Alternative search space representation and dynamic Inertia for solving the Knight Covering Problem using Particle Swarm Optimisation

FRANZ HERM (720096430), University of Exeter, United Kingdom

This project investigates the effect of utilising an alternative search space representation and an inertia weight update strategy to solve the Knight Covering Problem using Particle Swarm Optimisation. The proposed algorithm was tested along with several baseline algorithms and their performance was assessed based on the required number of fitness evaluations, utilised knights and the success rate of finding solutions. While the inertia update led to improved convergence behaviour for the binary and integer encoding, the integer search space representation did not display a performance improvement. However, more research is needed to comprehensively assess the proposed approaches' effectiveness.

Additional Key Words and Phrases: PSO, Knight Covering Problem, Inertia update, Penalty function, Search space, Hillclimber, Random search

1 INTRODUCTION

The Knight Covering Problem is part of the set of mathematical chess puzzles and aims at covering all squares of an $n \times n$ chessboard using the minimum number of knights. Although optimal solutions have been verified for small board sizes [1], the problem complexity rises exponentially when the number of squares increases. While Particle Swarm Optimisation (PSO) is a well-known population-based algorithm for quickly obtaining locally optimal solutions for *continuous problems*, a binary variant has been proposed in [2] allowing the technique's application to discrete tasks. This variant was previously used in [3] to solve the Knight Covering Problem. However, the authors detected multiple downsides to their approach, including the stark increase in complexity for larger board sizes due to their binary search space representation. Furthermore, various strategies for improving the initial binary PSO have been proposed in [4] and [5]. This project aims to apply these optimisations to the approach in [3] to assess their effect on overall performance and the convergence behaviour of the algorithm. The main changes applied to the algorithm include the proposal of an alternative search space representation and the utilisation of a dynamic inertia update strategy. After describing the related work and previous research conducted on the Knight Covering Problem and Particle Swarm Optimisation in section 2, section 3 explains the proposed changes and the resulting PSO algorithm. Following that, section 4 features the employed baseline algorithms, conducted experiments, and a discussion about potential improvements for future work.

2 RELATED WORKS

2.1 Knight Covering Problem

The *Knight Covering Problem* (KCP) is a mathematical puzzle that aims to find a configuration of knights on an $n \times n$ chessboard, so that each square is occupied or attacked by at least one knight. Determining the minimum number of knights needed – also known as the domination number – is a non-trivial task and no analytical exact or asymptotic solution is known for arbitrary board sizes [6]. However, there exists an upper bound, which is obtained by putting knights in diagonal lines on the board [7], as shown in Figure 1. This results in the following equation's maximum number of required knights:

$$K(n) = \begin{cases} \frac{1}{2}n^2 & n > 2 \land n \text{ even} \\ \frac{n^2+1}{2} & n > 1 \land n \text{ odd} \end{cases} \tag{1}$$

The domination numbers for small board sizes have been found and verified by hand. A list of them for board sizes up to 21×21 can be found in [1]. The number of research papers that try to solve the KCP using algorithmic approaches is rather limited. In [8], the authors used a hybrid backtracking approach for a 4×4 board, while [3] explores the influence of PSO parameters for small board sizes of the KCP. The latter will be explained in more detail in subsection 2.3.

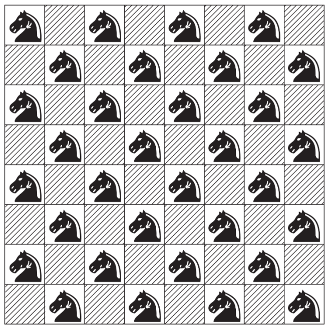

Fig. 1. Trivial solution to the knight covering problem by putting knights on diagonal lines [7]

2.2 Particle Swarm Optimisation

Initially proposed in [9], *Particle Swarm Optimisation* (PSO) is a meta-heuristic for solving non-linear continuous functions. The algorithm stores a set of solutions called particles. Unlike *Evolutionary Algorithms*, the solutions are moved through the search space by adding velocity values to their current location at every timestep. The velocity of a particle i is then updated based on its distance to the global best solution (global component) and the particle's best past solution (local component) [10]; see Equation 2.

$$\begin{aligned} L(i,t) &= \text{pbest}(i,t) - P_i(t) \\ G(i,t) &= \text{gbest}(t) - P_i(t) \\ V_i(t+1) &= \omega V_i(t) + c_1 r_1 \times L(i,t) + c_2 r_2 \times G(i,t) \end{aligned} \tag{2}$$

Here, $L(i,t)$ is the difference in fitness value between the historic best position of particle i up to timestep t and the current particle position, while $G(i,t)$ is the difference between the global best position at timestep t and the particle's current position. The constants c_1 and c_2 specify the importance of the local and the global components in updating the velocity, respectively. The random variables r_1 and r_2 are used to introduce additional variation, which helps to prevent particle movement from converging to an unchanging direction. The inertia parameter ω can be used to adjust the trade-off between exploration and exploitation as it defines the sensitivity of the particle to historic fitness values [10]. According to [4], adjusting the inertia value during the optimisation process can drastically affect the quality of the results obtained. The paper features a survey on various inertia update strategies, one of the most effective being linear inertia reduction.

2.3 Binary PSO

Binary PSO was proposed two years after the original algorithm and allowed PSO to be applied to binary search spaces [2]. In this version of the algorithm, the difference between two particle positions is computed using the *hamming distance* between the corresponding bit strings. The velocity vector consists of values between zero and one, indicating the probability of the particle position changing to 1. This normalisation is achieved by applying the sigmoid function (S) to the velocity values. The velocity update equation is the same as for standard PSO. The dimension d of the ith particle's position is then updated according to Equation 3, where $rand()$ is a random number.

$$x_{id}(t+1) = \begin{cases} 1 & \text{, if } rand() < S(v_{id}(t)) \\ 0 & \text{, otherwise} \end{cases} \tag{3}$$

$$x_{id}(t+1) = \begin{cases} \overline{x_{id}}(t) & \text{, if } rand() < S(v_{id}(t)) \\ x_{id}(t) & \text{, otherwise} \end{cases} \tag{4}$$

However, this initial version of the algorithm contains various flaws, addressed in [5]. Most prominently, the new position of a particle depends only on its velocity and not on the current position. Combined with the fundamentally different interpretation of velocity, this leads to small absolute velocities – which typically indicate a favourable particle position – to drastically change the location of a particle due to $S(0) = 0.5$. Thus, the authors of [5] proposed an alternative position update, where they flip the bit values of a position if $S(v_{id}) > rand()$ and otherwise keep them the same; see Equation 4.

In [3], the authors use the initially proposed variant of binary PSO with constant inertia to solve the KCP. For the search space, they use bit strings of size n^2, where n is the length of the chessboard and positive bits indicate the presence of a knight on the corresponding square. The fitness function of a particle i is shown in Equation 5.

$$F_i = E \times w_1 + P \times w_2 + \frac{C}{D} \tag{5}$$

E represents the number of empty squares on the board, while P is the number of knights. C stands for the number of covered squares, and D is the dimension of the chess board. The weights w_1 and w_2 specify the importance of the corresponding summand. According to the authors, the last term is supposed to discourage particles from locations where knights are very close to each other, and the fitness function is supposed to be minimised. As a larger number of covered squares represents a more favourable situation, it should lead to a lower fitness value. However, the value of the quotient increases with the number of covered squares, which indicates that C and D should be switched. Unfortunately, the authors do not provide much additional information on the meaning of C, which would be necessary to confirm whether this constitutes an error in the fitness function.

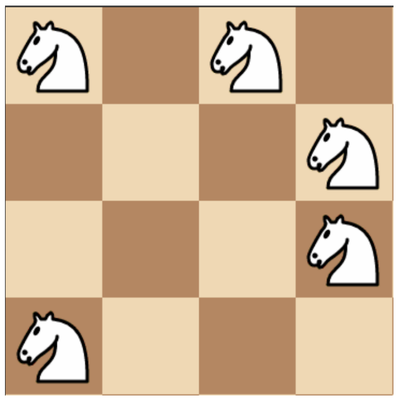
Fig. 2. Exemplary configuration on a 4×4 board, which can be represented by the vector $(1, 2, 5, 4, 1, 0, 0, 0)$ using the proposed integer encoding

3 METHODOLOGY

Regarding the binary PSO algorithm used in [3], the authors acknowledge that their binary encoding scheme would lead to complexity-related issues for large board sizes, as each board square is represented by another dimension. Furthermore, they did not employ a dynamic inertia update procedure, though these have been shown to improve convergence characteristics of the PSO algorithm, as discussed in subsection 2.2.

This project aims to improve the algorithm used in [3] by proposing an alternative search space representation, corresponding penalty functions and a dynamic inertia weight, which will be described in the following subsections.

3.1 Search Space Representation

The binary encoding used in [3] suffers from the number of dimensions necessary to represent the particle's position to increase with the board size quadratically. In the proposed alternative search space representation, the number of dimensions is based on the number of knights. Here, each element of the position vector is an element of the set $\mathbb{N}_0$ and represents the distance between two neighbouring knights. The board squares are indexed as in [3], with index numbers starting from 0 on the top left and increasing to the very right of the board before wrapping around to the left-most square of the next row. For the position of the first knight, an imaginary square with index -1 was taken as the reference point so that all zero elements of the position vector indicate an unused knight. These are shifted to the end of the vector to allow for a bijective mapping between board configuration and vector representation. Thus, the board configuration in Figure 2 would be uniquely represented by the vector $(1, 2, 5, 4, 1, 0, 0, 0)$.

The size of the vector is specified by Equation 1, exploiting the upper bound on the maximum number of knights needed to cover all squares. Therefore, the representation effectively reduces the number of dimensions by 50 per cent with respect to the binary encoding and, due to the vector elements being non-binary, does not require the use of binary PSO. The proposed search space representation will be referred to as *integer representation* in the following sections.

3.2 Penalty Functions

The integer representation introduces a new set of invalid solutions, as, by design, it does not prevent the sum of elements of a position vector from exceeding the number of squares on the board. In turn, this requirement on the validity of positions is enforced by introducing a penalty function, which is added to the fitness function to obtain the final fitness value of a solution. The penalty function is non-stationary, meaning that its value depends not only on the current solution, but also on the current iteration number. According to [11], non-stationary penalty functions have been shown to be superior to their stationary counterparts in most scenarios. The function equation is shown in Equation 6, where s is a position vector, i is the iteration number, and n is the board length. The second argument of the max function represents the distance from the last knight, specified by the position vector, to the final square of the board. Because this difference is always less or equal to zero for valid position vectors, the *max function* ensures that a penalty is only applied to invalid solutions. The function value for a specific invalid solution increases with the square root of the iteration number, leading to invalid positions becoming more unfavourable as the optimisation process goes on.

$$P(s, i, n) = \sqrt{i} \times max(0, \sum_{1}^{dim(s)} s_i - n^2) \tag{6}$$

3.3 PSO Algorithm

The pseudocode for the PSO algorithm using the proposed integer representation is displayed in Algorithm 1. The *InitialiseParticlePositions* outputs a matrix of dimension $N \times K(B)$, containing randomised initial locations for each particle. This is done by first placing a random number of knights in the range $[1, K(B)]$ on empty boards using binary encoding and then converting it to the corresponding integer representations using the procedure Algorithm 2. The K function specifies the maximum number of knights needed to solve the problem (see Equation 1) and is also used in line 4.

After all relevant variables have been initialised, the algorithm proceeds with the optimisation loop. Here, the number of fitness function evaluations was taken as the termination criterion, the reason for that will be further described in section 4. To infer the number of iterations to use in the for-loop, the number of fitness evaluations is divided by the number of particles (line 11). In line 12 to 14, the velocity is updated based on Equation 2 and clipped to an interval given by the maximum and minimum velocity. After that, the particle positions are updated by adding their velocities to their current positions and rounding the resulting vector elements to the nearest non-negative integer. Zero elements are shifted to the end of the vectors to ensure a bijective mapping between binary and integer representation. Line 17 ensures that the inertia factor is linearly reduced throughout the optimisation process from its starting value Ω_1 to Ω_2 after the last iteration. The value of the fitness function f for a single particle is obtained by adding the penalty term of Equation 6 to the ratio of uncovered squares, as shown in Equation 7. The number of uncovered squares is computed by transforming the integer representation into the binary encoding, using the inverse of the procedure in Algorithm 2. The current position and attacked squares for each knight are then stored in a list, duplicate entries are removed, and the length of the list then subtracts the total number of squares to yield the number of uncovered squares. Note that the function f in Algorithm 1 represents the vectorised version of

Algorithm 1 PSO with integer representation and dynamic inertia

```
Input:
    $B$: board length        $N$: number of particles
    $\Omega_1$: maximum inertia        $\Omega_2$: minimum inertia
    $c_1$: local velocity factor        $c_2$: global velocity factor
    $v_1$: maximum velocity        $v_2$: minimum velocity
    $I$: number of iterations
Output:
    $S$: set of best solutions for each iteration
 1: $S \leftarrow \emptyset$
 2: $P \leftarrow InitialiseParticlePositions(N, B)$
 3: $F \leftarrow f(P, B, 0)$                          ▷ initialise particle fitness
 4: $V \leftarrow rand(v_1, v_2, [N, K(B)])$ ▷ velocities with same shape as $P$
 5: $\omega \leftarrow \Omega$                              ▷ initialise inertia factor
 6: $F_{lb} \leftarrow F$                              ▷ best fitness for each particle
 7: $F_{gb} \leftarrow min(F_{lb})$                         ▷ best global fitness
 8: $P_{lb} \leftarrow P$                  ▷ best historic position of each particle
 9: $P_{gb} \leftarrow P[where(F == F_{gb})]$                 ▷ best global location
10: $S[0] \leftarrow (P_{gb}, F_{gb})$                ▷ add initial best solution to $S$
11: for each $i \in range(1, I/N)$ do
12:     $r_1 = rand(0, 1, [N, 1]), r_2 = rand(0, 1, [N, 1])$
13:     $V \leftarrow \omega V + c_1 r_1 \times (P_{lb} - P) + c_2 r_2 \times (P_{gb} - P)$
14:     $V \leftarrow Clip(V, V_1, V_2)$
15:     $P \leftarrow Round(P + V)$
16:     $P \leftarrow ShiftZeros(P)$
17:     $\omega \leftarrow \Omega_1 - i \times (\Omega_1 - \Omega_2) \div I$              ▷ linear $\omega$ update
18:     $F \leftarrow f(P, B, i)$
19:     $F_{lb}, F_{gb}, P_{lb}, P_{gb} \leftarrow UpdateBestSolutions(P, F)$
20:     $S[i] \leftarrow (P_{gb}, F_{gb})$
21: return $S$
```

Algorithm 2 BinaryToIntegerSpace

```
Input:
    $s$: binary encoded particle position
    $K$: maximum number of knights
Output:
    $S$: integer encoded particle position
 1: $S \leftarrow zeros(K)$
 2: $I \leftarrow argwhere(S == 1)$    ▷ get indices of squares with knights
 3: $I[1:] \leftarrow I[1:] - I[:-1]$ ▷ get distances between knights using
    reverse cumulative sum operation
 4: $S[:K] \leftarrow I[:K]$            ▷ copy first K knights to return vector
 5: $S[0] \leftarrow S[0] + 1$ ▷ reference square for first knight has index -1
 6: return $S$
```

Equation 7, returning a vector containing the fitness value for each particle in P instead of working with a single solution.

$$f(s, B, i) = \frac{NumUncoveredSquares(p, B)}{B^2} + P(s, i, B) \qquad (7)$$

The values of the variables containing the best solutions and their fitness values are then updated in line 19, and the global best solutions are stored in S, which is returned after the optimisation loop completes.

4 EXPERIMENTS

4.1 Baseline Algorithms

The PSO algorithm will be compared to three baseline algorithms, which all work with the binary encoding described in subsection 2.3 and use a fitness function similar to Equation 5 but with C and D switched. The first algorithm is a *random search* that randomly generates board configurations. The second algorithm implements *first-choice stochastic hill climbing* (SHC) as defined in [12]. The pseudo-code is given in Algorithm 3. For each iteration, the first neighbouring solution that is better than the current position is chosen. To allow the algorithm to overcome regions with plateauing fitness values, a tolerance value δ is employed, which defines how much of a fitness deviation between the current and neighbouring solution is considered a plateau value. The maximum number of allowed consecutive plateau steps is specified by parameter A. Adding to what is described in the pseudo-code, the implemented algorithm also keeps track of the number of fitness evaluations and stops once the corresponding termination criterion is reached. For conciseness-related reasons, this feature is not represented in Algorithm 3. The final baseline algorithm is an adaptation of the binary PSO defined in [3], which uses the optimised position update procedure (see Equation 4) and will be tested with constant and linearly decreasing inertia.

Algorithm 3 StochasticHillclimbing

```
Input:
    $B$: board length        $A$: allowed consecutive plateau steps
    $\delta$: plateau tolerance     $N$: maximum number of iterations
Output:
    $S$: list of solutions and fitness values
1:  $P \leftarrow InitialiseParticlePosition(B)$            ▷ current solution
2:  $F \leftarrow f(P)$ ▷ fitness value using adjusted version of Equation 5
3:  $S[0] \leftarrow (P, F)$
4:  $a \leftarrow 0$    ▷ current number of consecutive plateau steps taken
5:  for each $n \in range(1, N)$ do
6:      $b \leftarrow False$                     ▷ better solution found flag
7:      $I \leftarrow Shuffle(len(P))$ ▷ shuffled list of element indices of P
8:      for each $i \in I$ do     ▷ iterate through neighbour solutions
9:          $P_n \leftarrow P$
10:         $P_n[i] \leftarrow (P_n[i] + 1) \% 2$    ▷ invert bit to get neighbour
11:         $F_n \leftarrow f(P_n)$
12:         if $(F_n - F)/F < \delta$ then ▷ neighbour fitness is not worse
13:             if $(F_n - F)/F \geq 0$ then ▷ new fitness is on plateau
14:                 $a \leftarrow a + 1$
15:                 if $a > \delta$ then     ▷ num plateau steps too large
16:                     return $S$
17:             else ▷ new solution is better than current solution
18:                 $a \leftarrow 0$                          ▷ reset $a$
19:             $P \leftarrow P_n$               ▷ move to neighbouring solution
20:             $F \leftarrow F_n$
21:             $S[n] \leftarrow (P, F)$
22:             $b \leftarrow True$            ▷ set *better solution found* flag
23:             break                   ▷ leaves inner for each loop
24:     if $!b$ then ▷ no better neighbour found during this iteration
25:         return $S$
26: return $S$
```

4.2 Performance Metrics and Hyper-parameters

All algorithms were tested on board sizes ranging from 5×5 to 10×10 and each experiment, specified by the used algorithm and corresponding hyper-parameter configuration, was repeated ten times to limit the effect of randomness on the results. To ensure fair comparisons, the number of fitness evaluations was chosen as the termination criterion rather than the number of iterations and was set to 100, 000. For the PSO algorithms, the number of particles was set to 100; c_1 and c_2 in the velocity update procedure, see Equation 2, were chosen as 1. The maximum absolute velocity value was set to 4, as suggested by [2] and [3]. For the constant inertia, a value of 0.90 was used, while the dynamically updating inertia decreases from 0.9 to 0.1. For stochastic hill climbing, the maximum number of consecutive plateau steps was chosen as 200 and the plateau tolerance as 0.05.

All algorithms use an adapted version of the fitness function, shown in Equation 5, both with and without a penalty term. Here, w_1 and w_2 were set to 1. For the integer PSO, the penalty term defined in Equation 6 was used, while for all other algorithms, the penalty term consists of the number of utilised knights divided by the number of squares. The PSO algorithms were also tested with a simplified fitness function computed as the ratio of uncovered squares. The results obtained for all fitness and penalty function combinations were averaged and are reported in subsection 4.3

Multiple performance metrics are used to evaluate the tested algorithms. A run is deemed successful if the algorithm has found a solution with no uncovered squares. Optimal runs are successful runs which utilise the minimum number of knights for the specific board size, as listed in [1]. An algorithm's *success rate* (SR) can be expressed as the number of successful runs divided by the total number of runs conducted. The analogous term for optimal runs will be called *optimality rate* (OR). Furthermore, the number of function evaluations before a solution is found is used to visualise the complexity caused by larger board sizes, as done in [3]. Similarly, the same will be done for the number of knights used in successful runs. Finally, the behaviour of normalised fitness function values over time compares the algorithms' convergence speed concerning the employed inertia update strategy.

4.3 Results

The results obtained from the experiments are visualised in Figures 3 to 5 and Table 1. Figure 3 shows the spread of the minimum number of knights needed to cover all squares. To obtain this data, the number of used knights was extracted and grouped by board length and run for each data point where the number of covered squares is equal to the squared board length. Then, the minimum values for each run were averaged over all runs of the same board size to obtain the mean value. As can be seen from the plot, the random search algorithm needed the most knights to cover all squares of the board, with more than half the squares occupied by a knight on average. The integer PSO algorithm showed slightly higher mean values for board lengths up to 8 than binary PSO and did not manage to find any solutions for larger board lengths. While the binary PSO found solutions for all board lengths, its variance of utilised knights drastically increases for larger boards. The stochastic hill climbing algorithm displays competitive results for all board sizes compared to binary PSO, exhibiting similar mean values and reduced variation. The mean values for board lengths 6,7 and 10 are even smaller than those of binary PSO.

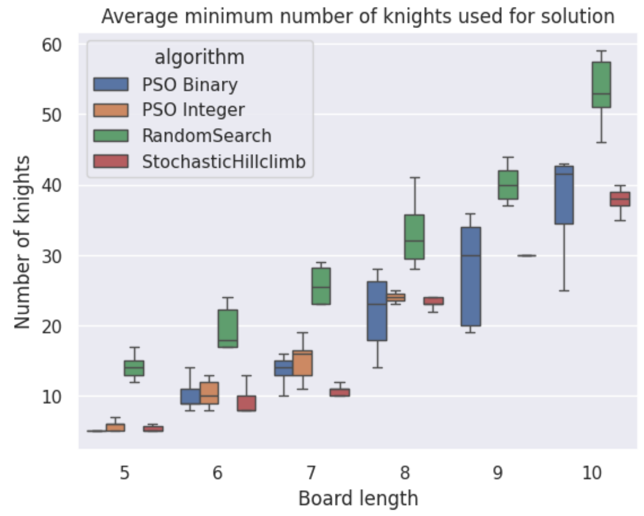

Fig. 3. Mean value of the minimum number of knights used in successful runs.

Looking at Figure 4, one can see that none of the random search solutions displayed in the previous plot were optimal. The integer PSO algorithm managed to find optimal solutions for board lengths 5 and 6, while stochastic hillclimb and binary PSO also obtained optimal solutions for board length 7. Comparing the number of fitness evaluations needed to obtain optimal solutions, integer PSO displays the worst performance and a significant increase in mean fitness evaluations going from board lengths 5 to 6. In contrast, the number of fitness evaluations increases more gradually for the binary PSO algorithm, albeit starting at a relatively high level of about 5000. The hill climbing algorithm needed only about 81 fitness evaluations to derive optimal solutions for board length 5 and 263 for board length 6, suggesting that it is the best choice for smaller board sizes out of the compared algorithms. However, it is also the algorithm most drastically affected by the increasing complexity of the search space, with the number of fitness evaluations growing

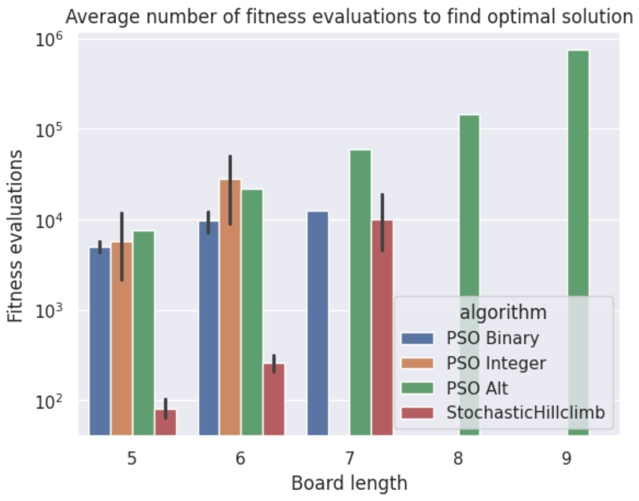

Fig. 4. Average number of fitness evaluations needed to find an optimal solution. Vertical lines represent the estimates' standard errors. PSO Alt contains the **minimum** number of fitness evaluations needed for the proposed binary PSO featured in [3]. The data was obtained by multiplying the respective number of iterations for a given board size, stated in [3], by the number of particles used (500).

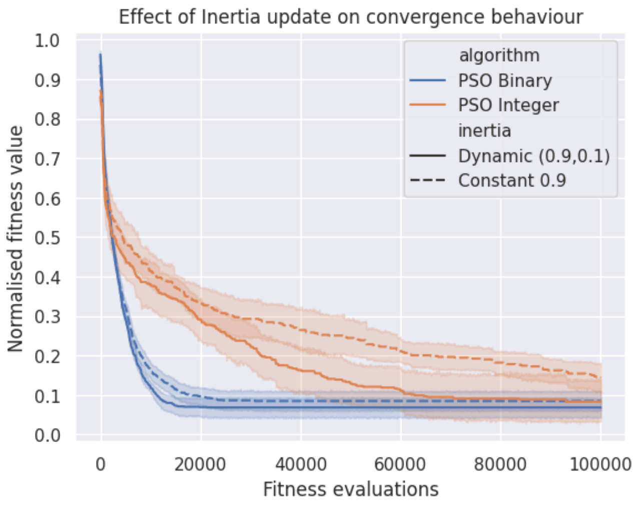

Fig. 5. Convergence behaviour of normalised fitness values, comparing binary and integer PSO, and constant versus dynamically updating inertia.

exponentially on the logarithmic scale used for the plot. This trend gives the impression that, for board sizes greater than 7, the binary PSO algorithm might perform better than hill climbing.

Comparing the results of the optimised binary PSO used in this project and the binary PSO proposed in [3], the former needed fewer fitness evaluations to obtain optimal solutions for board lengths up to 7. While the latter found optimal solutions for larger boards after more than 100,000 fitness function evaluations, the experiments conducted in this project were terminated before that. Thus, comparing the two algorithms for board lengths greater than 7 would require repeating the experiments with a larger termination threshold.

Figure 5 compares the convergence behaviour between constant and linearly decreasing inertia on the integer and binary PSO. The fitness values on the y-axis were normalised for each algorithm separately. The most noticeable difference between the binary and integer PSO is that the former converges much more rapidly, regardless of whether or not the inertia values are changing throughout the search process. However, both algorithms display improved convergence speeds and final fitness values for the linear decreasing inertia update strategy. This effect is most prominent for the integer PSO, with only minimal differences in convergence speed and fitness value regarding binary PSO.

Board\Alg.	**B-PSO**		**I-PSO**		**SHC**		**RS**	
	SR	**OR**	**SR**	**OR**	**SR**	**OR**	**SR**	**OR**
5 × 5	**100**	**95**	77	37	90	65	45	0
6 × 6	85	15	42	10	**100**	**55**	40	0
7 × 7	73	2	12	0	**100**	**40**	20	0
8 × 8	60	0	3	0	**100**	0	20	0
9 × 9	28	0	0	0	**100**	0	25	0
10 × 10	17	0	0	0	**100**	0	35	0

Table 1. Summary of the success rate (SR) and optimality rate (OR) for the tested algorithms with respect to the board size (see subsection 4.2). *B-PSO* is binary particle swarm optimisation, *I-PSO* is integer particle swarm optimisation, *SHC* is the stochastic hill climber and *RS* is random search. Numbers are given in percent. Bold numbers represent the best performances for the corresponding metric and board size.

Table 1 summarises the success and optimality rates of the conducted experiments, as defined in subsection 4.2. The columns specify the algorithms, while the rows show the board sizes. SHC is the only algorithm that consistently obtained solutions covering all the squares on the board. It also achieved the highest optimality rates for board lengths 6 and 7. While binary PSO was also able to find optimal solutions up to board lengths of 7, it was significantly less consistent, with only two per cent of runs conducted for this board size reaching an optimal solution.

4.4 Limitations and future work

As discussed in subsection 4.3, the integer PSO performed significantly worse than its binary counterpart. While for the binary encoding, each element of the location vector can take on two distinct values, resulting in 2^n possible positions, with n being the number of squares on the board, the integer case allows for a much larger number of possible values for each dimension. Thus, even though the number of dimensions was reduced, the number of positions in the search space increased due to the relatively high upper bound on the maximum number of knights needed to cover all squares, leading to $n^{n/2}$ combinations (see Equation 1). Looking at the actual number of nights needed to find an optimal solution [1], it becomes apparent that the utilised upper bound is much higher than necessary. For future work, it would, therefore, be interesting to observe the effect of further reducing the number of dimensions for the integer PSO to align with the numbers in [1]. While this approach would not be applicable to determine optimal solutions for previously unconsidered board sizes, it could prove helpful in improving or backing up empirical results for which a sufficiently small estimate of the maximum number of knights is already known.

Another limiting factor of this work is the number of fitness evaluation threshold. Although this termination criterion enabled a fair comparison with the non-population-based baseline algorithms, it drastically limited the number of iterations conducted for the PSO algorithms and the amount of particles used. Considering the results of [3], repeating the experiments with a higher number of particles and more iterations would allow for a better comparison between the PSO algorithms used in this work and the one proposed in [3] for larger board sizes. Furthermore, it would enable the investigation of the hypothesis made in subsection 4.3, stating that the PSO algorithm could demonstrate better performance than stochastic hill climbing for board lengths greater than 7 regarding the increase in the number of fitness evaluations displayed in Figure 4.

5 CONCLUSION

This project investigated the effect of using an integer encoding and inertia update strategy for a Particle Swarm Optimisation algorithm to solve the Knight Covering Problem, and it is based on the results from [3]. The improved PSO algorithm was tested with both binary and integer encoding, as well as various fitness and penalty functions. The results obtained from the conducted experiments show better performance for the stochastic hill climbing baseline algorithm than the PSO algorithms for small board sizes but suggest that PSO might work better for board lengths greater than 7. The integer encoding did not prove advantageous over its binary counterpart, which can be attributed to the unsuitably high upper bound of the maximum number of knights. Dynamically updating the inertia value led to improved convergence behaviour for both encodings. The improved binary PSO displayed better computational efficiency regarding the number of fitness evaluations than the algorithm proposed in [3]. However, more research is needed, utilising more particles and iterations, to extrapolate the findings to larger board sizes and come to a comprehensive conclusion about the effectiveness of the considered approaches.

REFERENCES

[1] Neil James Alexander Sloane. A006075 minimal number of knights needed to cover an n x n board. OEIS, June 2021. accessed: 27-04-2024.

[2] J. Kennedy and 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*, volume 5, pages 4104–4108 vol.5, 1997.

[3] N. Franken and A.P. Engelbrecht. Investigating binary pso parameter influence on the knights cover problem. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 282–289 Vol.1, 2005.

[4] J. C. Bansal, P. K. Singh, Mukesh Saraswat, Abhishek Verma, Shimpi Singh Jadon, and Ajith Abraham. Inertia weight strategies in particle swarm optimization. In *2011 Third World Congress on Nature and Biologically Inspired Computing*, pages 633–640, 2011.

[5] Mojtaba Ahmadieh Khanesar, Mohammad Teshnehlab, and Mahdi Aliyari Shoorehdeli. A novel binary particle swarm optimization. In *2007 Mediterranean Conference on Control & Automation*, pages 1–6, 2007.

[6] Noam D. Elkies and Richard P. Stanley. The mathematical knight. *The Mathematical Intelligencer*, 25(1):22–34, Dec 2003.

[7] Eric W. Weisstein. Knights problem. MathWorld, October 2008. accessed: 27-04-2024.

[8] J. C. Bansal, P. K. Singh, Mukesh Saraswat, Abhishek Verma, Shimpi Singh Jadon, and Ajith Abraham. Inertia weight strategies in particle swarm optimization. In *2011 Third World Congress on Nature and Biologically Inspired Computing*, pages 633–640, 2011.

[9] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[10] Yudong Zhang, Shuihua Wang, and Genlin Ji. A comprehensive survey on particle swarm optimization algorithm and its applications. *Mathematical Problems in Engineering*, 2015:931256, Oct 2015.

[11] Konstantinos Parsopoulos and Michael Vrahatis. *Particle Swarm Optimization Method for Constrained Optimization Problem*, volume 76, pages 214–220. IEEE, 01 2002.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.