Doctorate of Philosophy

# Schrödinger's Catwalk

Brian Flynn

University of Bristol

December, 2020

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LISTINGS

# ACRONYMS

| | |
|---|---|
| ES | exploration strategy. 2, 4, 7 |
| GA | genetic algorithm. 2 |
| GES | genetic exploration strategy. 2 |
| QHL | quantum Hamiltonian learning. 4 |
| QMLA | Quantum Model Learning Agent. vii, 2, 4 |

# GLOSSARY

| | |
|---|---|
| OF | objective function. 2 |
| instance | a single implementation of the Quantum Model Learning Agent (QMLA) algorithm. 4 |
| run | collection of QMLA instances. 4 |

Part I

# THEORETICAL STUDY

# 1 GENETIC ALGORITHMS

The QMLA framework lends itself easily to the family of optimsation techniques called *evolutionary algorithms*, where individuals, sampled from a population of candidates, are considered, in generations, as solutions to the given problem, and iterative generations aim to efficiently search the available population, by mimicing biological evolutionary mechanisms [1]. In particular, we develop a exploration strategy (ES) which incorporates an genetic algorithm (GA) in the generation of models; GAs are a subset of evolutionary algorithms where candidate solutions are expressed as strings of numbers representing some configuration of the system of interest [2]. Here we will first introduce the concept of a GA, before describing the adaptations which allow us to build a genetic exploration strategy (GES).

## 1.1 GENETIC ALGORITHM DEFINITION

GAs work by assuming a given problem can be optimised, if not solved, by a single candidate among a fixed, closed space of candidates, called the population, $\mathcal{P}$. A number of candidates are sampled at random from $\mathcal{P}$ into a single *generation*, and evaluated through some objective function (OF), which assesses the fitness of the candidates at solving the problem of interest. Candidates from the generation are then mixed together to produce the next generation's candidates: this *crossover* process aims to combine only relatively strong candidates, such that the average candidates' fitness improve at each successive generation, mimicing the biological mechanism whereby the genetic makeup of offspring is an even mixture of both parents. The selection of strong candidates as parents for future generations is therefore imperative; in general parents are chosen according to their fitness as determined by the OF. Buidling on this biological motivation, much of the power of GAs comes from the concept of *mutation*: while offspring retain most of the genetic expressions of their parents, some elements are mutated at random. Mutation is crucial in avoiding local optima of the OF landscape by maintaining diversity in the examined subspace of the population.

Pseudocode for a generic GA is given in Algorithm 1, but we can also informally define the procedure as follows. Given access to the population, $\mathcal{P}$,

1. Sample $N_m$ candidates from the population at random
   (a) call this group of candidates the first generation, $\mu$.
2. Evaluate each candidate $\gamma_j \in \mu$.
   (a) each $\gamma_j$ is assigned a fitness, $g_j$
   (b) the fitness is computed through an objective function acting on the candidate, $g(\gamma_j)$.
3. Map the fitnesses of each candidate, $\{g_j\}$, to selection probabilities for each model, $\{s_j\}$

   (a) e.g. by normalising the fitnesses, or by removing some poorly-performing candidates and then normalising.

4. Generate the next generation of candidates, $\mu'$
   (a) $\mu = \{\}$
   (b) Select pairs of parents, $p_1, p_2$, from $\mu$
      i. Each candidate's probability of being chosen is given by their $s_j$
   (c) Cross over $p_1, p_2$ to produce children candidates, $c_1, c_2$.
      i. mutate $c_1, c_2$ according to some random probabilistic process
      ii. keep $c_i$ only if it is not already in $\mu'$, to ensure $N_m$ unique models are tested at each generation.
   (d) until $|\mu'| = N_m$, iterate to step (b).
5. Until the $N_g^{th}$ generation is reached, iterate to step 2..
6. The strongest candidate on the final generation is deemed the solution to the posed problem.

Candidates are manifested as *chromosomes*, i.e. strings of fixed length, whose entries, called *genes*, each represent some element of the system. In general, genes can have continuous values, although usually, and for all purposes in this thesis, genes are binary, capturing simply whether or not the gene's corresponding feature is present in the chromosome.

### 1.1.1 *Example: knapsack problem*

One commonly referenced combinatorial optimisation problem is the *knacksack problem*: given a set of objects, where each object has a defined mass and also a defined value, determine the set of objects to pack in a knapsack which can support a limited weight, such that the value of the packed objects is maximised. Say there are $n$ objects, we can write the vector containing the values of those objects as $\vec{v}$, and the vector of their weights as $\vec{w}$. We can then represent configurations of object sets as candidate vectors $\vec{\gamma}_j$, whose genes are binary, and simply indicate whether or not the associated object is included in the set. For example, $\vec{\gamma}_j = 100001$ indicates a set of objects consisting only of those indexed first and last, with none of the intermediate objects included.

The fitness of any candidate is then given by the total value of that configuration of objects, $v_j = \vec{v} \cdot \vec{\gamma}_j$, but candidates are only admitted[1] if the weight of the corresponding set of objects is less than the capacity of the knapsack, i.e. $\vec{w}_j \cdot \gamma_j \leq w_{max}$.

1 Note there are alternative strategies to dealing with candidates who violate the weight condition, such as to impose a penalty within the OF, but for our purposes let us assume we simply disregard violators.

**Algorithm 1:** Genetic algorithm

```
Input: $\mathcal{P}$                              // Population of candidate models
Input: $g()$                                      // objective funtion
Input: map_g_to_s()                               // function to map fitness to selection probability
Input: select_parents()                           // function to select parents among generation
Input: crossover()                                // function to cross over two parents to produce offspring
Input: $N_g$                                      // number of generations
Input: $N_m$                                      // number of candidates per generation

Output: $\gamma'$                                 // strongest candidate

$\mu \leftarrow$ sample $(\mathcal{P}, N_m)$
for $i \in 1, ..., N_g$ do
    for $\gamma_j \in \mathbb{S}$ do
        $g_j \leftarrow g(\gamma_j)$              // assess fitness of candidate
    end
    $\{s_j\} \leftarrow$ map_g_to_s($\{g_j\}$)    // map fitnesses to normalised selection probability
    $\mu_c = \arg\max_{s_j} \{\gamma_j\}$         // record champion of this generation

    $\mu \leftarrow \{\}$                         // empty set for next generation
    while $|\mu| < N_m$ do
        $p_1, p_2 \leftarrow$ select_parents($\{s_j\}$)    // choose parents based on candidates' $s_j$
        $c_1, c_2 \leftarrow$ crossover($p_1, p_2$)        // generate offspring candidates based on parents
        for $c \in \{c_1, c_2\}$ do
            if $c \notin \mu$ then
                $\mu \leftarrow \mu \cup \{c\}$   // keep if child is new
            end
        end
    end
end
$\gamma' \leftarrow \arg\max_{s_j} \{\gamma_j \in \mu\}$   // strongest candidate on final generation

return $\gamma'$
```

For example with $n = 6$, where each individual object has value $< 50$ and weight $< 25$ and $w_{max} = 50$, recalling $\vec{\gamma}_j = 100001$, say,

$$\vec{v} = (48 \quad 15 \quad 26 \quad 19 \quad 39 \quad 15) \implies v_j = \vec{\gamma}_j \cdot \vec{v} = 48 + 15 = 63; \tag{1.1a}$$

$$\vec{w} = (21 \quad 7 \quad 2 \quad 17 \quad 10 \quad 20) \implies w_j = \vec{\gamma}_j \cdot \vec{w} = 21 + 20 = 41. \tag{1.1b}$$

We can hence assess the fitness of $\gamma_j$ as 63 and deem it a valid candidate since it does not exceed the weight threshold. We can likewise compute the total weight and value of a series of randomly generated candidates, and deem them valid or not.

| Candidate | Value | Weight | Valid |
|---|---|---|---|
| 110000 | 63 | 28 | Yes |
| 000011 | 54 | 30 | Yes |
| 011101 | 75 | 46 | Yes |
| 101010 | 113 | 33 | Yes |
| 000101 | 34 | 37 | Yes |
| 010111 | 88 | 54 | No |
| 011011 | 95 | 39 | Yes |
| 110011 | 117 | 58 | No |
| 000000 | 0 | 0 | Yes |
| 110001 | 78 | 48 | Yes |
| 100010 | 87 | 31 | Yes |
| 011110 | 99 | 36 | Yes |

Table 1.1: Candidate solutions to the knapsack problem.

The strongest (valid) candidates from Table 1.1 are 101010, 011110. By spawning from these candidates through a one-point crossover at the midpoint, we get $\gamma_{c_1} = 101110, \gamma_{c_2} = 011010$, from which we can see $v_{c_1} = 132, w_{c_1} = 50$, i.e. by combining two strong candidates we produce the strongest-yet-seen valid candidate.

By repeating this procedure, it is expected to uncover candidates which optimise $v_j$ while maintining $w_j \leq w_{max}$, or at least to produce near-optimal solutions, using far less time/resources than brute-force evaluation of all candidates, which is usually sufficient. For instance, if there $n = 100$ objects to consider, there are $2^{100} \approx 10^{30}$ candidates to consider; the most powerful supercomputers in the world currently claim on the order of Exa-FLOPs, i.e. $10^{18}$ operations per second, of which say $\sim 1000$ operations are required to test each candidate, meaning $10^{15}$

candidates can be checked per second. This would still require $10^{12}$ seconds to solve absolutely, so it is reasonable in cases like this to accept *approximately optimal* solutions[2].

### 1.1.2 *Selection mechanism*

### 1.1.3 *Offspring production*

crossover and mutation mechanisms

### 1.1.4 *Candidate evaluation*

## 1.2 ADAPTATION TO QMLA FRAMEWORK

## 1.3 OBJECTIVE FUNCTIONS

## 1.4 APPLICATION

2 Simply put: in machine learning, *good enough* is good enough.

# APPENDIX

# A FIGURE REPRODUCTION

Most of the figures presented in the main text are generated directly by the QMLA framework. Here we list the implementation details of each figure so they may be reproduced by ensuring the configuration in Table A.1 are set in the launch script. The default behaviour of QMLA is to generate a results folder uniquely identified by the date and time the run was launched, e.g. results can be found at the *results directory* qmla/Launch/Jan_01/12_34. Given the large number of plots available, ranging from high-level run perspective down to the training of individual models, we introduce a plot_level $\in \{1, ..., 6\}$ for each run of QMLA: higher plot_level informs QMLA to generate more plots.

Within the results directory, the outcome of the run's instances are stored, with analysis plots broadly grouped as

7. evaluation: plots of probes and times used as the evaluation dataset.
8. single_instance_plots: outcomes of an individual QMLA instance, grouped by the instance ID. Includes results of training of individual models (in model_training), as well as sub-directories for anlaysis at the branch level (in branches) and comparisons.
9. combined_datasets: pandas dataframes containing most of the data used during analysis of the run. Note that data on the individual model/instance level may be discarded so some minor analyses can not be performed offline.
10. exploration_strategy_plots plots specifically required by the ES at the run level.
11. champion_models: analysis of the models deemed champions by at least one instance in the run, e.g. average parameter estimation for a model which wins multiple instances.
12. performance: evaluation of the QMLA run, e.g. the win rate of each model and the number of times each term is found in champion models.
13. meta analysis of the algorithm' implementation, e.g. timing of jobs on each process in a cluster; generally users need not be concerned with these.

In order to produce the results presented in this thesis, the configurations listed in Table A.1 were input to the launch script. The launch scripts in the QMLA codebase consist of many configuration settings for running QMLA; only the lines in snippet in Listing A.1 need to be set according to altered to retrieve the corresponding figures. Note that the runtime of QMLA grows quite quickly with $N_E, N_P$ (except for the AnalyticalLikelihood ES), especially for the entire QMLA algorithm; running quantum Hamiltonian learning (QHL) is feasible on a personal computer in $< 30$ minutes for $N_e = 1000; N_p = 3000$.

```
#!/bin/bash
```

```
###############
# QMLA run configuration
###############
num_instances=1
run_qhl=1 # perform QHL on known (true) model
run_qhl_mulit_model=0 # perform QHL for defined list of models.
exp=200 # number of experiments
prt=1000 # number of particles

###############
# QMLA settings
###############
plot_level=6
debug_mode=0

###############
# Choose an exploration strategy
###############

exploration_strategy='AnalyticalLikelihood'
```

Listing A.1: "QMLA Launch scipt"

| Figure | Exploration Strategy | Algorithm | $N_E$ | $N_P$ | Data |
|---|---|---|---|---|---|
| **??** | AnalyticalLikelihood | QHL | 500 | 2000 | Nov_16/14_28 |
| **??** | DemoIsing | QHL | 500 | 5000 | Nov_18/13_56 |
| **??** | DemoIsing | QHL | 1000 | 5000 | Nov_18/13_56 |
| **??** | DemoIsing | QHL | 1000 | 5000 | Nov_18/13_56 |
| **??** | IsingLatticeSet | QMLA | 1000 | 4000 | Nov_19/12_04 |
| 3***??** | IsingLatticeSet | QMLA | 1000 | 4000 | Sep_30/22_40 |
|  | HeisenbergLatticeSet | QMLA | 1000 | 4000 | Oct_22/20_45 |
|  | FermiHubbardLatticeSet | QMLA | 1000 | 4000 | Oct_02/00_09 |

Table A.1: Implementation details for figures used in the main text.

# B

# EXAMPLE EXPLORATION STRATEGY RUN

A complete example of how to run the ;sqmla framework, including how to implement a custom ES, and generate/interpret analysis, is given.

# BIBLIOGRAPHY

[1] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[2] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.