10 or 15 particles, which suggests that using higher number of particles is not beneficial in general.

VI. CONCLUSIONS

The n-queen is a combinatorial optimization problem. In this paper, we have applied a variant of Cooperative Particle Swarm Optimization (CPSO2) to solve n-queen problem in a more efficient way. We change the velocity equation of CPSO by using global best position of a sub-particle instead of a particle. We also introduce some new concepts, such as, update factor and update constant. Applying those concepts shows a remarkable improvement in the result. We have found in our experiments that the number of fitness function evaluations to find a solution using our method is less than that of CPSO and other meta-heuristics-based methods for all values of n. The time required for finding the first solution using our method is much less than that of CPSO for n greater than or equal to 500, but for n less than 500 our method requires a little more time than CPSO, which suggest that our approach is much better for larger values of n. Number of particles in a swarm is also a big factor for finding solution. Our approach shows better result when there are only 2 particles in the swarm. Further study on the values of some parameters, such as, learning factor, inertia weight, and update constant could give better result in future.

REFERENCES

[1] Amooshahi, A., Joudaki, M., Imani, M. and Mazhari, N. (2012) 'Presenting a new method based on cooperative PSO to solve permutation problems: A case study of n- queen problem', 3rd International Conference on Electronics Computer Technology (ICECT), At India.

[2] Martinjak, I. and Golub M. 'Comparison of Heuristic Algorithms for the N-Queen Problem', in Proceedings of the ITI 2007 29th International Conference on Information Technology Interfaces, June 25-28, Cavtat, Croatia.

[3] Bell, J. and Stevens, B. (2009) 'A survey of known results and research areas for n-queens', Discrete Mathematics, vol. 309, pp. 1-31.

[4] Segundo, P.S. (2011) 'New Decision rules for exact search in N-queens', J. Global Optimization, pp. 497-514, DOI: 10.1007/s10898-011-9653-x.

[5] Wang, Y., Lin, H. and Yang, L. (2012) 'Swarm refinement PSO for solving N-queens problem', Third International Conference on Innovations in Bio-Inspired Computing and Application.

[6] Van den Bergh, F. and Engelbrecht, A.P. (2000) 'Cooperative Learning in Neural Networks using Particle Swarm Optimizers', South African Computer Journal, Vol.2000, Issue 26, pp. 84-90

[7] Dirakkhunakon, S. and Suansook, Y. (2009) 'Simulated Annealing with iterative improvement', International Conference on Signal Processing Systems, Singapore, pp. 302-306.

[8] Hu, X., Eberhart, R. C. and Shi, Y (2003) 'Swarm intelligence for permutation optimization: a case study of n-Queens problem', in Proceedings of the IEEE Swarm Intelligence Symposium (SIS '03), pp. 243-246, Indianapolis, Ind, USA.

[9] Homaifar, A., Turner, J. and Ali, S. (1992) 'The N-queens problem and genetic algorithms', in Proceedings of the IEEE Southeast Conference, pp. 261-262.

[10] Kennedy, J. and Eberhart, R. C. (1995) 'Particle Swarm Optimization', in Proceedings of the IEEE International Conference on Neural Networks, pp. 1942-1948.

[11] Kale, L. V. (1990) 'An Almost Perfect heuristic for the N non attacking queens problem', Information Processing Letters, pp. 173-178.

[12] Masehian, E., Akbaripour, H. and Nasrin, M. (2014) 'Solving the n-Queens Problem Using a Tuned Hybrid Imperialist Competitive Algorithm', The International Arab Journal of Information Technology, Vol. 11, No. 6.

[13] Sosic, R. and Gu, J. (1994) 'Efficient Local Search with Conflict Minimization: A Case Study of the n-Queens Problem', IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 5, pp. 661-668.

[14] Van den Bergh, F. and Engelbrecht, A.P. (2000) 'Cooperative Learning in Neural Networks using Particle Swarm Optimizers', South African Computer Journal, Vol. 2000, Issue 26, pp. 84-90.

[15] Van den Bergh, F. (2001) 'An Analysis of Particle Swarm Optimizers', PhD thesis, Department of Computer Science, University of Pretoria, Pretoria, South Africa.

[16] Van den Bergh, F. and Engelbrecht A.P. (2004) 'A Cooperative Approach to Particle Swarm Optimization', IEEE Transactions on Evolutionary Computation, Vol. 8, No. 3, pp. 225-239

[17] V Glover, F., & Laguna, M. (1998). Tabu search. In *Handbook of combinatorial optimization* (pp. 2093-2229). Springer, Boston, MA.

[18] Kennedy, J. (2011). Particle swarm optimization. In *Encyclopedia of machine learning* (pp. 760-766). Springer, Boston, MA.