Particle Swarm Optimization Algorithm for Multi-dimensional Knapsack Problem

Algorithm Overview

Below is the pseudo code for my Pariticle Swarm Optimization (PSO) algorithm to solve the Multi-dimensional Knapsack Problem (MKP). Compared to the original PSO, I add a feasibility repair function to improve algorithm performance.

```
randomly init particle positions and velocity (between [0,10])
while (iter < MAX_ITER && timeSpent < MAX_TIME){
    for each particle{
        for each item in the particle {
            update velocity
            update position by using sigmoid
        }
        evaluate_solution(particle)
        fitness_func(particle)
        update pbest, gbest

        update_best_solution(particle)
    }

    feasiblitiy_repair(best_sln)
}
```

Solution Encodings

Each particle represents one possible solution to the MKP problem, therefore, each particle is stored in a solution_struct pointer array. Each solution_struct contains problem_struct, objective and feasibility of the current solution, an array that stores the left capacity in all dimensions, and solution encoding vectors for each item's position and for each item's velocity.

```
struct solution_struct
{
    struct problem_struct *prob; //copy of the problem data
    float objective; // objective of the current solution
    int feasibility; // feasiblity of the solution (negative for infeasible)
    int *x;          // solution position vector
    int *v;          // solution velocity vector
    int *cap_left;   //capacity left in all dimensions
};
```

Fitness Function

In my algorithm, I use the fitness function introduced in [3] as the heuristic function. In Equation (1), the former term is the profit of all the items in the bag and the latter term is the penalty coefficient multiply the sum of exceeded sizes in all dimensions (calculated by Equation (2)). The penalty coefficient `P` is set to 5000.

$$g(\vec{x}) = \underbrace{\sum_{j=1}^{n} p_j x_j}_{\text{profit}} - \underbrace{P \sum_{i=1}^{m} poslin\left(\sum_{j=1}^{n} w_{ij} x_j - M_i\right)}_{\text{penalty}}$$

Equation (1)

$$poslin(x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$

Equation (2)

Methods for Diversifications and Intensification

Feasibility Repair Function According to [6], penalty function (fitness function) have certain drawbacks.

1. **Parameter Setting**: The penalty parameter could greatly affect the results. If it is a relatively large number (e.g. 6000), it is likely to penalize particles too harshly, thus leading to local optima. Also, it is set too low, it may not function well to find the best feasible solutions.
2. **Problem Specific Information**: As penalty function does not take problem specific information into account, it might limit the generalizability of the algorithm.

By introducing feasibility repair function into the original PSO algorithm, it allows particles to search out of the feasible space and then "drag" them back to the feasible space. Such action could ensure a more comprehensive searching in the feasible space and avoid local optima in an efficient way. The feasibility repair function is greedy-based, which take out items with least ratio(profit / sum of size in all dimension) until the solution is feasible.

Particle Number: The population size is a critical factor for PSO algorithm. If the particle size is too small, the population is inclined to be trapped in a limited space, thus leading to the local optima. Therefore, A large population size is set in my PSO algorithm, to add more randomness in the initial state and carry on a more exhaustive search.

Parameter Settings

Optimal PSO parameters

w	c1	c2	Particle Number	Penalty Coefficient	V_MAX
1	3	2	2000	5000	25

When tunning the parameters, I first settle down the particle numbers for my PSO. I have tried to set particle number to 250/500/1000/1500/2000, it turns out that population with small particle number takes longer time to find the optimal solution, while population with large particle number converge to a near-global-optimal in a short time period. Also, populations with large particle number could obtain a greater solution objective in my case. Therefore, to improve the tunning efficiency and overall algorithm performance, I set the particle number to 2000.

For max speed of each item, I set the max value to 25. In my experiment, I use V_MAX = 10 as the starting point. I have tried different V_MAX values such as 10/15/20/25, which make negligible differences to the final results. The reason for this might be that the problem is a binary problem, which the state of item (1 in the bag, 0 vice versa) is determined by a sigmoid transformation function with item velocity as the input. Mathematical calculation shows that when V_MAX=5, the output of sigmoid function is already 0.997, which indicates that the item is highly likely to be placed into the bag. The final value for V_MAX is used to get a wider speed range and differentiate each items' velocity.

The optimal value for cognitive learning factor is 3. Although it exceeds the recommended range in paper [3], empirical study shows that it obtains the best results among 1.5/2.2/2.4/2.6/2.8/3.

The rest parameters remain unchanged.

Results

Generally speaking, my PSO achieves gap with the best-known objective below 1% for 84% of the given problem instances, between 1% and 2% for 14% of the problems and between 2% and 5% for the rest of the problems.

Average Gap for Each File

	mknapcb1	mknapcb2	mknapcb3	mknapcb4	mknapcb5	mknapcb6	mknapcb7	mknapcb8
GAP (%)	0.44	0.4	0.3	0.67	0.58	0.71	0.9	0.88

The above table presents the average gap for each file that contains 30 problem instances. All average gap is below 1%.

Selected Results Showcase

Problem Name	GAP (%)	Problem Name	Gap (%)
5.250-28	0.01	5.500-24	0.14
5.250-29	0.03	5.500-25	0.14
10.100-25	0.36	10.250-16	0.6
10.100-26	0.21	10.250-17	0.31
10.500-03	1.01	30.250-25	0.34
10.500-08	1.56	30.250-28	0.44

One key observation from the experiment results is that my PSO algorithm performs well for problems with relatively small item number and item dimensions while it may get certain anomaly for problems with large item number and dimensions.

PSO Derivatives Try-out

In this section, I will introduce two PSO derivatives that I have tried to solve the MKP. A co-evolution binary particle swarm optimization with a multiple inertia weight strategy (CBPSO-MIWS) [1] and Local Optima Avoidable Particle Swarm Optimization (LOAPSO) [2] are similar PSO derivatives to avoid local optima in the original PSO algorithm. CBPSO-MIWS separate the whole initial population into multiple sub-populations and apply a time-changing inertia weight to

ensure the diversification in the search space, while LOAPSO proposes an "avoidance rate" which modify the traditional PSO velocity updating equation and separate the population into two sub-populations with one search for the previous best and social best positions, and the other one move away from the best positions in the first group. I have implemented both algorithms for the MKP, since they share similar methodology to break down the whole population into sub-populations. However, due to the limited time, I have not tunned these two algorithms to obtain results as good as the submitted one.

Reflection

1. **Literature Study**: To fulfill the gap of my knowledge, I not only read the paper recommended in the lecture [3], but also locate several novel PSO derivatives [1, 2, 4, 5, 6] to grasp the main idea of the state-of-the-art techniques towards the MKP. [3] uses a CUDA accelerated PSO to solve the MKP, which contains detailed explanations of the original PSO algorithm and its discrete version for binary problems. With [3] as a starting point, I search further for other novel derivatives. [4] incorporates Tabu Search into PSO to avoid local optima, [6] proposes a repair-operator-based PSO instead of using fitness function constrain the particles in the feasible space while [5] introduces a time-changing inertia weight to strike a balance between exploitation and exploration. I believe my literature study help me gain a better understanding of the fields, which is also beneficial to my modifications based on the original PSO algorithm.
2. **Algorithm Performance** :

   Advantages: Similar to orginal PSO, my PSO algorithm share similar advantages, such as easiness of implementation, quickness to acquire solutions.

   Disadvantages: Currently, my PSO algorithm still has trouble in striking a balance between exploration and exploitation, which might lead to premature convergence. Also, the relatively large population size could result in the creation/formation of particles with similar item velocities and positions, which degrades searching efficiency and increases the possibility of being trapped in local optima. Finally, the performance of my PSO may not further improve as the number of iteration is increasing (the best solution remains unchanged as the iteration proceeds).
3. **Results**:

   As the experimental result shows, my PSO obtains a relatively high performance for the given dataset. However, it still has some room for improvement, especially for problem with large item number and dimension number. For problem with multiple dimensions (e.g. 30), the performance gap of my PSO could reach 2+%. The underlying reason might be that the current feasibility repair function design limit the solutions. Current feasibility repair function is greedy-based, which take out items with least ratio(profit / sum of size in all dimension) until the solution is feasible. Such repair function might not be the optimal one. In the future, it is essential to design a more suitable repair function for MKP.

References

[1] Too, Jingwei & Abdullah, Abdul Rahim & Mohd Saad, Norhashimah. (2019). A New Co-Evolution Binary Particle Swarm Optimization with Multiple Inertia Weight Strategy for Feature Selection. Informatics. 6. 21. 10.3390/informatics6020021.

[2] Salehizadeh, Sma & Yadmellat, Peyman & Menhaj, M.B.. (2009). Local Optima Avoidable Particle Swarm Optimization. 2009 IEEE Swarm Intelligence Symposium, SIS 2009 - Proceedings. 16 - 21. 10.1109/SIS.2009.4937839.

[3] Drahoslav Zan and Jiri Jaros. 2014. Solving the Multidimensional Knapsack Problem using a CUDA accelerated PSO. 2014 IEEE Congress on Evolutionary Computation (CEC), July 6-11, 2014, Beijing, China.

[4] Ktari, Raida & Chabchoub, Habib. (2013). Essential Particle Swarm Optimization queen with Tabu Search for MKP resolution. Computing. 95. 10.1007/s00607-013-0316-2.

[5] Mingchang Chih, Chin-Jung Lin, Maw-Sheng Chern, Tsung-Yin Ou. Particle swarm optimization with time-varying acceleration coefficients for themultidimensional knapsack problem. Applied Mathematical Modelling, Volume 38, Issue 4, 2014, Pages 1338-1350. DOI:doi.org/10.1016/j.apm.2013.08.009.

[6] Min Kong and Peng Tian. 2006. Apply the particle swarm optimization to the multidimensional knapsack problem. In Proceedings of the 8th international conference on Artificial Intelligence and Soft Computing (ICAISC'06). Springer-Verlag, Berlin, Heidelberg, 1140–1149. DOI:doi.org/10.1007/11785231_119.