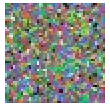
A Concurrent Implementation for Genetic Algorithms



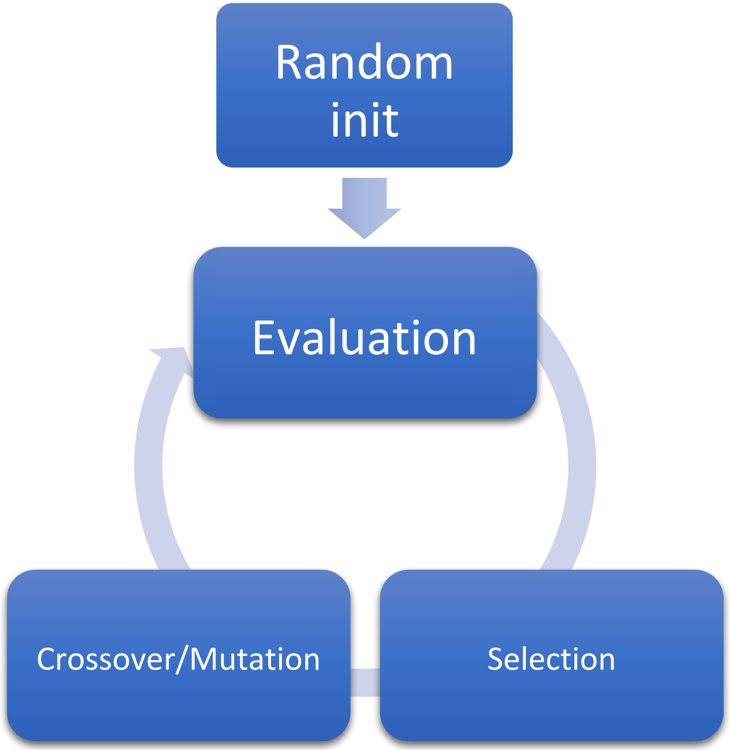
Nick Knowles (knowlen@wwu.edu)

Western Washington University

Presentation Overview

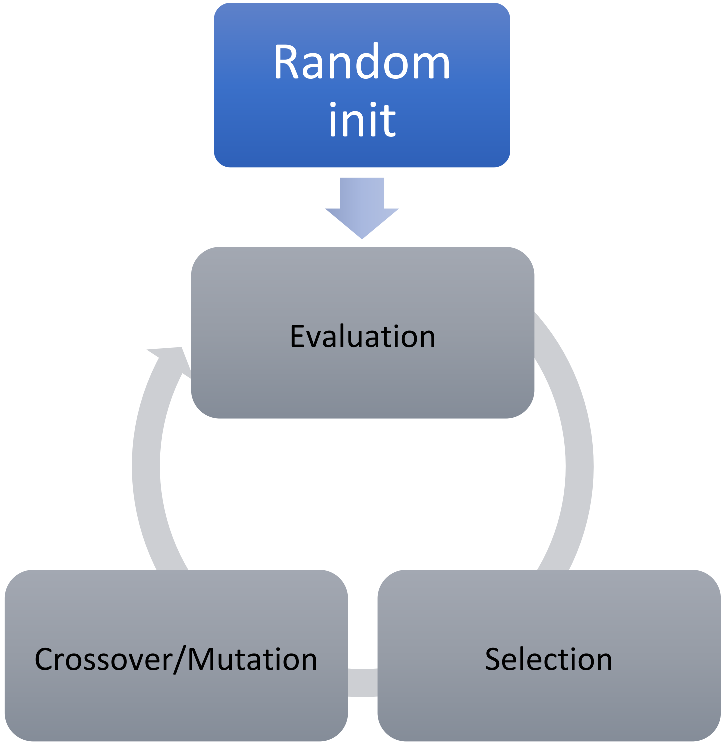
- Algorithm overview
- Motivation for concurrency
- Image approximation results
- Continued work

Genetic Algorithms



Mimics biological evolution and natural selection to evolve a population of "candidate solutions" towards a global optimum for some underlying task or goal.

Genetic Algorithms

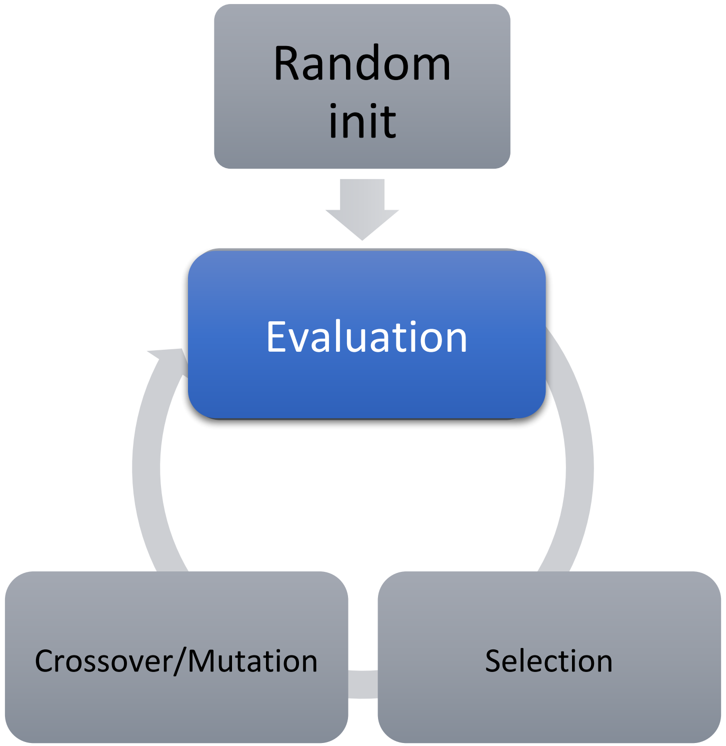


Create a population vector, **P** of randomly initialized "candidate solutions"

$$\mathbf{P} = \begin{pmatrix} [18,\ 5,\ 56,\ 1] \\ [42,\ 1,\ 35,\ 12] \\ [3,\ 27,\ 83,\ 7] \\ \vdots \\ [\ldots,\ \ldots,\ \ldots,\ \ldots] \end{pmatrix}$$

Genetic Algorithms

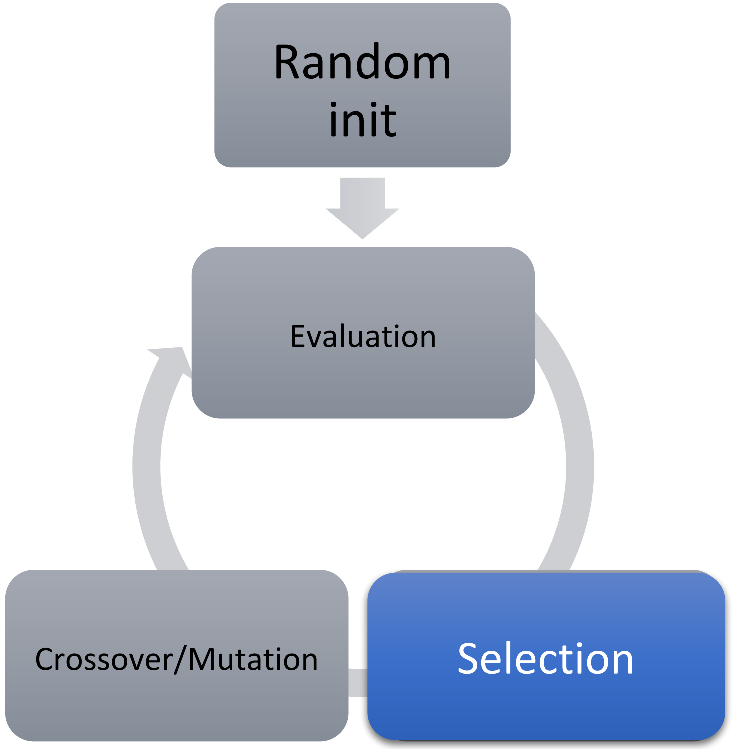
For each **c** in **P**:
run **c** through **target function** and store how well it does



Create a population vector, **P** of randomly initialized "candidate solutions"

$$
\mathbf{P} = \begin{pmatrix} [18,\ 5,\ 56,\ 1] \\ [42,\ 1,\ 35,\ 12] \\ [\ 3,\ 27,\ 83,\ 7] \\ \cdot \\ \cdot \\ \cdot \\ [\ldots,\ \ldots,\ \ldots,\ \ldots] \end{pmatrix}
$$

Genetic Algorithms



Create a population vector, **P** of randomly initialized "candidate solutions"

$$\mathbf{P} = \begin{pmatrix} [18,\ 5, 56,\ 1] \\ [42, 1, 35, 12] \\ [\ 3, 27, 83,\ 7] \\ . \\ . \\ . \\ [\ldots, \ldots, \ldots, \ldots] \end{pmatrix}$$

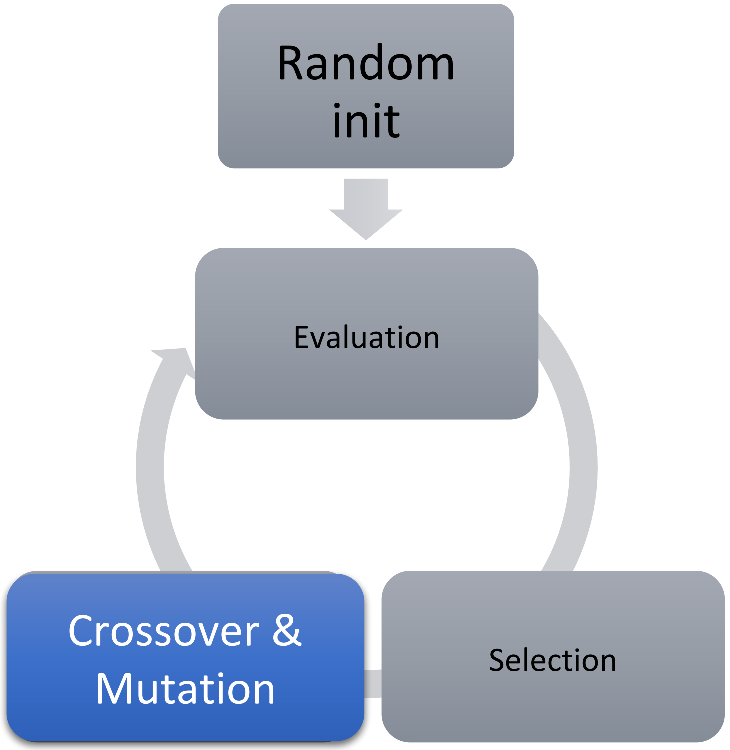
For each **c** in **P**:
run **c** through **target function** and store how well it does

Sample a pseudo random set of candidates from **P**, biased towards higher evaluation scores

Genetic Algorithms

For each **c** in **P**:
run **c** through **target function** and store how well it does

Sample a pseudo random set of candidates from **P**, biased towards higher evaluation scores



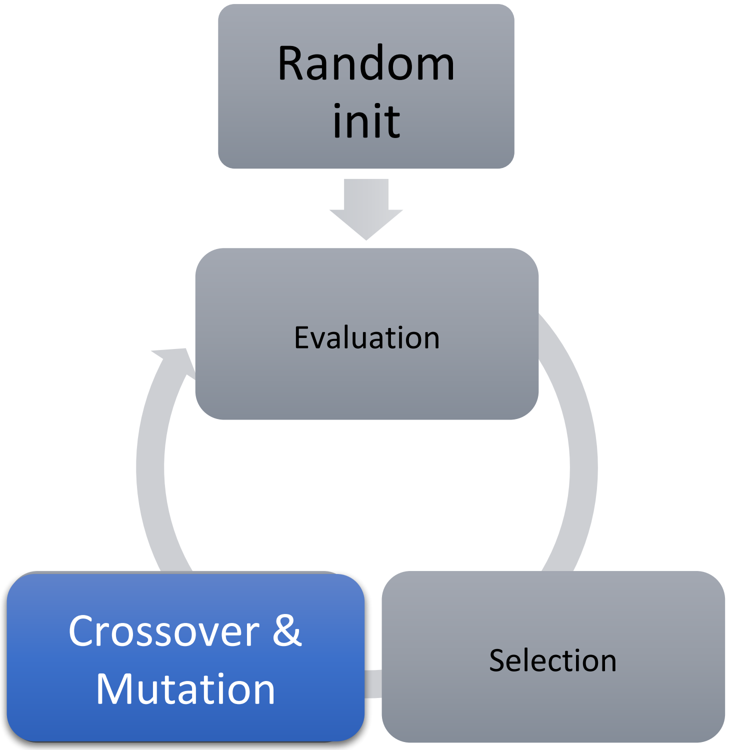
Create a population vector, **P** of randomly initialized "candidate solutions"

P =

[18, 5, 56, 1]
[42, 1, 35, 12]
[3, 27, 83, 7]
.
.
.
[..., ..., ..., ...]

Generate a random bit mask of length len(**c**), apply mask and **XOR**(mask) to two "parent" candidates and sum them to create two "child" candidates

Genetic Algorithms



For each **c** in **P**:
run **c** through **target function** and store how well it does

Sample a pseudo random set of candidates from **P**, biased towards higher evaluation scores

Create a population vector, **P** of randomly initialized "candidate solutions"

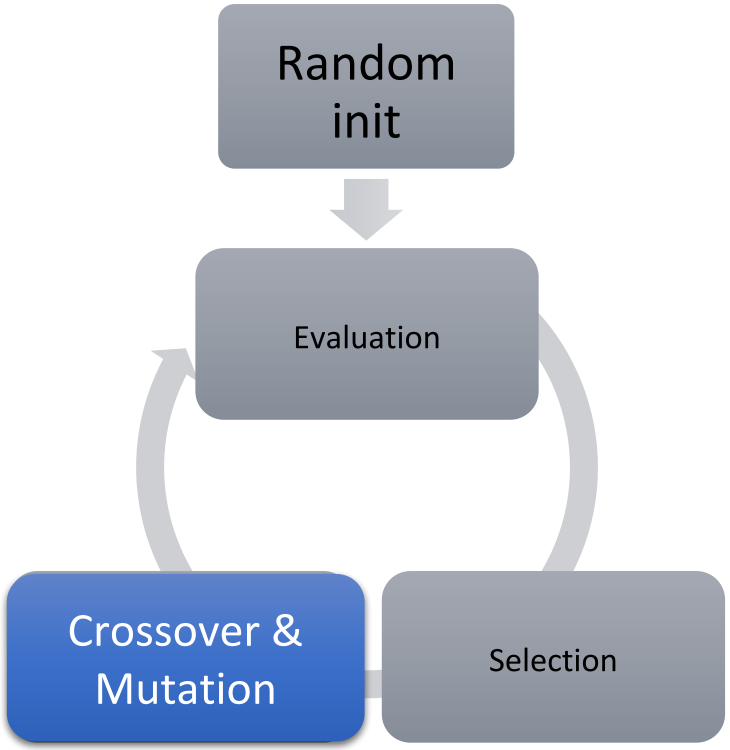
$$\mathbf{P} = \begin{pmatrix} [18,\ 5, 56,\ 1] \\ [42, 1, 35, 12] \\ [\ 3, 27, 83,\ 7] \\ \cdot \\ \cdot \\ \cdot \\ [\ldots, \ldots, \ldots, \ldots] \end{pmatrix}$$

Generate a random bit mask of length len(**c**), apply mask and **XOR**(mask) to two "parent" candidates and sum them to create two "child" candidates

Eg; mask = [0, **1**, 0, **1**]
P1 = [5, 3, 7, 1]
P2 = [6, 8, 2, 4]

C1 = [5, 8, 7, 4]
C2 = [6, 3, 2, 1]

Genetic Algorithms



Create a population vector, **P** of randomly initialized "candidate solutions"

$$\mathbf{P} = \begin{bmatrix} [18,\ 5, 56,\ 1] \\ [42, 1, 35, 12] \\ [\ 3, 27, 83,\ 7] \\ \cdot \\ \cdot \\ \cdot \\ [\ldots, \ldots, \ldots, \ldots] \end{bmatrix}$$

For each **c** in **P**:
  run **c** through **target function** and store how well it does

Sample a pseudo random set of candidates from **P**, biased towards higher evaluation scores

Generate a random bit mask of length len(**c**), apply mask and **XOR**(mask) to two "parent" candidates and sum them to create two "child" candidates

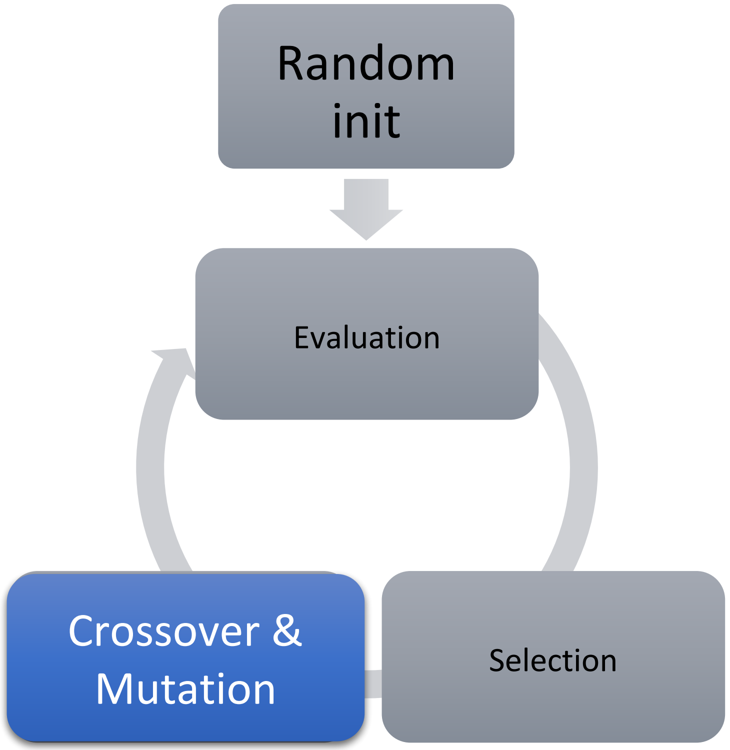
Eg; mask = [0, **1**, 0, **1**]
P1 = [5, 3, 7, 1]
P2 = [6, 8, 2, 4]

C1 = [5, 8, 7, 4]
C2 = [6, 3, 2, 1]

Every child has a small chance to mutate in some random way:

C1 = [7, 8, 7, 4]

Genetic Algorithms



Create a population vector, **P** of randomly initialized "candidate solutions"

P = [18, 5, 56, 1]
[42, 1, 35, 12]
[3, 27, 83, 7]
.
.
.
[..., ..., ..., ...]

For each **c** in **P**:
run **c** through **target function** and store how well it does

Sample a pseudo random set of candidates from **P**, biased towards higher evaluation scores

Generate a random bit mask of length len(**c**), apply mask and **XOR**(mask) to two "parent" candidates and sum them to create two "child" candidates

Eg; mask = [0, **1**, 0, **1**]
P1 = [5, 3, 7, 1]
P2 = [6, 8, 2, 4]

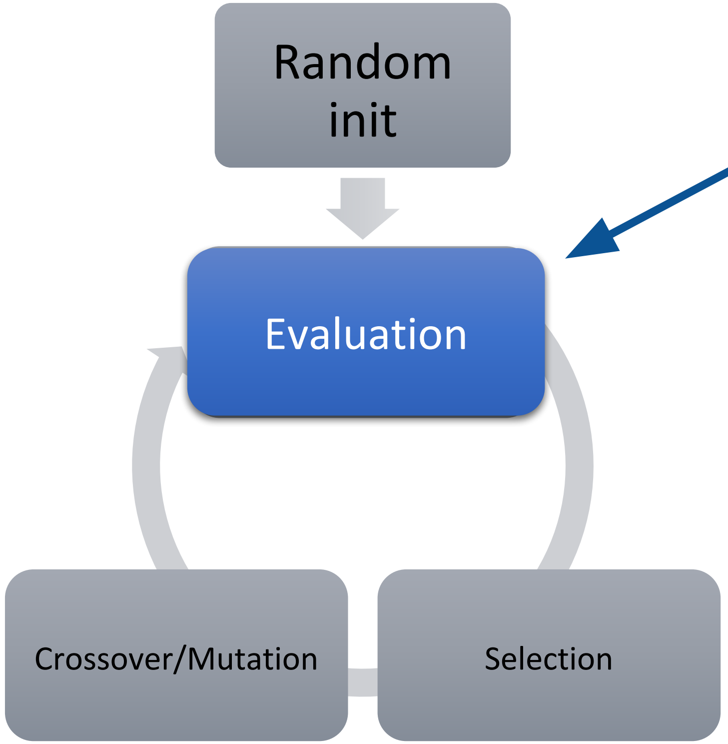
C1 = [5, 8, 7, 4]
C2 = [6, 3, 2, 1]

Replacing parents

Every child has a small chance to mutate in some random way:

C1 = [7, 8, 7, 4]

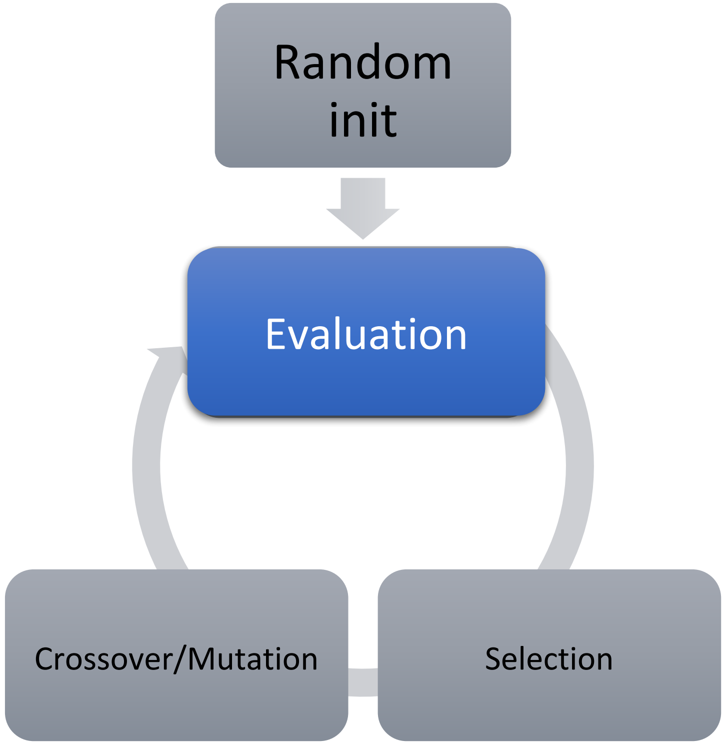
Need for Concurrency



This can take seconds, hours, or days depending on the problem

> Many problems have very large evaluation times.
> Need to evaluate on 100s or 1000s of candidate solutions at each iteration.
> Bayesian optimization or Simulated Annealing favored over Genetic Algs because of this (despite comparable precision).

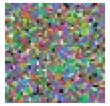
Executive Summary



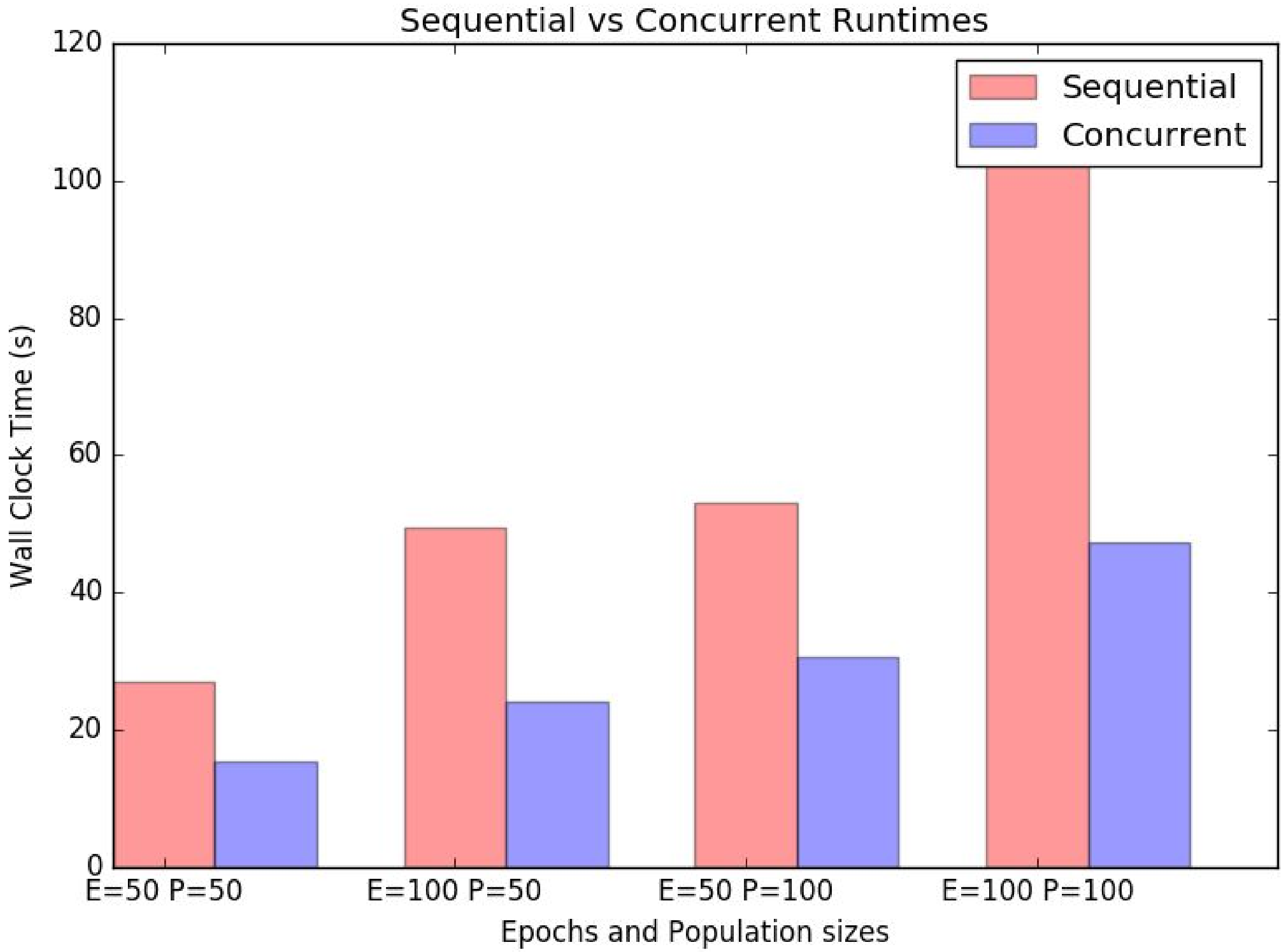
My implementation uses multiprocessing (because Python...) to perform these evaluations in parallel.

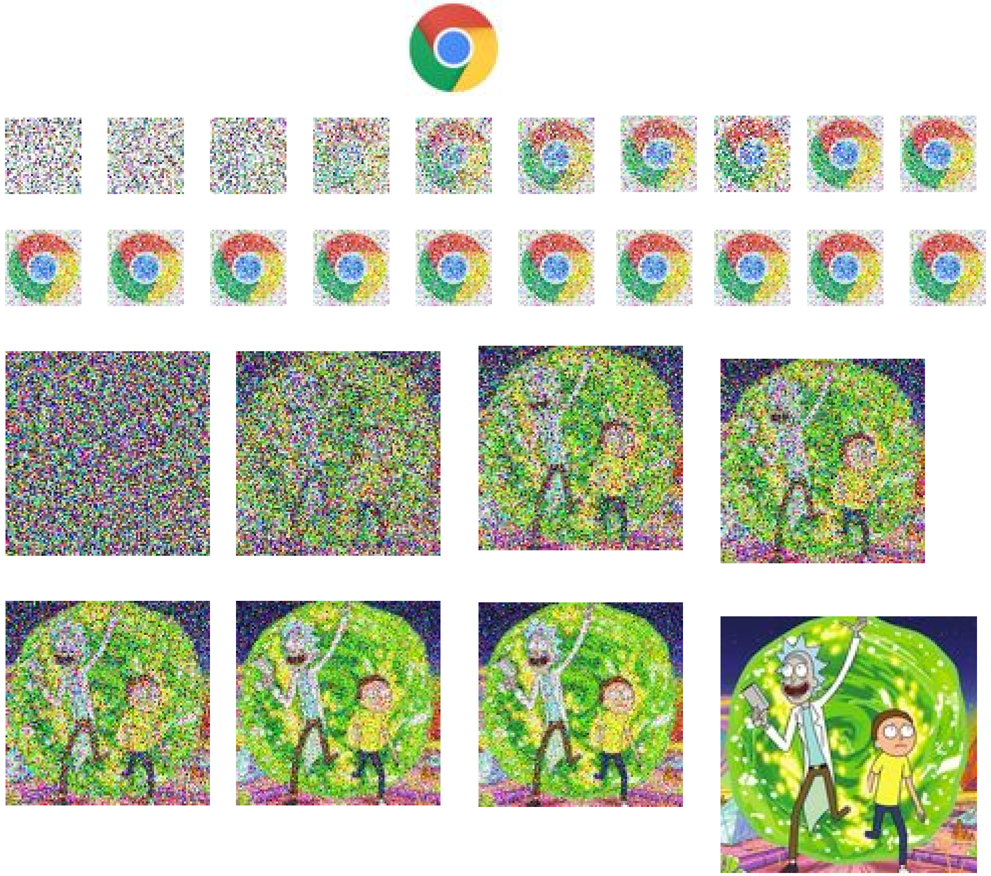
This allows the algorithm to complete more evaluations per wallclock time, and allows it to scale with iterations (epochs) at a more reasonable rate.

I apply the algorithm to the task of image approximation. Starting with a population of randomly generated NumPy arrays, and evolving towards a specified target image.



Results (Preliminary)





Runtimes key

26.930s	49.678s	53.150s	1m43.949s
15.532s	24.156	30.727	47.233s

Next steps…

> Extend concurrent elements of the Python code with C++.

> Experiment over a wider range of hyperparameters.

> Implement different selection & replacement algorithms.

> Look into distributed computing methods for larger problems (eg; dispatch concurrent evaluations to every computer on some cluster).

> Apply to more practical domains.

Thanks!

/knowlen

WESTERN
WASHINGTON UNIVERSITY

HUTCH
RESEARCH