Available online at http://scik.org

J. Math. Comput. Sci. 11 (2021), No. 2, 2058-2074

https://doi.org/10.28919/jmcs/5513

ISSN: 1927-5307

HYBRIDIZATION OF PARTICLE SWARM OPTIMIZATION AND SIMULATED ANNEALING FOR MAXIMUM PARTITION PROBLEM

WAEL MUSTAFA*

Department of Computer Science, An-Najah National University, 97300 Nablus, Palestine

Abstract: The maximum partition problem (MPP) is to divide the set of nodes in directed weighted graph into two disjoint subsets such that the sum of weights of edges crossing from one subset of the partition to the other is maximized. The MPP is NP-hard. Hence, this paper presents a hybrid discrete particle swarm optimization (DPSO) simulated annealing (SA) algorithm to solve MPP. The proposed algorithm first applies DPSO to the problem until improvement in fitness slows down (stagnates). Then, the algorithm uses SA augmented with a heuristic method to improve the fitness of the solution obtained from DPSO. Experiments on randomly generated graphs of different size show that the proposed algorithm produces better partitions than conventional DPSO. Additionally, the proposed algorithm converges to near optimal solutions faster than conventional DPSO.

Keywords: maximum partition problem; discrete particle swarm optimization; simulated annealing; combinatorial optimization.

2010 AMS Subject Classification: 65Y04.

*Corresponding author

E-mail address: wamustafa@yahoo.com

Received February 5, 2021

1. INTRODUCTION

The maximum partition problem (MPP) of a directed weighted graph with a set of nodes N and a set of edges E is to divide the set of nodes N into two nonempty disjoint subsets N_0 and N_1 such that the sum of the weights of edges that connect a node in N_0 to a node in N_1 is maximized. As an example, Fig. 1 shows a weighted directed graph (a) and its maximum partition (b). This partition can be expressed as a pair (N_0={c, d}, N_1={e,f}). The sum of the weights of edges that cross from N_0 to N_1 in this example is 22.

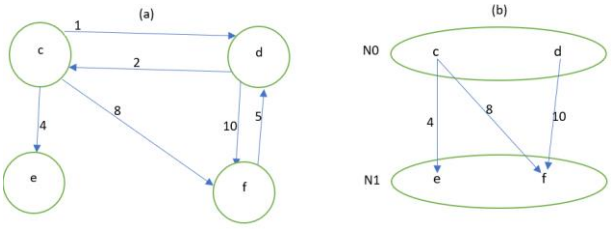

Figure 1. A weighted directed graph and its maximum partition. (a) Weighted directed graph, (b) maximum partition.

MPP applications include data clustering where a set of objects needs to be separated into classes such that the distance (dissimilarity) between objects in different classes is maximized [1-3]. Another application of MPP is in text compression that supports direct search in compressed data [4]. In this application, nodes in the graph represent characters and the weight of an edge from character x to character y represents the number of occurrences of the string xy in the text that is being compressed. The maximum partition of the set of characters is used to find the pairs of characters that occur consecutively the most in the text. Compression is achieved by coding such pairs using single unused characters in the text.

The MPP problem is NP-hard problem with a large search space. For a directed graph with n nodes, the number of possible partitions is

$$\sum_{i=1}^{n-1} \binom{n}{k} = 2^n - 2.$$

This is the number of subsets of the set of n nodes excluding the empty set and the set itself. These two subsets are excluded from possible partitions because they assign all nodes to one side of the partition and result in no edge crossing between the two sides of the partition; the total weight of crossing edges is zero.

Existing methods [6-8] to solve the MPP, known as heuristic or approximate methods, are often efficient with respect to execution time, but they do not guarantee an optimal solution. These methods suffer from the local maximum problem where a good solution is reached that is better than the neighboring solutions, but is not better than a global optimal solution. Discrete particle swarm optimization (DPSO) have been widely used to solve combinatorial problems effectively [9-12].

DPSO converges to a good solution for MPP in a relatively short time. However, solutions obtained from DPSO alone are often not optimal and these solutions can be improved further using simulated annealing (SA) which has the ability to escape a local maximum solution. This is achieved by accepting moves that result in slightly worse solutions. After accepting a bad move, SA often recovers from the move and has a good chance of reaching better solutions later. This paper presents a hybrid DPSO-SA algorithm to solve the MPP. The proposed algorithm consists of two phases. First, DPSO executes until improvement in fitness slows down. Then, SA starts and continue on improving the solution produced by DPSO. The SA phase in the proposed algorithm is augmented with a heuristic method that moves nodes between two sides of the partition to improve fitness further.

The remaining sections of this paper proceed as follows. Section two reviews algorithms related to the proposed algorithm. Section three formulates the MPP problem and discuses concepts and parameters needed to solve MPP using DPSO and SA. Section four presents the proposed algorithm and section five discuss the results of the algorithm. Finally, section six concludes the paper and suggests future work.

2. BACKGROUND

This section describes the algorithms that are used in the proposed hybrid algorithm.

2.1 Particle Swarm Optimization

The basic particle swarm optimization (PSO) mimics the collective behavior of a group of birds searching for food location [13-15]. To solve a particular problem, a population of random

solutions, called particles, is first created. The set of all particles is called swarm. The algorithm searches for a high-quality solution by updating particles over generations. In PSO, each particle has current location, fitness, velocity, local best location, and global best location. Fitness of a particle, which represents the quality of solution, is computed using certain objective function that is usually problem dependent. The local best of a particle is the best location the particle encountered so far during execution while global best is the best location found by the entire swarm so far.

In basic PSO, each particle X at iteration i has a location vector $X_i = (x_{1i}, x_{2i}, \ldots, x_{ni})$, and also has a velocity vector $V_i = (v_{1i}, v_{2i}, .., v_{ni})$, where n is usually the number of variables of the continuous function being optimized. The velocity vector components are real numbers within a limited range. The velocity vector determines the movement of the particle in the next iteration of the algorithm. At each iteration, velocity is updated and used to update the position of the particle using the two equations:

$$V_{i+1} = V_i + c_1 r_1 (local\_best - X_i) + c_2 r_2 (global\_best - X_i)$$

$$X_i = X_i + V_{i+1}$$

Where r_1 and r_2 are two random variables in the range of zero to one, c_1 and c_2 are positive constants that control how how much the particle moves in the direction of local best and global best, respectively. The component $c_1 r_1 (local\_best - X_i)$ is known as personal influence which moves the particle in the direction of local best, while $c_2 r_2 (global\_best - X_i)$ is known as social influence which moves the particle in the direction of global best solution.

2.2 Discrete Particle Swarm Optimization

The basic PSO has been found to be efficient in optimizing continuous functions in various domains. However, many practical problems, such as MPP, travelling salesman, and N-queens are combinatorial optimization problems that involve optimizing discrete functions. The work in [16] presented a discrete binary version of PSO (DPSO) for such problems. As in basic PSO, a particle has a location and velocity. However, in DPSO location is a vector of bits and every bit has a velocity that determines the probability of the bit being set to one or zero. In DPSO, higher velocity

increases the chances of bit to be set to one, while lower velocity increases the chance of the bit to be set to zero. Like the basic continuous PSO, the formula used to compute the velocity of a bit at the next iteration in DPSO is

$$V_{i+1} = V_i + c_1(local\_best - X_i) + c_2(global\_best - X_i)$$

Where c_1 and c_2 are small constants. *Local_best*, *global_best*, and X_i are all vectors of bits. In [16], the authors suggest that velocity range to be between -6 and +6. After computing the velocity of a bit *b* in a particle, the value of *b* in the next iteration is determined as follows:

$$\text{if } (rand\ () < Sig(v)) \text{ then } b = 1; \text{ else } b = 0,$$

where *Sig* is the sigmoid function, defined as $Sig(x) = 1/(1+e^{-x})$, and *rand*() is a random number between zero and one. For a bit with high velocity *v*, *Sig*(*v*) is closer to one and the bit will likely to be one in the next iteration. On the other hand, for a bit with low velocity *v*, *Sig*(*v*) is closer to zero and it is likely that the bit to be zero in the next iteration.

2.3 Simulated Annealing

The concept of annealing in combinatorial optimization was introduced in [17] and has been used in the optimization of various combinatorial problems [18-23]. This concept is based on a strong analogy with the physical annealing of metal objects. This process can be summarized by the following two steps: bring the solid metal object to a very high temperature until melting; then cool the object according to a specific temperature decreasing schedule to reach a solid state of a desired shape. When used in optimization, the algorithm usually starts from a random state representing a solution attempt and improves the quality of the current solution (state) by moving to a better neighboring state. When this process is repeated, the algorithm often results in high-quality solutions. In the remainder of this paper, the two terms *solution* and *state* are used interchangeably.

A basic characteristic of SA algorithm is its ability to accept, with some computed probability, moves to less desired states. The probability of accepting such moves depends on $e^{\Delta/temp}$, where e=2.178, *temp* is the current temperature, and Δ is a negative value representing the difference between current and successor state fitness. Higher temperature values increase the probability of

accepting bad moves while higher |Δ| values decrease the probability of accepting bad moves.

In the early stages of SA, the temperature is usually set high. This increases chances of the SA algorithm to accept moves that worsen the current solution. Through accepting bad moves, SA algorithm often avoids local maximum's and explores the search space more thoroughly. As the temperature decreases while the algorithm executes, the algorithm tends to accept only moves that improve the current solution. Near the end of execution, the temperature becomes usually near zero and no move that results in a worse solution than the current solution is accepted. Fig. 2 contains the general framework of the SA algorithm.

```
     Simulated_Annealing_Algorithm
1        Create initial state C randomly;
2        F_C=Fitness of C;
3        for (temp= temp_max; temp >=temp_min; temp=next_temp(temp)){
4           for (i=0; i<i_max; i++) {
5              N=successor of C;
6              F_N=Fitness(N);
7              Δ=F_N-F_C;
8              if (Δ>0)
9                 C=N;
10             else
11                if (e^(Δ/temp) > random(0,1))
12                   C=N;
13                else
14                   C=C
15          }
16       }
17       Output C;
18    End
```

Figure 2. Conventional simulated annealing algorithm.

The inner *for* loop, line 4, controls the number of chances the algorithm is given to move to a worse state during different temperature iterations. This loop also affects the execution time and quality of the output solution. Larger number of iterations for this loop increases the chances of escaping local maximum's, but increases the execution time. Often, the parameters of the outer temperature loop, known as the cooling schedule, and the number of iterations of the inner loop, *i_max*, are interrelated.

3. PROBLEM FORMULATION

This section formulates the concepts and operations used in applying DPSO and SA to the MPP. This includes defining: state, particle, neighboring states, successor operation, and fitness function.

A directed weighted graph $G= (N, E)$, where N is the set of nodes and E is the set of weighted directed edges, is represented by a square $|N|x|N|$ adjacency matrix A. Elements of A represent the weights of edges in E. The element in the i'th row and j'th column in A is equal to the weight of the edge from node i to node j in G; $A=[a_{ij}]$, $a_{ij}=w_{i\to j}$. A state in SA and also a particle in DPSO is a partition of N into two disjoint subsets, N_0 and N_1. A partition (N_0, N_1) is represented by an n-tuple of bits $(b_1, b_2, \ldots, b_n)$, where n is the number of nodes in G and

$$b_i\,(1\le i\le n) = \begin{cases} 0, & i'\text{th node in } N \in N0 \\ 1, & i'\text{th node in } N \in \mathrm{N1} \end{cases}$$

In SA, the successor operation takes a partition represented by a tuple of bits and returns a neighboring partition by complementing one of the bits in the tuple. Hence, a partition represented by an n-tuple has n possible successors.

The fitness of either a partition, a state, or particle is the sum of the weights of directed edges that leave N_0 to N_1. For a partition $p= (b_1, b_2, \ldots, b_n)$, the fitness is computing using the formula

Fitness(p)= $\sum_{i,j=1}^{n} A[i][j]$, b_i =1 and b_j =0,

where the indexes i and j go over all the n bits in the tuple p and $A[i][j]$ is the weight of the edge connecting node i to node j in the graph being partitioned; G.

4. THE HYBRID ALGORITHM

The proposed hybrid algorithm is shown in Fig. 3. The DPSO is implemented in lines 2-35. Lines 2-4 initialize the M particles (swarm) with N random bits in each particle, where N is the number of nodes in the graph. Lines 5-7 initialize the velocities of the bits in the particles to real numbers in the range -V_MAX to +V_MAX. Lines 8-14 initialize the local best for the particles. Line 15 initializes the *global best* particle to the particle with highest fitness in the initial random swarm.

In lines 18-35, DPSO keeps executing until either there is no improvement in fitness of the *global best* for a certain number of consecutive iterations (stagnation), or until the number of iterations reaches K. In each iteration, the velocity of each bit in every particle is first computed and then used to compute the new value of the bit. As in DPSO, bit values are computed, based on

the velocity, using the sigmoid function.

```
1   Hybrdid_PSO_SA_Algorithm
2     for (i = 0; i < M; i++)
3         for (j=0; j<N; j++)
4             Particle[i][j]=random value ∈{0,1};
5     for (i=0; i < M; i++)
6         for (j = 0; j < N; j++)
7             Velocity[i][j] = random value ∈[-V_MAX, V_MAX];
8     for (i=0; i < M; i++)
9         for (j = 0; j < N; j++)
10            Local_best[i][j] = Particle[i][j];
11    for (i=0; i < M; i++){
12        Compute Particle[i].fitness;
13        Local_best_fitness[i] = Particle[i].fitness;
14     }
15     Global_best = Particle with maximum fitness;
16     Stagnation=false;
17     Count=0;
18     for (iter=1; not(Stagnation) and iter < K; iter++) {
19         for (i = 0; i < M; i++) {
20             for (j = 0; j < N; j++) {
21                 Velocity[i][j] = Velocity[i][j] +3*(Local_best[i][j] – Particle[i][j]) +
                   3*(Global_best[j] - Particle[i][j]);
22                 if (Velocity [i][j] > V_MAX) Velocity [i][j] = V_MAX;
23                 if (Velocity [i][j] < -V_MAX) Velocity[i][j] = -V_MAX;
24                 if (random(0,1) < sigmoid(V[i][j])) Particle[i][j]=1; else Particle[i][j]=0;
25             }
26           Compute Particle[i].fitness;
27           if (Particle[i].fitness > Local_best_fitness[i]){
28               Particle[i].Local_best = Particle[i]; Local_best_fitness[i]= P[i].fitness;
29           }
30         }
31     }
32     Update Global_best;
33     if (Global_best does not improve){ Count++; if (Count==STAGNATION_INDEX) Stagnation=true;}
34     else Count=0;
35    }
36    for (temp= temp_max; temp >=1.0; temp=temp/2) {
37      for (i=0; i<i_max; i++) {
38          S=successor of Global_best;
39          Compute Global_best_fitness;
40          Compute S_fitness;
41          Δ=S_fitness - Global_best_fitness;
42          if (Δ>0) Global_best=S;
43          else if (e^(Δ/temp) > random (0,1)) {
44                  Global_best=S;
45                  if (random(0,1)<HA_PROB) Global_best = HA(Global_best);
46              }
47      }
48     }
49    Output Global_best;
50  End.
```

Figure 3. Hybrid DPSO-SA to solve MPP

Lines 16-17 initialize the variables used to implement stagnation. When DPSO executes a number of consecutive iterations without an improvement of the global best fitness, the algorithm considers this a stagnation and terminates the DPSO phase. The

number of consecutive iterations that define stagnation is represented using *STAGNATION_INDEX*.

At line 36, simulated annealing starts and continue on improving the global best partition obtained using DPSO. During each iteration of the temperature loop, SA either moves to a better partition (line 42), rarely moves to a worse partition (line 44), or stays at the same partition. When SA moves to a worse partition, it occasionally uses the heuristic algorithm (HA), to improve the fitness of this partition, line 45. To keep execution time low, the HA is performed with a low probability; HA_PROB. Based on experimentation, executing the HA in every iteration of SA does not improve fitness of the partition further. The HA, shown in Fig. 4, was suggested in [4] to find pairs of characters that occur often consecutively in a given text. The original algorithm is slightly modified for integration within the proposed algorithm. Here, the HA takes a partition p, represented by a vector of n bits, and keeps flipping bits to improve the fitness of the partition. The HA terminates when flipping any of the n bits does not improve the fitness of partition p.

```
Heuristic_Algorithm (p)
    Create an empty queue Q;
    for each node i from 1 to n do
      enqueue (i, Q);
    while (Q is not empty do) {
      i = dequue(Q);
      if (flipping i'th bit in p improves the fitness) {
        flip the i'th bit in p;
        for each node i from 1 to n do
          if (flipping i'th bit in p improves fitness)
            enqueue (i, Q);
      }
    }
    Output p;
End
```

Figure 4. Heuristic algorithm to improve partition fitness.

Since the SA goal is to improve the fitness of the near-optimal state produced by DPSO, SA needs a maximum temperature that results in a low probability of accepting bad moves. Through experimentation with a large number of graphs of various size, this low probability is about 0.25. This low probability discourages SA from accepting bad moves and increases the chances of maintaining the current good partition or improving it further.

The probability of moving to a worse partition is implemented in SA using $e^{\Delta/temp}$, where Δ is the difference in fitness between current partition and its successor, and temp is the current

temperature. So, in order to have the probability of moving to a worse partition in the early stages of SA to be about 0.25, we need $e^{\Delta/temp\_max} \cong 0.25$, which is equivalent to $\frac{\Delta}{temp\_max} \cong 1.38$. After monitoring the negative Δ values inside temperature loop for a large number of random graphs of different size, the author often finds that $\Delta \cong 0.35 * W_{max} * \sqrt{n}$, where W_{max} is the maximum weight an edge may have and *n* is the number of nodes in the graph. Hence, the maximum temperature used in SA needs to be about $0.25 * W_{max} * \sqrt{n}$.

To give SA less chances to move to a worse state and also to reduce execution time, SA in the hybrid algorithm reduces temperature rapidly between successive iterations using the cooling function $temp_i = \frac{temp_{i-1}}{2}$, where $temp_i$ is the temperature at iteration *i*. The author experimented with other cooling functions including *exponential multiplicative cooling:* $temp_i = \alpha\ temp_{i-1}$, where $0.8 \leq \alpha \leq 0.9$, *linear multiplicative cooling:* $temp_i = temp_0/(1+i)$, *quadratic multiplicative cooling:* $temp_i = temp_0/(1+i^2)$. These cooling functions increased execution time without much improvement of output partition fitness.

In the SA phase of the hybrid algorithm, the number of iterations of the inner loop (line 37), *i_max*, needs to be high, compared to conventional simulated annealing, due to the cooling function being used which decreases temperature rapidly. The large *i_max* value gives SA more chances to find a better neighboring partition before reducing the temperature. Smaller *i_max* values reduce the chances of SA to recover from the local maximum problem and often result in partitions with less fitness. On the other hand, too large *i_max* values increase execution time without much gain in fitness.

5. RESULTS AND DISCUSSION

To compare the performance of conventional DPSO algorithm with the proposed hybrid DPSO-SA algorithm, the two algorithms were implemented as functions in a single program in visual C++ 2019. Conventional DPSO was implemented based on the parameters shown in Table 1 and the proposed hybrid DPSO-SA algorithm was implemented based on the parameters in Table 2. The resulting code was executed on an HP computer with an i7 CPU, running Windows 10; 2.2

GHz speed; and an eight GB of main memory. Each algorithm was applied to 25 directed graphs G_i, 1<=i<=25. Graph size starts at 100 nodes, increases by 100, until it reaches 500. The 25 graphs are distributed equally over the mentioned sizes; 5 graphs per each size. All of the graphs are complete; for every two different nodes *x* and *y*, there is a directed weighted edge from *x* to *y* and there is another directed weighted edge from *y* to *x*. For example, graph *G25* has 500 nodes and 500*499 edges.

Table 1. The parameters of the conventional DPSO algorithm.

Number of particles (*M*)	20
V_MAX	6
Number of iterations (*K*)	1000

Table 2. The parameters of the Hybrid DPSO-SA algorithms.

Number of particles (*M*)	20
V_MAX	6
Number of iterations (*K*)	1000
temp_max	340*$\sqrt{n}$
i_max	400
HA_PROP	0.2
STAGNATION_INDEX	10, 20, 30

To ensure better randomness of the weights in the graphs and to obtain reproducible results, weights in each graph are based on a different seed; weights in graph G_i, 1<=*i*<=25, are obtained by first calling *seed*(*i*) C++ function, and then calling *rand()* function to generate the weights themselves. The range of random weights in the graph is restricted to be from 0 to 100 to avoid an extremely large sum of weights (fitness); a complete graph of 500 nodes has about 250,000 directed edges and the max fitness value is 25,000,000. Increasing the maximum weight limit results in fitness values that are too large to store using unsigned *long long* in C++.

The hybrid DPSO-SA algorithm was executed on each graph 3 times using different stagnation index values: 10, 20, 30. The results of applying the two algorithms to the 25 graphs are given in

appendix A. Fig. 5 shows execution time, to the nearest second, for conventional DPSO and hybrid DPSO-SA with stagnation index values 10, 20, and 30. The execution time at each graph size was computed as the average execution time of five graphs of the same size. For larger graphs, the hybrid DPSO-SA algorithm with stagnation index 10 results in less execution time than both conventional DPSO and also the hybrid DPSO-SA itself using stagnation index values 20 and 30. For stagnation index value of 30, the hybrid DPSO-SA execution time becomes worse than that of conventional DPSO. Larger stagnation index value means that there is a higher chance that DPSO phase of the hybrid algorithm executes the entire *K* iterations. In such cases, the execution time of the hybrid DPSO-SA algorithm will be the sum of execution time of conventional DPSO in addition to the execution time of SA.

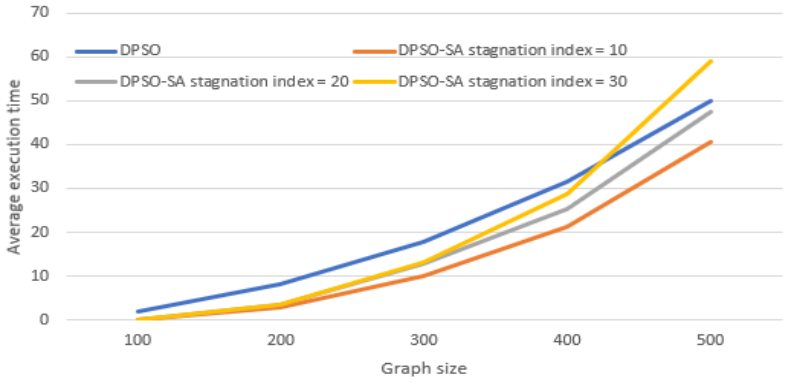

Figure 5. Execution time to the nearest second of DPSO and hybrid DPSO-SA algorithms.

Fig. 6 shows average number of DPSO iterations after which stagnation occurs at different graph size for stagnation indeces 10, 20, and 30. For smaller graphs of 100 nodes, it takes about 100 iterations of DPSO to stagnate. For larger graphs of 500 nodes, it takes about 300-700 iterations to stagnate; depending on stagnation index value being used. In general, it takes larger number of iterations for a stagnation to occur as graph size increases and also as stagnation index increases. This suggests using a larger stagnation index values for larger graphs in the proposed hybrid DPSO-SA algorithm.

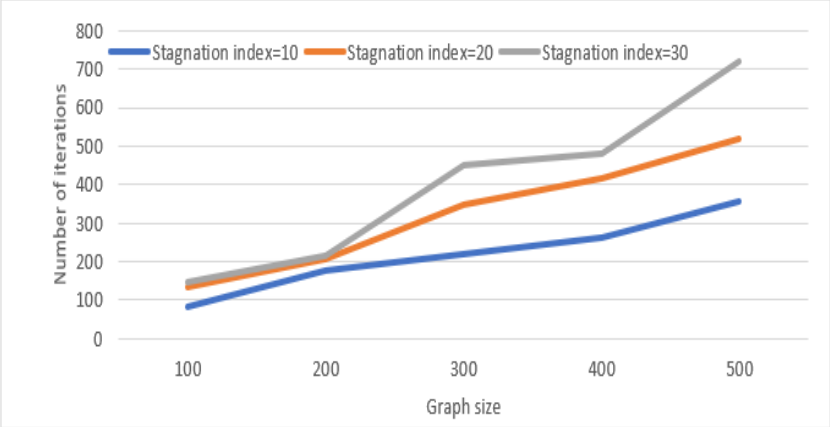

Figure 6. DPSO stagnation

The average fitness improvement of the hybrid DPSO-SA algorithm over conventional DPSO for each graph size is shown in Fig. 7. The hybrid DPSO-SA algorithm results in higher fitness using each of the three stagnation indexes 10, 20, and 30 at every graph size. Furthermore, the improvement in fitness becomes larger as graph size increases. For a graph size of 100, average improvement in fitness is about 300 whereas for a graph of size 500 nodes the average fitness improvement is about 4000. For smaller graphs, stagnation index 10 results in most fitness improvement, while for larger graphs stagnation index of 30 results in most fitness improvement. Again, this suggests using a larger stagnation index in the hybrid DPSO-SA algorithm for large graphs.

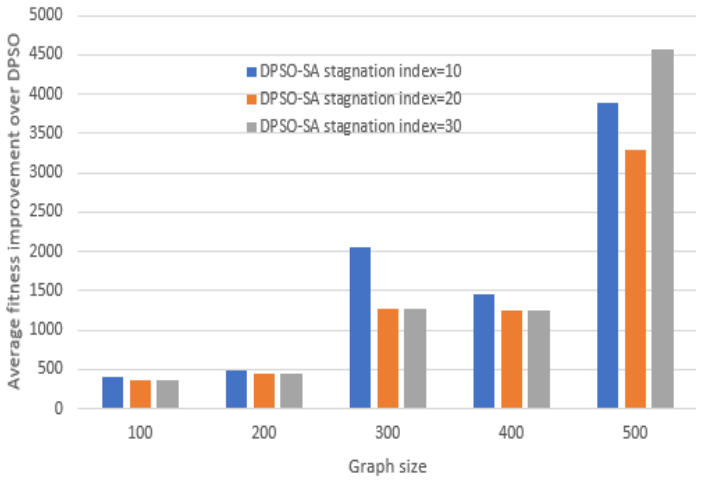

Figure 7. Average fitness improvement of hybrid DPSO-SA over conventional DPSO

Fig. 8 shows convergence of conventional DPSO and hybrid DPSO-SA with time for the

graphs *G10* (200 nodes; 39,800 edges), *G15* (300 nodes; 89,700 edges), *G20* (400 node; 159,600 edges), and *G25* (500 nodes; 249,500 edges). The results in Fig. 8 were obtained by performing the two algorithms on the same graph for K=1000 iterations, and using stagnation index value 10 in the hybrid DPSO-SA. Global best fitness in the two algorithms was recorded every 2 seconds during execution time. Although the hybrid DPSO-SA starts at global best with a lower fitness in the four graphs, the hybrid DPSO-SA algorithm is able to catch up with conventional DPSO and reach a better solution later. SA improves global fitness the most in its early iterations. This occurs after about 4 seconds in *G10*, 6 seconds in *G15*, 14 seconds in *G20*, and 18 seconds in *G25*. This illustrates the benefit of the cooling schedule used in SA phase of the algorithm in which temperature is reduced rapidly; Using this schedule, SA is able to improve fitness of the solution produced by DPSO in a short time

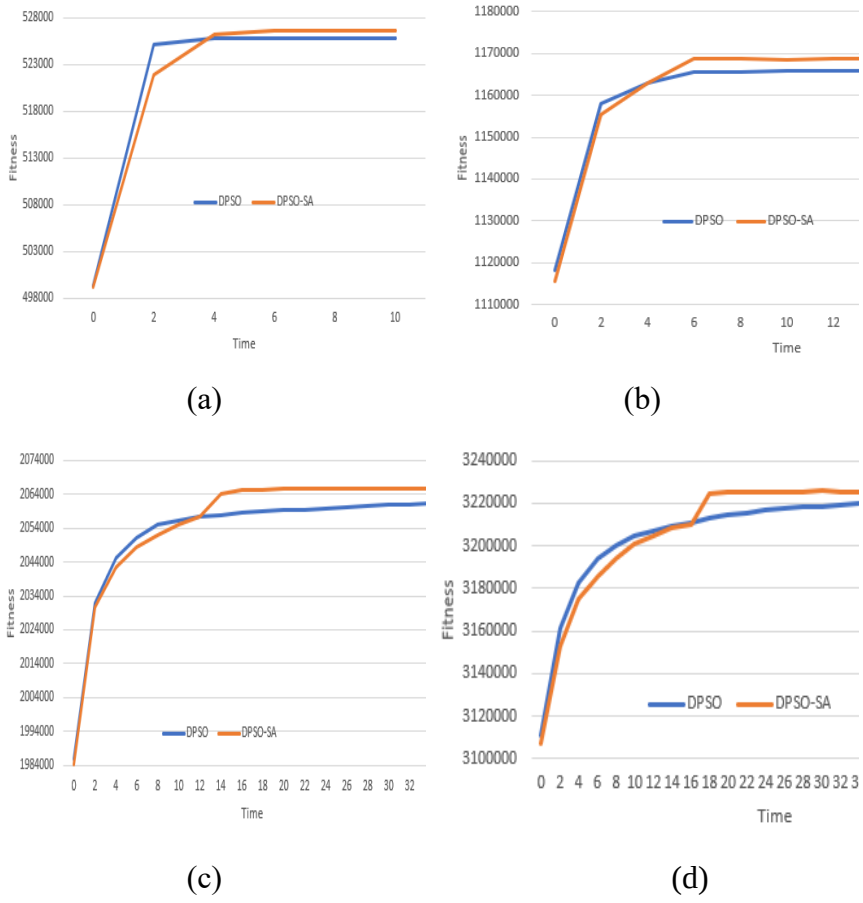

Figure 8. Convergence of conventional DPSO and hybrid DPSO-SA. (a) Graph *G10*, (b) Graph *G15*, (c) Graph *G20*, (d) Graph *G25*.

6. CONCLUSION AND FUTURE WORK

This paper presented a hybrid algorithm to solve the MPP. The hybridization is implemented by performing simulated annealing on the result of applying discrete particle swarm optimization to MPP. Experiments on random graphs with different size show that the proposed algorithm produces better partitions than conventional DPSO. Furthermore, the proposed algorithm converges faster to near optimal solutions than conventional DPSO.

Future work includes quantifying the relation between graph size and the most appropriate stagnation index value. Another future work, is to extend the proposed algorithm to deal with partitioning into more than two disjoint subsets, apply the resulting algorithm to data clustering and compare the quality of the resulting clusters against exiting methods.

APPENDIX A.

Results of applying conventional DPSO and hybrid DPSO-SA to 25 random graphs. On each graph, the hybrid DPSO-SA is performed three times using stagnation index values 10, 20, and 30.

Graph	Size	DPSO fitness	DPSO time	DPSO-SA (Stag 10) fitness	DPSO-SA (Stag 10) time	DPSO-SA (stag 10) iteration	DPSO-SA (Stag 20) fitness	DPSO-SA (Stag 20) time	DPSO-SA (stag 20) iteration	DPSO-SA (Stag 30) fitness	DPSO-SA (Stag 30) time	DPSO-SA (stag 30) iteration
G1	100	135224	2	135701	0	99	135673	0	168	135578	0	99
G2	100	133948	2	134193	0	46	134122	0	179	134250	0	189
G3	100	133731	2	134116	0	79	134146	0	89	134090	0	122
G4	100	137037	2	137261	0	99	137261	0	109	137193	0	172
G5	100	134724	2	135366	0	91	135275	0	123	135366	0	157
G6	200	523716	8	523286	3	158	523328	3	194	523597	3	286
G7	200	525192	8	526810	3	148	526780	3	189	526331	3	278
G8	200	526747	8	526968	3	172	526968	3	182	527771	3	192
G9	200	526252	8	526270	3	191	526478	4	233	525980	4	290
G10	200	524507	8	525520	3	206	525100	4	232	525569	4	370
G11	300	1165233	18	1167588	9	170	1163388	11	312	1167302	11	322
G12	300	1167449	17	1168599	9	189	1168592	14	173	1168426	16	571
G13	300	1172814	18	1174719	10	254	1175023	10	264	1174595	10	274
G14	300	1169505	18	1171437	11	287	1171390	14	485	1171138	15	573
G15	300	1165777	18	1168675	11	283	1168688	15	512	1168668	14	522
G16	400	2072590	31	2073775	20	209	2073854	26	477	2073794	29	522
G17	400	2065784	32	2066996	20	215	2065187	26	426	2065187	29	479
G18	400	2073504	31	2072575	22	288	2072658	27	504	2072793	34	705
G19	400	2063948	31	2065149	20	227	2066479	23	299	2066317	24	309
G20	400	2061157	32	2065759	24	367	2065100	24	377	2065180	27	387
G21	500	3217073	51	3220260	43	380	3219100	53	631	3220762	65	803
G22	500	3217572	50	3222099	41	407	3219825	47	566	3222056	72	1000
G23	500	3213520	49	3216541	42	362	3217610	44	434	3218514	53	540
G24	500	3208369	50	3213093	39	340	3212565	45	449	3213784	44	459
G25	500	3221706	50	3225638	38	299	3225594	48	516	3225929	61	786

CONFLICT OF INTERESTS

The author(s) declare that there is no conflict of interests.

REFERENCES

[1] E. Hancer, D. Karaboga, Comprehensive survey of traditional, merge-split and evolutionary approaches proposed for determination of cluster number, Swarm Evol. Comput. 32 (2017), 49-67.

[2] R. Xu, D. Wunsch, Survey of clustering algorithms, IEEE Trans. Neural Networks. 16 (2005), 645-678.

[3] K. Premalatha, A. Natarajan, A literature review on document clustering, Inform. Technol. J. 9 (2010), 993-1002.

[4] U. Manber, A text compression scheme that allows fast searching directly in the compressed file, ACM Trans. Inform. Syst. 15 (1997), 124–136.

[5] B. Kernighan, S. Lin. An efficient heuristic procedure for partitioning graphs, Bell Syst. Techn. J. 49 (1970), 91–307.

[6] S. Dutt, New faster Kernighan-Lin-type graph-partitioning algorithms, in: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), IEEE Comput. Soc. Press, Santa Clara, CA, USA, 1993: pp. 370–377.

[7] Hyunchul Shin, Chunghee Kim, A simple yet effective technique for partitioning, IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 1 (1993), 380–386.

[8] C.M. Fiduccia, R.M. Mattheyses, A Linear-Time Heuristic for Improving Network Partitions, in: 19th Design Automation Conference, IEEE, Las Vegas, NV, USA, 1982: pp. 175–181.

[9] J. Qin, Hybrid discrete particle swarm algorithm for graph coloring problem, J. Computers. 6 (2011), 1175-1182.

[10] S.D. Kapade, S.M. Khairnar, B.S. Chaudhari, A new particle swarm intelligence-based graph partitioning technique for image segmentation, J. Electric. Syst. Inform. Technol. 7 (2020), 4.

[11] Y.-R. Wang, H.-L. Lin, L. Yang, Swarm Refinement PSO for Solving N-queens Problem, in: 2012 Third International Conference on Innovations in Bio-Inspired Computing and Applications, IEEE, Kaohsiung City, Taiwan, 2012: pp. 29–33.

[12] N. Blas, O. Tolic, Clustering using particle swarm optimization, Int. J. Inform. Theor. Appl. 23 (2016), 24-33.

[13] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95 - International Conference on Neural Networks, IEEE, Perth, WA, Australia, 1995: pp. 1942–1948.

[14] R. Eberhart, R. Dobbins, P. Simpson, Computational Intelligence PC Tools. Academic Press, Boston, (1996).

[15] J. Kennedy, The particle swarm: social adaptation of knowledge, in: Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97), IEEE, Indianapolis, IN, USA, 1997: pp. 303–308.

[16] J. Kennedy, R.C. Eberhart, A discrete binary version of the particle swarm algorithm, in: 1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, IEEE, Orlando, FL, USA, 1997: pp. 4104–4108.

[17] S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimization by simulated annealing, Science. 220 (1983), 671-680.

[18] X. Geng, Z. Chen, W. Yang, D. Shi, Kai Zhao, Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search, Appl. Soft Comput. 11 (2011), 3680-3689.

[19] C. Gallo, V. Capozzi, A simulated annealing algorithm for scheduling problems, J. Appl. Math. Phys. 7 (2019), 2579-2594.

[20] D. Brown, C. Huntley, A practical application of simulated annealing to clustering, Pattern Recognition. 25 (1992), 401-412.

[21] H. Samma, J. Mohamad-Saleh, S. Azmin Suandi, B. Lahasan, Q-learning-based simulated annealing algorithm for constrained engineering design problems, Neural Comput. Appl. 32 (2020), 5147–5161.

[22] C. Liu, Y. Zhang, Research on MTSP Problem based on Simulated Annealing, in: Proceedings of the 2018 International Conference on Information Science and System, ACM, Jeju Republic of Korea, 2018: pp. 283–285.

[23] W. Odziemczyk, Application of simulated annealing algorithm for 3D coordinate transformation problem Solution, Open Geosci. 12 (2020), 491–502.