A Simple Evolved Controller for the Physical Travelling Salesperson Problem

Robert M. MacCallum

Stockholm Bioinformatics Center
Stockholm University
106 91 Stockholm
Sweden
maccallr@sbc.su.se

1 Summary

A controller function was evolved using genetic programming in order to steer an agent around a map according to the rules of the Physical Travelling Salesperson Problem devised by Simon Lucas as part of the competition section of GECCO 2005. The inputs to this function are

- current position (vector)
- current velocity (vector)
- positions of N non-visited cities ($2 \times N$ matrix, arbitrary order)

The output of the function is a single vector, which is then quantised into one of the 5 allowed acceleration vectors used to control the salesperson. No external software or algorithms for the Travelling Salesman Problem (TSP) were used; the idea is that the controller function will make its own strategic decisions based on the locations of all unvisited cities. The fitness function rewards the number of cities visited, penalises the number of time steps taken (capped at 1400) as follows:

$$\text{fitness} = \text{nVisited} - \frac{\text{nVectors}}{2000} - \frac{4-d}{20}$$

where d is the number of *different* non-zero directions used in the solution. The total number of non-zero acceleration forces (nForces) is not used.

The function was trained on random maps (the same map was used for ca. 50 generations before being replaced by another), while performance was monitored on the competition map throughout the run. The results were mediocre, but it was encouraging to see that a sensible controller function could actually be evolved.

2 Implementation

This work was done using PerlGP[1] making heavy use also of the Perl Data Language (PDL[2]). The template controller function is given below:

```
sub simulation {
  my ($cities, $max) = @_;        # $max is 1400
  my $ncities = $cities->nelem/2; # $cities is 2 x N
  my @f = ();                     # list of force ids
  my $pos = pdl(160,120);         # initial position
  my $vel = pdl(0,0);             # velocity

  # while there are cities to visit and nVectors < max
  while (@f < $max && $cities->nelem) {

    my $thrust = {VECTOR};        # the important bit!

    # quantise thrust vector to force id
    my $f = 0;
    my $axis = $thrust->abs->qsorti->at(-1);
    my $mag = $thrust->at($axis);
    if ($mag > 0.1) {
      $f = ($axis == 0) ? 2 : 3;
    } elsif ($mag < -0.1) {
      $f = ($axis == 0) ? 4 : 1;
    }
    # push force id onto array
    push @f, $f;

    # now do the physics (with bug)
    my $acc = $FORCES[$f];        # lookup table
    $pos += 0.1 * $vel;
    $pos += 0.005 * $acc;
    $vel += $acc;

    # remove visited cities with some PDL magic
    $cities = $cities->dice(X,
      which(sumover(($cities-$pos)*($cities-$pos))>=25));
  }
  # return list of forces and nVisited
  return (\@f, $ncities-$cities->nelem/2);
}
```

This is called a template because the `{VECTOR}` part is expanded by the PerlGP system using a grammar to an arbitrary expression which evaluates to a PDL vector object (PDL is a high level matrix manipulation library, similar in spirit to MATLAB). We define a set of types: `SCALAR`, `VECTOR` (2×1), `CITYVALS` $(1 \times N)$, `CITYVECS` $(2 \times N)$, while the grammar dictates how they may be combined. Using various PDL operations, the different data types can be interconverted. For example, a `CITYVECS` quantity can be transformed into a `VECTOR` using the PDL average function (after switching the dimension order):

```
  VECTOR ::= CITYVECS->mv(-1,0)->average
```

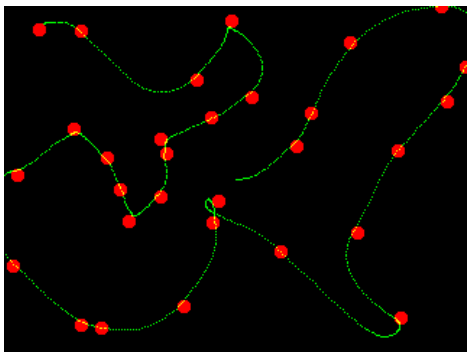
The full grammar is too lengthy to present here, but will be provided on request. One important feature, however, is the ability for the evolved program to pick (using `dice`) one `VECTOR` out of a `CITYVECS` matrix using highest (or lowest) value from a corresponding `CITYVALS` matrix. For example the closest city to the present position could be calculated using this type of sub-expression.

The evolution was performed using 10 inter-migrating populations of 2000 individuals each for three days.

3 Results

The typical number of steps needed to visit all cities on the Map30-g5 map was around 1000, however occasionally better solutions were seen. The best was 942, and its controller function is given below, followed by the trajectory.

```
(((-5 + (($cities->nelem/2) + (-4 + (($cities->nelem/2) +
sqdist($cities, ($pos + $vel)))))) / sqdist($cities, ((5 +
$cities->mv(-1,0)->average) + $vel)))->maximum * (($cities->dice(X,(3
+ (sqdist($cities, (5 + $cities->mv(-1,0)->average)) / ((-3 + (-5 +
(-4 + (-5 + (-4 + sqdist($cities, ($pos + $vel)))))))) /
$cities->maximum)))->qsorti->at(-1))->flat - (($vel + $pos) + ($vel /
2))) / ($cities->nelem/2)))
```



4 Discussion

I have not analysed the controller function to see if any interesting strategies were evolved. It would be interesting to add third-party heuristics to this approach.

References

1. R. M. MacCallum. Introducing a Perl Genetic Programming System: and Can Meta-evolution Solve the Bloat Problem? In *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 369–378, 2003.
2. C. Soeller, R. Schwebel, T. J. Lukka, T. Jenness, D. Hunt, K. Glazebrook, J. Cerney, and J. Brinchmann. The Perl Data Language. http://pdl.perl.org.