Specifying Ant System in ESDL

Steve Dower*

Swinburne University of Technology
Melbourne, Australia
January 10, 2011

Abstract

Evolutionary Systems Definition Language (ESDL) is a domain-specific language for search algorithms based on iterative improvements to a solution population. Ant System (AS) is a swarm algorithm that uses a communication scheme based on the pheromone trail mechanism of some ant species. This report describes AS using ESDL and validates the performance against earlier work.

1 Introduction

1.1 ESDL

Evolutionary System Definition Language (ESDL) is a domain-specific language for describing the flow of evolutionary algorithms [1]. It defines a system as *groups* of *individuals* and the process followed to create new groups from existing ones. Groups are created by selecting and modifying individuals from existing groups or from *generators*.

ESDL does not specify the implementation or behaviour of specific operations: it requires an underlying framework to provide the functionality. For this report, `esec`[1] is the supporting framework.

1.2 Ant System

Ant System (AS) was originally presented as "Ant-cycle", one of the first three ant-inspired algorithms [2]. The general algorithm is now known as Ant Colony Optimisation (ACO), with the common feature of ACO algorithms being the use of environmental stigmergy (pheromone trails, as used by some species of ants) to constructively generate potential solutions [3]. Excerpt 1 shows the basic ACO algorithm. Specific algorithms vary in construction or pheromone update schemes.

Excerpt 1: The general Ant Colony Optimization algorithm from [3]

```
Set parameters, initialize pheromone trails
while termination condition not met do
    ConstructAntSolutions
    ApplyLocalSearch (optional)
    UpdatePheromones
end while
```

One of the first applications of ACO was the Travelling Salesman Problem (TSP), which is still commonly used as a benchmark. In the TSP the goal is to find the shortest path that visits each

*Contact via http://stevedower.id.au/

[1]Available online at http://code.google.com/p/esec. The plugin and configuration files used in this report are also available from here.

member of a set of cities without visiting any city more than once. ACO applied to TSP uses each simulated ant to select a path at random, biased towards shorter segments and those with higher levels of simulated pheromone. In AS, the amount of pheromone added to each segment is inversely proportional to the final length of all tours that used it, encouraging future ants to re-use segments that have previously contributed to shorter paths. Over a number of iterations, positive reinforcement results in all ants converging to the same path. With a suitable set of AS parameters, this path will represent one of the better solutions. Unsuitable AS parameters create an excessively strong pheromone effect, resulting in convergence to the first path found, or a strong bias towards a nearest-neighbour solution, making the algorithm overly greedy.

2 System Definition

ESDL can be used to define AS and apply it to a TSP instance, as shown in Listing 1. Several elements external to the ESDL definition are required: `create_pheromone_map` (a function to initialise a pheromone map object), `build_tours` (a generator for creating candidate solutions) and `cost_map` (the matrix of costs between connected nodes). An evaluator to calculate the length of a given tour is also provided externally. The transient group created at lines 4–6, `ants`, is used to update `pheromone_map` once per iteration. Values for `alpha`, `beta`, `rho`, `Q` and `colony_size` are parameterised, allowing multiple experiments to be conducted with different parameter sets but the same definition.

Listing 1: ESDL definition of the AS algorithm applied to a TSP

```
 1 pheromone_map = create_pheromone_map(initial=(Q))
 2 
 3 BEGIN GENERATION
 4     FROM build_tours(cost_map=cost_map, cost_power=(beta), \
 5          pheromone_map=pheromone_map, pheromone_power=(alpha)) \
 6         SELECT (colony_size) ants
 7     YIELD ants
 8 
 9     pheromone_map.update(source=ants, persistence=(1.0-rho), strength=(Q))
10 END GENERATION
```

The pheromone map is an automatically expanding array with a helper method to update each value (described in Listing 2). Allowing the pheromone map to expand removes the need to obtain the size of the landscape within the system definition.

Listing 2: Pseudocode definition for `update`

```
function update(source, persistence, strength):
    for each element $\tau$ of the pheromone map:
        $\tau = \tau \times$ persistence

    for each individual in source:
        delta = strength $\div$ (length of the individual's tour)

        for each link $i \rightarrow j$ in the individual:
            $\tau_{ij} = \tau_{ij} +$ delta
```

Behaviour for the `build_tours` generator is specified in Listing 3. It assumes a fully connected graph with disconnected nodes simulated by assigning a prohibitively high cost to the edge between them.

Listing 3: Pseudocode definition for `build_tours`

```
function build_tours(cost_map, cost_power, pheromone_map, pheromone_power):
    while individual is requested:
        $i = 0$
        options = { all nodes except $i$ }
        tour = ( $i$ )

        while options is not empty:
            $j$ = node selected with probability from $\{p_{ij} : j \in \text{options}\}$
            append $j$ to tour
            remove $j$ from options
            $i = j$

        yield tour
```

The definition of p_{ij} is stated as

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{l \in \text{options}} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}} \tag{1}$$

where τ_{ij} and η_{ij} are the pheromone level and cost respectively between nodes i and j, options is the set of unvisited nodes and α and β are the scaling powers `pheromone_power` and `cost_power`, respectively.

3 Validation

Results reported by Dorigo et al. [2] covered sixteen parameter variations for AS. For validation of the ESDL implementation, we reproduced these sixteen experiments. The TSP problem selected was the Oliver30 graph (fully specified in Appendix A). Each experiment was repeated 10 times for 5000 generations; the results compared are the mean of the shortest path length found. In each case, the number of ants used was 30, to match the number of cities in Oliver30, and distances were not rounded to integers. `esec` was used as the underlying framework; listings 2 and 3 and (1) were implemented in Python (full code is given in Appendix B).

Three parameters varied were `alpha`, `beta` and `rho` (as specified in Listing 1). The value for `Q` was fixed at 100. A full listing of parameters, the original results and the reproduced results are shown in Table 1.

Since ten runs are not sufficient to determine consistent averages on a largely stochastic process, some discrepancies are expected.[2] Figure 1 shows that the ESDL-based implementation produces similar results to the original publication, but more importantly is sensitive to parameter variations in a manner consistent with [2].

[2]While it would have been possible to conduct more experiments, the decision was made that a faithful reproduction of the experiment was preferable and suitable for the validation work.

Table 1: Parameter values used and mean shortest path length comparison

`alpha` (α)	`beta` (β)	`rho` (ρ)	Results from [2]	ESDL Results
0	1	0.7	651.27	664.80
0.5	1	0.7	533.49	471.40
1	1	0.7	427.44	443.35
2	1	0.7	456.11	476.77
1	**0**	0.7	848.31	837.65
1	**0.5**	0.7	452.62	446.01
1	**1**	0.7	427.44	443.55
1	**2**	0.7	424.63	441.84
1	**5**	0.7	424.25	437.03
1	**10**	0.7	428.35	433.27
1	**20**	0.7	438.88	434.87
1	1	**0.3**	427.85	440.98
1	1	**0.5**	426.86	442.05
1	1	**0.7**	427.44	440.61
1	1	**0.9**	428.28	439.98

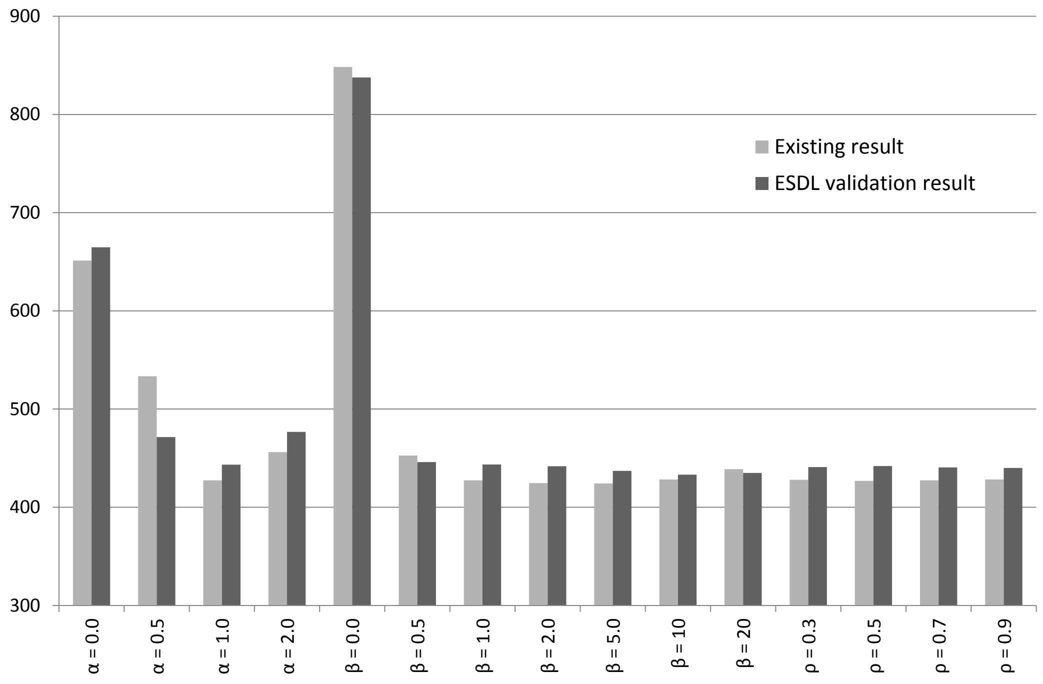

Figure 1: Average shortest path for each parameter set

4 Summary

This report provides a description of the Ant System algorithm using ESDL. The classic algorithm is shown concisely in ESDL with specific elements shown as pseudocode or equations as appropriate.

The ESDL description was used with `esec` to produce a working implementation, which was validated against a TSP benchmark problem. Results show that the ESDL implementation is consistent with existing published results.

References

[1] S. Dower and C. Woodward, "Evolutionary System Definition Language," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1, 2010. 1

[2] M. Dorigo, V. Maniezzo, and A. Colorni, "Positive feedback as a search strategy," Polytechnic University of Milano, Tech. Rep., 1991. 1, 3, 4

[3] M. Dorigo, M. Birattari, and T. Stützle, "Ant Colony Optimization," IRIDIA, Tech. Rep. TR/IRIDIA/2006-023, 2006. 1

[4] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the travelling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, 1987, pp. 224–230. 6

A Oliver30 TSP

The Oliver30 TSP is shown in Table 2 as a list of coordinates. Figure 2 shows a coordinate map of city locations. The values were obtained from [4].

Table 2: City locations for the Oliver30 TSP

Coordinates	
54	67
54	62
37	84
41	94
2	99
7	64
25	62
22	60
18	54
4	50
13	40
18	40
24	42
25	38
44	35
41	26
45	21
58	35
62	32
82	7
91	38
83	46
71	44
64	60
68	58
83	69
87	76
74	78
71	71
58	69

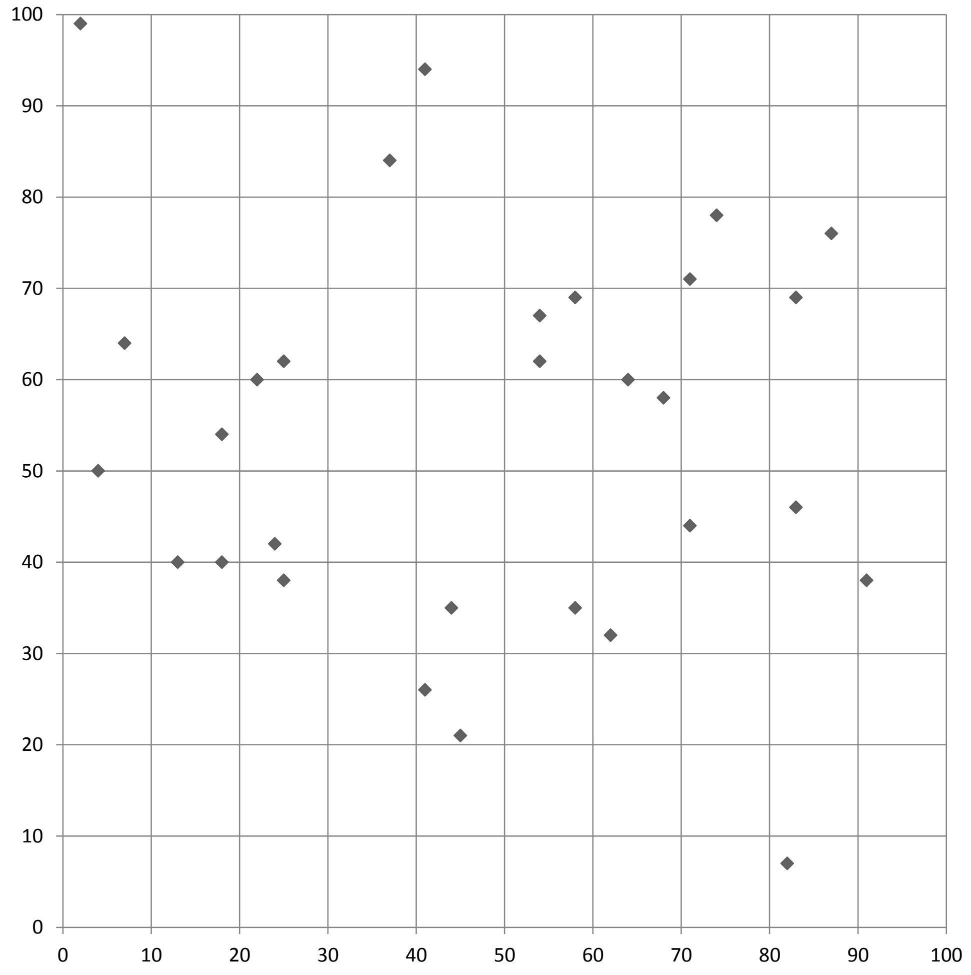

Figure 2: City locations for the Oliver30 TSP

B Python Implementation

Listing 4 shows the configuration used for conducting the experiments in this report. Listing 5 shows the implementation of `build_tours` using the `TourIndividual` shown in Listing 6, which is based on `esec`'s `IntegerIndividual`. Listing 7 implements a `PheremoneMap` class, including the `update` method from Listing 2. The implementation of (1) is shown in Listing 8.

(The code shown has been simplified and comments removed from the full code available online at `http://code.google.com/p/esec/`.)

Listing 4: The `esec` configuration used

```
from plugins.ACO import *

city_graph = tsp.Landscape(cost_map="cfgs/Oliver30.csv")

config = {
    'system': {
        'alpha': 1.0,
        'beta': 1.0,
        'rho': 0.7,
        'Q': 100,
        'colony_size': 30,
        'cost_map': city_graph.cost_map,
        'city_graph': city_graph,
        'definition': r'''
  pheromone_map = create_pheromone_map(initial=(Q))

  BEGIN GENERATION
    FROM build_tours(cost_map=cost_map, cost_power=(beta), \
                     pheromone_map=pheromone_map, pheromone_power=(alpha)) \
        SELECT (colony_size) ants

    EVAL ants USING city_graph
    YIELD ants

    pheromone_map.update(source=ants, persistence=(1-rho), strength=(Q))
  END GENERATION
        ''',
        'create_pheromone_map': pheromone.PheromoneMap,
    },
    'monitor': {
        'report': 'brief+local_header+local_int+local_unique',
        'summary': 'status+brief+best_phenome',
        'limits': {
            'generations': 5000,
        },
        'primary': 'ants',
    },
}
```

Listing 5: Python implementation of `build_tours`, based on Listing 3

```
def build_tours(cost_map, cost_power=2.0, \
                pheromone_map=None, pheromone_power=2.0):
    irand = rand.randrange
    frand = rand.random

    length = max(cost_map)[0] + 1

    while True:
        current_city = 0

        # Remaining options
        options = set(xrange(length))
        options.discard(current_city)
        genes = [ current_city ]

        while options:
            prob_list = init_fitness_wheel(current_city, options, \
                                           cost_map, cost_power, \
                                           pheromone_map, pheromone_power)
            total = sum(i[1] for i in prob_list)
            selection = frand() * total

            next_city = prob_list[0][0]
            i = 0
            while selection > 0.0:
                selection -= prob_list[i][1]
                i += 1
            i -= 1
            if 0 < i < len(prob_list):
                next_city = prob_list[i][0]
            else:
                # Greedy selection as a fallback
                next_city = prob_list[0][0]

            current_city = next_city
            genes.append(current_city)
            options.discard(current_city)

        yield TourIndividual(genes, parent=self)
```

Listing 6: Python implementation of `TourIndividual`, required by Listing 5.

```
class TourIndividual(IntegerIndividual):
    @property
    def phenome(self):
        p = [ ]
        current = self.genome[0]
        for g in self.genome[1:]:
            p.append((current, g))
            current = g
        p.append((current, self.genome[0]))
        return p

    @property
    def phenome_string(self):
        return ' -> '.join(str(i) for i in self.genome)
```

Listing 7: Python implementation of a pheromone map

```
class PheromoneMap(object):
    def __init__(self, initial=0.0):
        self.initial = initial
        self._pheromone = { }

    def __getitem__(self, key):
        return self._pheromone.get(key, self.initial)

    def update(self, source, strength, persistence):
        pheromone = self._pheromone

        for key in pheromone.keys():
            pheromone[key] *= persistence

        # decay the initial value as well
        self.initial *= persistence

        for indiv in source:
            delta = strength / float(indiv.fitness.values[0])

            for p in indiv.phenome:
                pheromone[p] = self[p] + delta
```

Listing 8: Python implementation of `init_fitness_wheel` based on (1)

```
def init_fitness_wheel(current_city, options, \
                       cost_map, cost_power, \
                       pheromone_map, pheromone_power):
    cost_list = [(i, cost_map[(current_city, i)]) for i in options]
    pher_list = [(i, pheromone_map[(current_city, i)]) for i in options]

    prob_list = [(i, \
        (c ** -cost_power if c else 1) * \
        (p ** pheromone_power if p else 1)) \
        for (i, c), (i2, p) in zip(cost_list, pher_list) if i == i2]

    return sorted(prob_list, key=lambda i: i[1], reverse=True)
```