Experiments in Developing Discrete Particle Swarm Optimisers

Martin Gate

April 24, 2019

1 Introduction

1.1 Continuous Case model

Particle Swarm Optimisers (PSO) introduced in [KE95] have been developed to optimise problems with continuous tuning parameters. The PSO consists of a finite collection (called a swarm) of particles with current state (often called *Parameters*) x_t^i personal best state p_t^i and velocity V_t^i; where i is the particle index and t is the iteration index. These three properties of each particle are updated at every iteration using

$$V_{t+1}^i = \mu\left(x_t^i, V_t^i, N_t^i\right) \tag{1.1}$$

$$x_{t+1}^i = \xi\left(x_t^i, V_t^i\right) \tag{1.2}$$

$$p_{t+1}^i = \begin{cases} p_t^i & \text{if } x_{t+1}^i \notin \Omega \text{ or } F(p_t^i) < F(x_{t+1}^i) \\ x_t^i & \text{otherwise} \end{cases} \tag{1.3}$$

where $F(x)$ is the cost of the state x; N_t^i is the collection of personal best states used by the ith particle to update its velocity from iteration t to $t+1$: Ω is a feasible subset of states usually characterised as satisfying a collection of constraints .

The PSO has several variants and the above description is a general summery where for continuous parameters the state is a d dimensional vector space of real numbers $\mathbb{R}^d$. Typically the velocity is made up from random multiples of the vector diference between x_t^i personal bests and members ofN_t^i giving the swarms directions to go to look for improved $F(x)$ values. Also μ and ξ are not true functions since they depend on hidden random variables that are crucial for ensuring the ability to find a low cost near optimal personal best after sufficient iteration, but are otherwise processes that depend on the variables given as their arguments.

1.2 Combinatorial PSO

The current work looks at combinatorial optimisation problems (COP) using discrete tuning parameters in the form of binary strings (BS) for particle state

with big integer cost functions. Such a PSO will be called a Set PSO (abbreviated to SPSO) to emphasise that it uses sets to represent the parameters of the SPSO.

The restriction to big integer is strictly not required since the cost function is used only to compare the cost between solutions so at a later stage one may extend to floating point or rational cost functions with extended resolution. Most problems are manageable if the number of possible values is kept reasonably small by restricting the range of integers.Big integers are used to include some of the difficult cases although the chance of finding global best solutions to these problems is minimal. The algorithms are expected to however produce good useful approximations to these hard problems. for the moment we adopt the principle that real number cost functions can be converted into an approximate big integer by multiplying by a suitable large constant to achieve the desired resolution after rounding to the nearest integer. I would like to suggest that main area of application *is machine learning where one is interested in good approximate minimusation solutions of difficult problems*.

COP problems such as the Travelling Sales Person(TSP) have constraints defining Ω that must be satisfied if to be of any use so one must continuously ensure the candidate solution is in Ω before evaluating its cost. Often feasible points Ω do not form an open subset of the space of states so a raw update of x_t^i will not give a feasible state most of the time. To cope with this ξ computes a raw update $\check{x}_{t+1}^i$ which it then attempts to convert into a feasible one before being returned as a update. Sometimes this may not be possible and the returned x_t^i is not in Ω. As indicated in the update equation this results in no change to the personal best thus ensuring that personal bests are updated to feasible states.

The link between COP and SPSO is forged by using binary strings to represent candidate algorithms which are obtained by decoding the string. I use the term algorithm to represent any method for finding approximate solutions to a COP problem including just providing an approximate solution with no use of an algorithm. I think decoding to an algorithm rather than just a solution gives the PSO significantly more expressive power to find approximate solutions. The structure of candidate solutions may not be the same as binary strings so several binary strings may map to the same solution. The resulting solution is then evaluated to give its cost indicating how well the algorithm fits the data in the form of a (possibly large) integer; this could include some cost representing the complexity of the used algorithm; lower values being preferred to higher values. This procedure gives a mapping from a binary string to a integer as a cost of the binary string. The SPSO is used to learn a low cost global solution by using a swarm of proposed solutions to iteratively search for a low cost one.

For the moment we restrict the SPSO state to be at most N bits. Each binary string x can thus be regarded as a subset of integers in the range $1 \cdots N$ where $j \in x$ when $x[j] = 1$. To this extent each non zero bit represents the inclusion of some element with a feature and as such encodings that favour this interpretation is expected to do better using the following algorithms.

1.3 Reading Across from the Continuous Case

The notation for the continuous case can be now used where x_t^i is just a subset rather than a vector. The velocity is interpreted as a vector of probabilities where the jth component $V_t^i[j]$ is just the probability that $x_t^i[j]$ will flip during the update going from 0 to 1 or *vice versa*.

In the continuous case the raw update is given by adding the velocity component to the current state. For the SPSO case by analogy we use flipping to give a raw update from the velocity.

let *rand* be a random number generator that produces a number in the range $0 \leq rand < 1$ for each use of it then put for each component

$$(V_{t+1}^i[j]), \breve{x}_{t+1}^i[j]) \leftarrow \begin{cases} (0, \neg x_t^i[j]) & \text{if } V_{t+1}^i[j] > rand \\ (V_{t+1}^i[j], x_t^i[j]) & \text{otherwise} \end{cases} \tag{1.4}$$

where $\neg$ is the logical negation and $\leftarrow$ is the assignment of the left side to the right side of the arrow.

At this stage the raw state $\breve{x}_{t+1}^i$ may not satisfy the constraint so to keep things general a function `ToConstraint()`[1] supplied by the cost-function interface is applied to attempt to make the raw state satisfy the constraint based on the raw state as a starting hint and the old state,x_t^i to facilitate this attempt.If it fails then revert back to the old state.Thus:

$$\breve{x}_{t+1}^i \leftarrow \text{ToConstraint}(x_t^i, \breve{x}_{t+1}^i) \tag{1.5}$$

$$x_{t+1}^i \leftarrow \begin{cases} \breve{x}_{t+1}^i & \text{if } \breve{x}_{t+1}^i \in \Omega \\ x_t^i & \text{otherwise} \end{cases} \tag{1.6}$$

this gives an update that satisfies the constraints, Ω where the initalisation of the state x_0^i is chosen to ensure it also satisfies the constraint. this is done by random selection untill the application of `ToConstraint()` gives something that satisfies the constraint.

References

[KE95] J. Kennedy and R. Eberhart. Particle swarm optimization. *International conference on neural net- works, IEEE*, 4(3):1942–1948, July 1995.

[1] in the code this function has a diferent interface to minimise the API but is used to acheive the same thing.