Evaluating Emergent Computing Algorithms

Lecture 5
Why do we need experiments anyway?

- EC algorithms are stochastic
  - Every time you run your algorithm, you may get a different answer
- So how do you know how well an algorithm really works:
  - Need a scientific method for experimenting and analysing the results
Good experiments need an objective!

- To get a good solution for a given problem
- To show that the algorithm is applicable in a new domain
- Show that a new “feature” improves a standard algorithm
- Show that your emergent computing algorithm is better than some other techniques
Good Experiments Need an Objective:

- Find the best parameters for your algorithm
- Get insights into algorithm behaviour
- See how it scales up with problem size
- See how the performance is affected by parameters of:
  - Algorithm
  - Problem
Performance Measures

- How can we measure algorithm performance?
- Partly, it depends on your objective:
  - Success Rate
  - Effectiveness (solution quality)
  - Efficiency (speed/memory)
Performance: Success Rate

- The percentage of runs ending in success
- Or... percentage ending in (say) 5% of optimum
- This is OK if you know the optimum - what if you don’t?
  - Compare to a previous result?
  - E.g. success achieved if this year’s timetable is 10% better than last year's
Performance: Mean best fitness (MBF)

- If your algorithm has an explicit evaluation function, you can measure MBF:
  - Measure the best fitness found after some fixed number of iterations
  - Average the best fitness values over many runs
Which one should I choose?

- Think about your application
  - What are you interested in?
- Low success rate and high MBF:
  - Algorithm consistently gets close but rarely gets to optimum
- Low MBF and high success rate:
  - When it goes wrong, it goes horribly wrong!
Measuring MBF and SR

- Both these measures reflect performance after some fixed amount of computational effort.
- If this changes, performance might change too!
Algorithm Efficiency

- Another approach is to measure how much computing power is needed to achieve a solution
  - Elapsed computer time?
  - CPU time?
  - User time?
- But this depends on hardware, operating system, compiler, network load, etc...
- Doesn’t make for reproducible results!
Counting Average Evaluations

- A better measure of efficiency is to consider how many solution evaluations the algorithm did to find a solution.
  - Effectively, the number of points visited in the search space.
- Again, you need to do this over lots of runs and average the results.
- Generally gives a fair measure of algorithm speed....
Counting Average Evaluations

- But, sometimes misleading!
  - Hidden labour - maybe each evaluation uses some local search during mutation
  - Some evaluations take longer than others - turning an infeasible solution into a feasible one
  - The evaluations are quick compared to the other parts of the algorithm
  - Difficult to compare different algorithms using this measure as the search steps might be different
A and B each design an algorithm and run it three times on the same problem and record max fitness:

<table>
<thead>
<tr>
<th>Expt1</th>
<th>Expt2</th>
<th>Expt3</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>50</td>
<td>51</td>
</tr>
</tbody>
</table>

MBF = \( \frac{49 + 50 + 51}{3} = 50 \)

<table>
<thead>
<tr>
<th>Expt1</th>
<th>Expt2</th>
<th>Expt3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

MBF = \( \frac{0 + 50 + 100}{3} = 50 \)
Statistics

- We can’t infer anything mathematically by just comparing two average values.
- The averages can be misleading:
- We need to perform some proper statistical analysis:
  - Standard deviations
  - T-Tests
Normal Distributions

- Most data-sets follow a normal (or Gaussian) distribution
- Most values are close to the average, and a few lie at either extreme
Standard Deviation (SD)

- SD tells how tightly a set of values is clustered around the average of those same values.
- It's a measure of dispersal, or variation, in a group of numbers.

- Calculate via: \[ \sigma = \sqrt{\frac{\sum (x - \bar{x})}{n-1}} \]

`STDDEV(A1:H1)`

![Histograms showing Large SD and Small SD](image)
So how does this help EC experiments?

- Reporting Standard Deviation gives you some information about the reliability of the algorithm.
- It gives you an idea of the spread of values produced from the algorithm.
- Allows you to directly compare average results from 2 experiments more meaningfully.
More about comparing results

- Assume we do 2 experiments with an evolutionary algorithm in which the objective is to maximise some function.
  - Expt 1: population size is 100
  - Expt 2: population size is 750
- Question: Does increasing the population size improve results?
Comparing Results:

<table>
<thead>
<tr>
<th></th>
<th>Pop size 100</th>
<th>Pop size 750</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mean</strong></td>
<td>100.033</td>
<td>96.309</td>
</tr>
<tr>
<td><strong>SD</strong></td>
<td>0.907</td>
<td>16.23</td>
</tr>
</tbody>
</table>

- Conclusion: Small population sizes are better than big ones
- NO! But a t-test can tell us!
The t-test tells us the probability that the 2 sets of data can come from different distributions - i.e. the experiments are different.

- Completely different dist.!
- Same dist., unlucky expts!
The t-test

- T-test gives you the probability the expts. are different.
- Scientifically, a probability of $\geq 95\%$ to make a claim with confidence
- $< 95\%$: data came from the same distribution, the experiments are equal!
- OR approach it the other way round:
  - probability $<5\%$ the distributions are the same
  - referred to as the null hypothesis
T-test

- Applying t-test to this data gives a probability of 68% that the distributions are different:
- SO, we cannot say with any confidence that there is any difference between the distributions
- Remember, we need a probability of $\geq 95\%$
Some points about t-test:

- Assumes a normal distribution
  - In practice, many things come from a normal distribution so it's OK
- Data comes from some approx. continuous interval
  - Not lots of isolated values, e.g. 1000, 50000
- Size of the two groups should be roughly similar
- If not, there are other tests: Wilcoxon etc.
  - Look them up if you are interested!
Using a T-test in practice

- There are two kinds of t-test:
  - Paired
  - Unpaired (mainly use this)
- A lot of software available for calculating confidence using t-test:

  http://graphpad.com/quickcalcs/ttest1.cfm
Test Problems for Experimental Comparisons

- If you want to say something about how good a technique is in general, you need to try a range of problems:

- Where to get the problems?
  - Benchmarks (web)
  - Problem generators (write your own)
  - Real problems (can be hard to obtain)
Reporting Experiments

- List all the parameters of your algorithm:
  - E.g. pop size, mutation rate, crossover method, size of chromosomes, # iterations, evaluation criteria

- **Key Question:** Can someone reproduce my experiments from my report?
Good experimental practice:

- Need to do a lot of runs!
  - How many? At least 10, 50 is better
- Report some measure of average solution quality AND sd, or t-test confidence values
- Try and automate this process
  - That’s what computers are for!
Reporting Results

- Tables
  - Can be hard to read if large
  - They should **summarise** results:
  - Report averages, not every result!
  - Highlight best/worst

<table>
<thead>
<tr>
<th>Pop. Size</th>
<th>Average</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>76.54</td>
<td>1.24</td>
</tr>
<tr>
<td>200</td>
<td>79.12</td>
<td>2.45</td>
</tr>
<tr>
<td>300</td>
<td><strong>85.21</strong></td>
<td>1.08</td>
</tr>
<tr>
<td>400</td>
<td>24.95</td>
<td>2.97</td>
</tr>
</tbody>
</table>
Reporting Results

- Graphs can be good as well
- Use error bars to mark min/max or standard deviation
- Make sure you label the axes!
Is this a good experiment?

- I invent a new mutation operator to be used with EA to solve a TSP problem:
- I download 10 examples problems from the web
- I run my EA and a standard EA 10 times on each problem
- I record the average result in each case
- My EA produces better results on 7 of the problems
- CONCLUSION: My mutation operator is great!
Checklist for good experiments:

- What has my experiment told me?
- How relevant are the results:
  - Are the test functions typical of real problems?
  - Are they relevant from an academic perspective?
- What the scope of the claims about the technique is?
  - Why does my technique work on some problems and not others?
  - Can I correlate problem features with algorithm performance?
- If the results generalise to other problems in this area?
Checklist:

- Can someone reproduce my experiments from my report?
- Do I know:
  - what would have happened if a different performance metric was used
  - how sensitive are the results to the algorithm’s parameters?
  - If the differences observed are statistically significant or due to random effect?
Summary

- If you can answer “yes” to the previous questions, then
  - The experimental methodology is excellent!
- If you can’t:
  - Think again.....
Some Tips for Doing Good Experiments

- Be organized
- Decide what you want & define appropriate measures
- Choose test problems carefully
- Make an experiment plan (estimate time when possible)
- Perform sufficient number of runs
- Keep all experimental data (never throw away anything)
Some Tips for Doing Good Experiments

- Use good statistics ("standard" tools from Web, Excel etc)
- Present results well (figures, graphs, tables, ...)
- Watch the scope of your claims
- Aim at generalisable results
- Publish code for reproducibility of results (if applicable)
Additional Material:

- [http://www.cs.vu.nl/~7Egusz/ecbook/ecbook-course.html](http://www.cs.vu.nl/~7Egusz/ecbook/ecbook-course.html)
- Very good lecture notes covering the material from this lecture and more detail