Implementation of the Particle Swarm Optimization algorithm for the multi-dimensional knapsack problem

Coursework Artificial Intelligence 2

JeGa
Applied Computer Science,
HTWG Konstanz

June 15, 2014

Abstract

Particle Swarm Optimization (PSO) is a metaheuristic search algorithm which uses a swarm of solutions/particles to iteratively optimize a problem. PSO was intentionally designed by Kennedy and Eberhart [1] for continues optimization problems. Later, several papers were published that cover PSO for discrete search spaces. This document describes the implementation of the discrete PSO algorithm to solve the multi-dimensional knapsack problem.

Contents

1 Introduction

Because PSO was designed for continues problems, some changes need to be done to use it for discrete problems. This document covers shortly the PSO for continues problems (2) and the changes to be made to use PSO for discrete problems (3). The next chapter (4) describes multidimensional the knapsack problem and the adoption of the PSO. Implementation details are outlined in chapter 5. In chapter 6 the performance of the algorithm and implementation is analysed. The last chapters (7) and (8) describe the application and the improvements that can be made.

2 PSO for continues problems

Particle Swarm Optimization is a swarm algorithm based on the flocking behaviour of birds were a swarm of birds searches for food. Every bird (particle) uses its own knowledge and the swarms knowledge to direct its search towards the food. In PSO, a particle represents a possible solution. It consist of a position x_i and a velocity v_i. Each particle saves its best position (particle best - pBest). Additionally the best position of all particles in the swarm is saved (global best - gbest). Each particle updates its position based on the current velocity, the particles best position and the global best position. If the problem is multidimensional, based on each dimension d of the solution vector, the new velocity/position must be calculated.

$v_i = \omega * v_i + c_1 * rand_1 * (x_pbest - x_i) + c_2 * rand_2 * (x_gbest - x_i)$

$x_i = x_i + v_i$

ω is the inertia weight which specify how strong the current velocity contributes to the new velocity. c_1 and c_2 are constants which weight the personal and the swarms evidence. $rand_1$ and $rand_2$ are random values from $[0, 1]$. Additionally, the new velocity v_i is limited by a constant v_ax.

3 PSO for discrete binary problems

To use PSO for discrete binary problems, some changes need to be made. Unlike in continues space, a multidimensional solution vector (position) consist now of discrete binary values (0/1). The velocity of a particle is expressed as a vector of probabilities, that the elements at each dimension have the value 1 (rate of change). This means, a particle flies fast if all the binary values change and slow if nothing changes. However, the velocity formula remains the same, but the calculation of the new position changes slightly. At first, to limit the velocity value to a valid probability, a logistic transformation is made.

$S(v_i) = 1.0/(1.0 + e^{(-v_i)})$;

The output of formula is a velocity limited with the sigmoid function to $[0.0, 1.0]$. To calculate the new position, the following formula is used.

$todo$

4 Multidimensional knapsack problem

Because the multidimensional (multi-constraint) knapsack problem is already described in detail in a lot of papers, it is only briefly described here. To sum up, a number of elements need to be packed into a knapsack. Each element has a profit value, and several constraint values. The aim is, to maximize the profit (fitness value) while not violating any constraint. A solution is described as a binary vector with the dimension as the number of elements to pack. 1 means the element is in the knapsack, 0 means it is not in the knapsack. An important addition to the traditional discrete PSO are the constraint values. Special care has to be taken to identify invalid solutions.

5 Implementation

This chapter describes the implementation in detail and shows sum extracts from the source code.

5.1 Algorithm

At first, the particles are initialized with random positions and random velocities. After that, the profit is of the particular solution is calculated and saved as gBest value. The highest profit is saved as the swarms gBest value.

Listing 1: ”Solver.cpp”

```
for (auto &i : swarm.getParticles()) {
            Solution position = getRandomSolution();
            Velocity velocity = getRandomVelocity();

            i.setPosition(position);
            i.setVelocity(velocity);

            // Fitness value / Profit of the solution/position.
            int profit = calculateProfit(position);

            // ... Set gBest and pBest
}
```

After initialization, the method `findSolution()` is called, to iterate through all particles of the swarm. For each dimension of the particles the formula 2 and 2 are applied.

Listing 2: ”Solver.cpp”

```
double newVelocityD = parameters.getInertiaWeight() *
    currentVelocityD +
                      parameters.getConstant1() *
                          randomParticleNumber * (pBestD -
                          currentPositionD) +
                      parameters.getConstant2() * randomGlobalNumber
                           * (gBestD - currentPositionD);
```

```
if (newVelocityD > parameters.getVMax())
        newVelocityD = parameters.getVMax();

/* K. and E. */
int newPositionD = updateStrategy_standard(currentPositionD,
    newVelocityD);
```

At last, the profit and the penalty values are calculated. With this information the pBest and gBest values can be updated.

Listing 3: ”Solver.cpp”

```
int pBestTmp = calculateProfit(i.getPosition());
pBestTmp -= calculatePenalty(i.getPosition(), pBestTmp);

// Update pBest and gBest position/solution
if (pBestTmp > i.getBestValue()) {
    i.setBestPositionAndValue(i.getPosition(), pBestTmp);
        if (pBestTmp > swarm.getBestValue())
            swarm.setBestPositionAndValue(i.getPosition(), pBestTmp)
                ;
}
```

5.2 Update strategies

There are several strategies available, to update the particles position based on the velocity. The current used startegie in the implementation is the original strategie from Kennedy and Eberhart. A novel based update strategy and the improved update strategie from Qi Shen are also implemented.

5.3 Parameter

The available parameters are explained in the following table.

Inertia weight (ω)	Specifies the impact of the current velocity to the new velocity
Constant 1 (c_1)	Weights the evidence of the particle
Constant 2 (c_2)	Weights the evidence of the swarm
Maximal velocity (V_{max})	Limits the velocity

The best found configuration is displayed in image TODO.

6 Tests

7 Application

8 Improvements

Local pBest.

References

[1] Kennedy, J.; Eberhart, R. *Particle Swarm Optimization*. Proceedings of IEEE International Conference on Neural Networks IV. 1995

[2] Ling Wang, Xiuting Wang, Jingqi Fu, Lanlan Zhe. *A Novel Probability Binary Particle Swarm Optimization Algorithm and Its Application*. JOURNAL OF SOFTWARE, VOL. 3, NO. 9, DECEMBER 2008

[3] Anne L. Oken. *PENALTY FUNCTIONS AND THE KNAPSACK PROBLEM*. Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on.

[4] James Kennedy, Russell C. Eberhart. *A DISCRETE BINARY VERSION OF THE PARTICLE SWARM ALGORITHM*. Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on.