Assignment_4_Populations_jdonova6-withIndividual

September 23, 2021

1 Assignment 4: Populations

In our last assignment, we used systematic experimentation to figure out how and why our hill-climber was working -- paying particular attention to how we modify selection pressure, and how that affects search trajectories over fitness landscapes of varying ruggedness.

In this assignment, we'll continue to build out the basic backbone of our evolutionary algorithm, while applying the same lens of systematic investigation and exploration around implementation details. In particular, we'll expand our single-parent/single-child hillclimber into a full population, and explore how crossover and selection manifest themselves in our algorithm.

```
In [1]: # imports
        import numpy as np
        import copy
        import matplotlib.pyplot as plt
        plt.style.use('seaborn')

        import scikits.bootstrap as bootstrap
        import warnings
        warnings.filterwarnings('ignore') # Danger, Will Robinson! (not a scalable hack, and m

        import scipy.stats # for finding statistical significance
        import random
```

1.0.1 Q1: Implementing Individuals within a Population

As we beging to work with populations, it will get a increasingly messy to keep track of each individual's genome and fitness seperately as they move around the population and through generational time. To help simplify this, let's implement each individual within a population as an instance of an `Individual` class. To start, this class will be quite simple and will just be an object which has attributes for both the individual's `genome` and its `fitness`. Since we will only be using fitness functions that depend on a single individual in this assignment, let's also implement an `eval_fitness` for each individual that will evaluate and update its stored fitness value when called.

```
In [2]: class Individual:

            def __init__(self, fitness_function, bit_string_length):
```

```
        self.genome = []
        for i in range(bit_string_length):
            self.genome.append(random.randint(0,1))
        self.fitness_function = fitness_function
        self.fitness = self.eval_fitness()


    def eval_fitness(self):
        self.fitness = self.fitness_function(self.genome)
```

1.0.2 Q2: Modifying the hillclimber

Let's take the basic hillclimber from our last assignment and turn it into a full fleged evolutionary algorithm. Again, please feel free to leverage your prior work (or our prior solution sets) and copy-and-paste liberally.

In particular, our first version of this algorithm will have a number of parents and a number of children given as parameters (a la evolutionary strategies), two-point crossover (of randomly selected parents), and truncation selection. Please also include arguemtns to this evolutionary_algorithm function which allow you dictate whether the algorithm will use mutation (the same single bit flip we used before), crossover, or both (for use in the following question).

To get a finer-grain look at convergence rates of these different approaches, let's also modify the output of this function to return the fitness of the top individual at each generation.

```
In [3]: def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=1
            """ Evolutinary Algorithm (copied from the basic hillclimber in our last assignmen

                parameters:
                fitness_function: (callable function) that return the fitness of a genome
                                  given the genome as an input parameter (e.g. as defined in
                total_generations: (int) number of total iterations for stopping condition
                num_parents: (int) the number of parents we downselect to at each generation (
                num_children: (int) the number of children (note: parents not included in this
                bit_string_length: (int) length of bit string genome to be evoloved
                num_elements_to_mutate: (int) number of alleles to modify during mutation (0 =
                crossover (bool): whether to perform crossover when generating children

                returns:
                fitness_over_time: (numpy array) track record of the top fitness value at each
            """

            # initialize record keeping
            fitness_over_time = []

            # the initialization proceedure
            parents = []
            children = []
            for i in range(num_parents):
                parents.append(Individual(fitness_function, bit_string_length))
```

```
# get population fitness
for i in range(num_parents):
    parents[i].eval_fitness()

for i in range(total_generations): # repeat

    # the modification procedure
    children = []

    # inheretance
    if not crossover:
        children = copy.deepcopy(parents)

    # crossover
    if crossover:
        while len(children)<(num_children):
            random.shuffle(parents)
            for j in range(0, num_parents, 2):

                # select both parents
                first_parent = parents[j]
                second_parent = parents[j+1]

                # randomly select two points for crossover
                crossover_points = random.sample(range(0,bit_string_length), 2)

                # Do we want to disallow crossover from higher to lower numbers?
                # TODO: Remove below line and edit crossover to work when higher n
                crossover_points = sorted(crossover_points)

                # switch genes between these two points between the two individual
                child1 = Individual(fitness_function, bit_string_length)
                child1.genome = parents[j].genome[:crossover_points[0]] + parents[
                child2 = Individual(fitness_function, bit_string_length)
                child2.genome = parents[j+1].genome[:crossover_points[0]] + parents

                # Or do we just keep the children no matter what
                children.append(child1)
                children.append(child2)


        if len(children) > num_children:
            children = children[:num_children]

    # mutation
    if num_elements_to_mutate > 0:
        # loop through each child
```

```
            for j in range(len(children)):

                # create storage array for elements to mutate
                elements_mutated = []
                elements_mutated = random.sample(range(0,bit_string_length), num_eleme

                # loop through the array of indices to be mutated
                for k in range(len(elements_mutated)):
                    # set the bit to the opposite of what it currently is
                    if children[j].genome[elements_mutated[k]] == 0:
                        children[j].genome[elements_mutated[k]] = 1
                    else:
                        children[j].genome[elements_mutated[k]] = 0

        # the assessement procedure
        for j in range(num_children):
            children[j].eval_fitness()

        # selection procedure
        parents += children.copy()
        parents = sorted(parents, key=lambda x: x.fitness, reverse=True)
        parents = parents[:num_parents]

        # record keeping
        fitness_over_time.append(parents[0].fitness)

    return fitness_over_time
```

1.0.3 Q3: Running Experiments

Similar to last week, let's systemtically run and plot the results. To start let's use 50 parents (mu) and 50 children (lambda). For simplicity, let's go back to the one-max problem (and normalize the fitness, using `np.mean` isntead of `np.sum` for our fitness function in case we want to make comparisons across different genome lengths -- though for now, let's start with a bit string genome of length 200).

Also taking pieces from your experimental comparison scripts from last week, please run this for the the case of mutation only, crossover only, and employing both mutation and crossover. Run 20 independent repitions for each condition.

In [4]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 200
num_parents = 50
num_children = 50
fitness_function = np.mean
experiment_results = {}
```

```
experiment_results['mutation_only'] = []
experiment_results['crossover_only'] = []
experiment_results['crossover_and_mutation'] = []

for i in range(num_runs):
    # mutation only
    m_only = evolutionary_algorithm(fitness_function, total_generations, num_parents, r
    experiment_results['mutation_only'].append(m_only)

    # crossover only
    c_only = evolutionary_algorithm(fitness_function, total_generations, num_parents, r
    experiment_results['crossover_only'].append(c_only)

    # both crossover and mutation
    c_and_m = evolutionary_algorithm(fitness_function, total_generations, num_parents,
    experiment_results['crossover_and_mutation'].append(c_and_m)
```

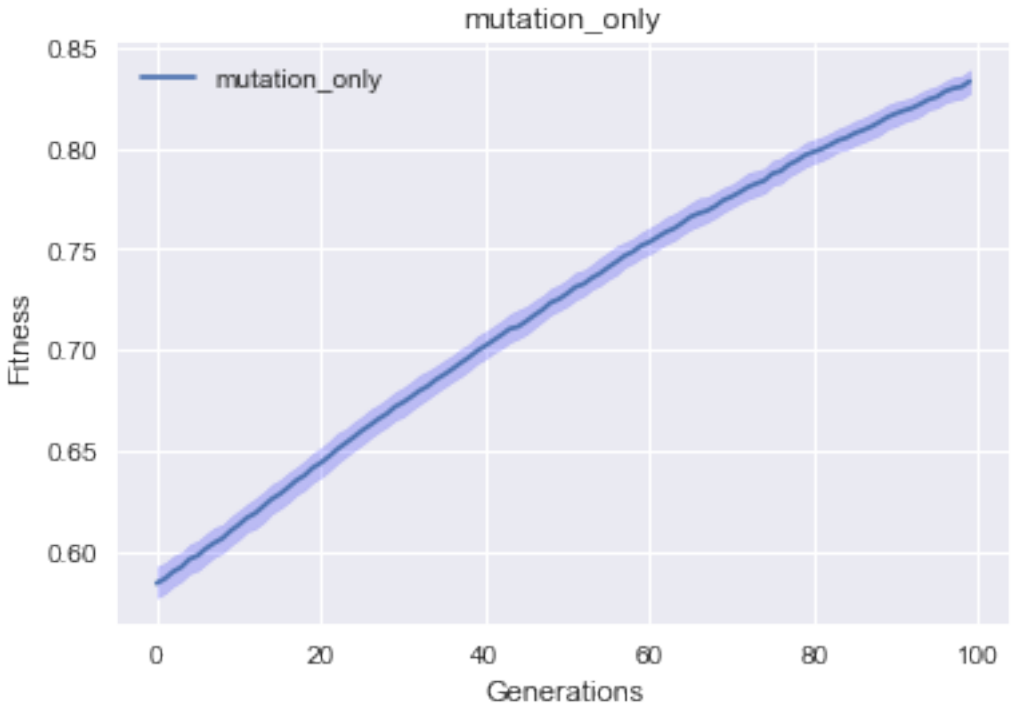
1.0.4 Q3b: Visualization

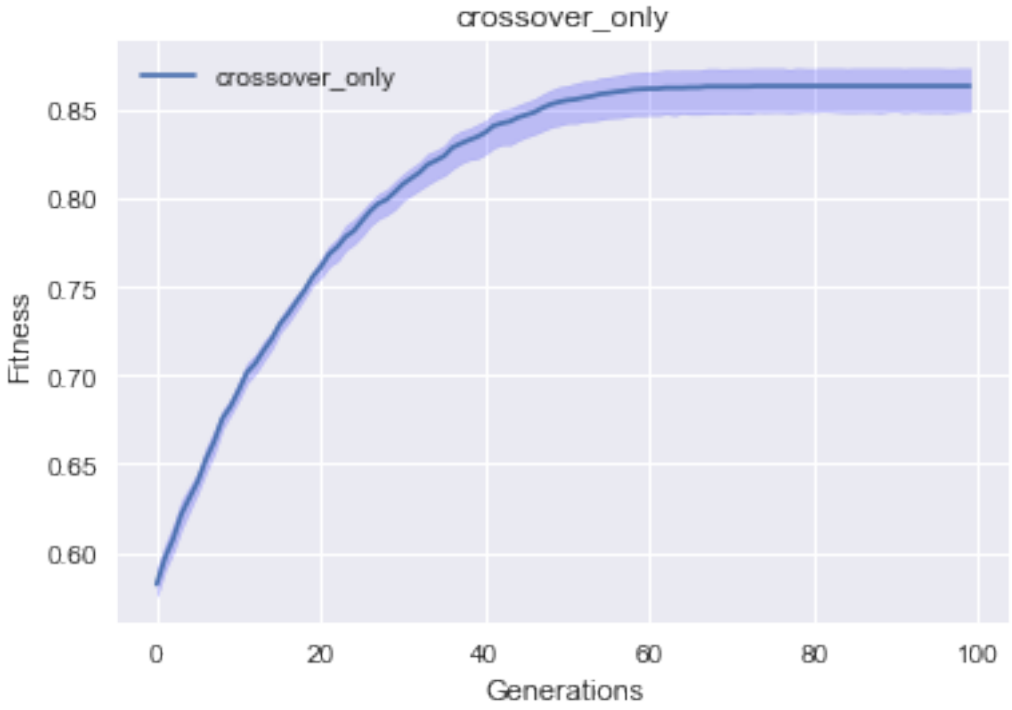
We will also modify our plotting scripts from before to show how fitness increases over generational time across these three treatments (with boostrapped confidence intervals as before). As we also did previously, please plot the three experimental conditions run above on the same figure for ease of comparisons.

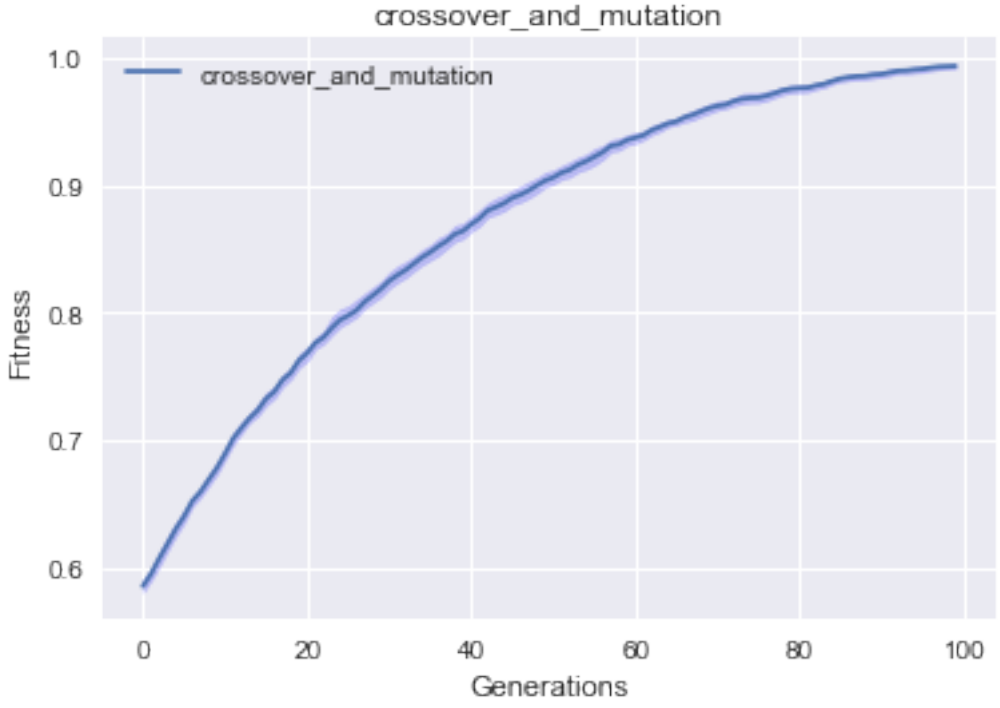
```
In [5]: def plot_mean_and_bootstrapped_ci_over_time(input_data = None, name = "change me", x_la
            """

            parameters:
            input_data: (numpy array of shape (generations, num_repitions)) solution metric to
            name: (string) name for legend
            x_label: (string) x axis label
            y_label: (string) y axis label

            returns:
            None
            """

            generations = input_data.shape[0]

            CIs = []
            mean_values = []
            for i in range(generations):
                mean_values.append(np.mean(input_data[i]))
                CIs.append(bootstrap.ci(input_data[i], statfunction=np.mean))
            mean_values=np.array(mean_values)

            print(CIs)
```

```
    high = []
    low = []
    for i in range(len(CIs)):
        low.append(CIs[i][0])
        high.append(CIs[i][1])

    low = np.array(low)
    high = np.array(high)
    fig, ax = plt.subplots()
    y = range(0, generations)
    ax.plot(y, mean_values, label=name)
    ax.fill_between(y, high, low, color='b', alpha=.2)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.legend()
    if (name) and len(name)>0:
        ax.set_title(name)


plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
```

```
[array([0.577 , 0.5925]), array([0.57925, 0.59425]), array([0.58275, 0.59775]), array([0.58525
[array([0.5755, 0.5915]), array([0.5905, 0.605 ]), array([0.6005, 0.616 ]), array([0.61375, 0.6
[array([0.58025, 0.591  ]), array([0.59 , 0.599]), array([0.601  , 0.61225]), array([0.61325, 0
```




crossover_only
crossover_only
0.85
0.80
0.75
0.70
0.65
0.60
Fitness
0
20
40
60
80
100
Generations


crossover_and_mutation
crossover_and_mutation
1.0
0.9
0.8
0.7
0.6
Fitness
0
20
40
60
80
100
Generations

1.0.5 Q4: Analysis of Crossover

Is crossover effective on this problem? How does crossover compare to mutation? How do the two interact?

Crossover appears very effective compared to mutation on this problem. Crossover is able to converge to a solution much faster than mutation probably due to its ability to make larger changes and explore the fitness landscape a little better. When the two interact together, crossover is able to make larger changes that explore the fitness landscape some, while mutations make smaller changes allowing for more of the landscape (potentially outside of the alleles present in the population) to be explored. This allows them to find very close to the global optimum on this problem within the generations allowed.

1.0.6 Q5: Propose and Implement a New Crossover Modification

We've implemented one specfic type of crossover (two-point crossover with two randomly chosen parents). What other variatons to crossover might you consider? Describe it in the box below, and what you anticipate as the effects of it (positive or negative).

I propose uniform crossover. Uniform crossover involves traversing over each gene in the genome and with some probability choosing the first parent's or second parent's allele for it. In this case, we will choose each parent's allele with a 50% probability. I anticipate that the effects will be similar to what we see above in the crossover + mutation graph with possibly a bit of improvement. My reasoning is that this type of crossover has the ability to explore more of the space of existing genomes since it doesn't just take a slice, but can choose a genome from either parent with a given probability.

1.0.7 Q5b: Let's test it!

Copy your evoluationary_algoirthm code and modify it to include your new experimental treatment. Run and visualize this treatment as above. Feel free to alo pull in any statistical test scripts/functions from last week, should that help you to analyze and compare this new approach.

```
In [6]: def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=1
            """ Evolutinary Algorithm (copied from the basic hillclimber in our last assignmen

                parameters:
                fitness_function: (callable function) that return the fitness of a genome
                                  given the genome as an input parameter (e.g. as defined in
                total_generations: (int) number of total iterations for stopping condition
                num_parents: (int) the number of parents we downselect to at each generation (
                num_children: (int) the number of children (note: parents not included in this
                bit_string_length: (int) length of bit string genome to be evoloved
                num_elements_to_mutate: (int) number of alleles to modify during mutation (0 =
                crossover (bool): whether to perform crossover when generating children

                returns:
```

```
    fitness_over_time: (numpy array) track record of the top fitness value at each
"""

# initialize record keeping
fitness_over_time = []

# the initialization proceedure
parents = []
children = []
for i in range(num_parents):
    parents.append(Individual(fitness_function, bit_string_length))

# get population fitness
for i in range(num_parents):
    parents[i].eval_fitness()

for i in range(total_generations): # repeat

    # the modification procedure
    children = []

    # inheretance
    if not crossover:
        children = copy.deepcopy(parents)

    # crossover
    if crossover:
        while len(children)<num_children:
            random.shuffle(parents)
            for j in range(0, num_parents, 2):

                # select both parents
                first_parent = parents[j]
                second_parent = parents[j+1]

                # randomly select two points for crossover
                crossover_points = random.sample(range(0,bit_string_length), 2)

                # Do we want to disallow crossover from higher to lower numbers?
                # TODO: Remove below line and edit crossover to work when higher n
                crossover_points = sorted(crossover_points)

                # switch the tail end of the genomes at this point between the two
                child1 = Individual(fitness_function, bit_string_length)
                child2 = Individual(fitness_function, bit_string_length)
                for k in range(bit_string_length):
                    child1.genome[k]=parents[random.randint(j,j+1)].genome[k]
                    child2.genome[k]=parents[random.randint(j,j+1)].genome[k]
```

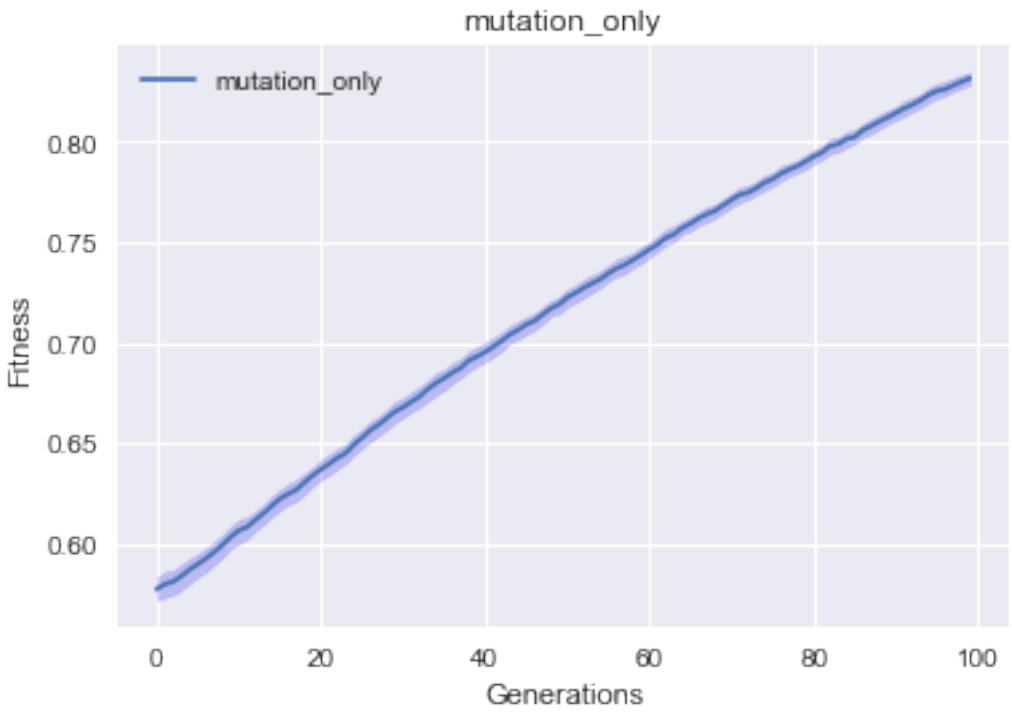
```
                        child1.eval_fitness
                        child2.eval_fitness

                        # Or do we just keep the children no matter what
                        children.append(child1)
                        children.append(child2)


                if len(children) > num_children:
                    children = children[:num_children]

            # mutation
            if num_elements_to_mutate > 0:
                # loop through each child
                for j in range(num_children):

                    # create storage array for elements to mutate
                    elements_mutated = []
                    elements_mutated = random.sample(range(0,bit_string_length), num_eleme

                    # loop through the array of indices to be mutated
                    for k in range(len(elements_mutated)):
                        # set the bit to the opposite of what it currently is
                        if children[j].genome[elements_mutated[k]] == 0:
                            children[j].genome[elements_mutated[k]] = 1
                        else:
                            children[j].genome[elements_mutated[k]] = 0

                    children[j].eval_fitness()


            # the assessement procedure
            for j in range(num_children):
                if children[j].fitness == 0 or children[j].fitness == None:
                    children[j].eval_fitness()

            # selection procedure
            parents += children.copy()
            parents = sorted(parents, key=lambda x: x.fitness, reverse=True)
            parents = parents[:num_parents]

            # record keeping
            fitness_over_time.append(parents[0].fitness)

        return fitness_over_time
```

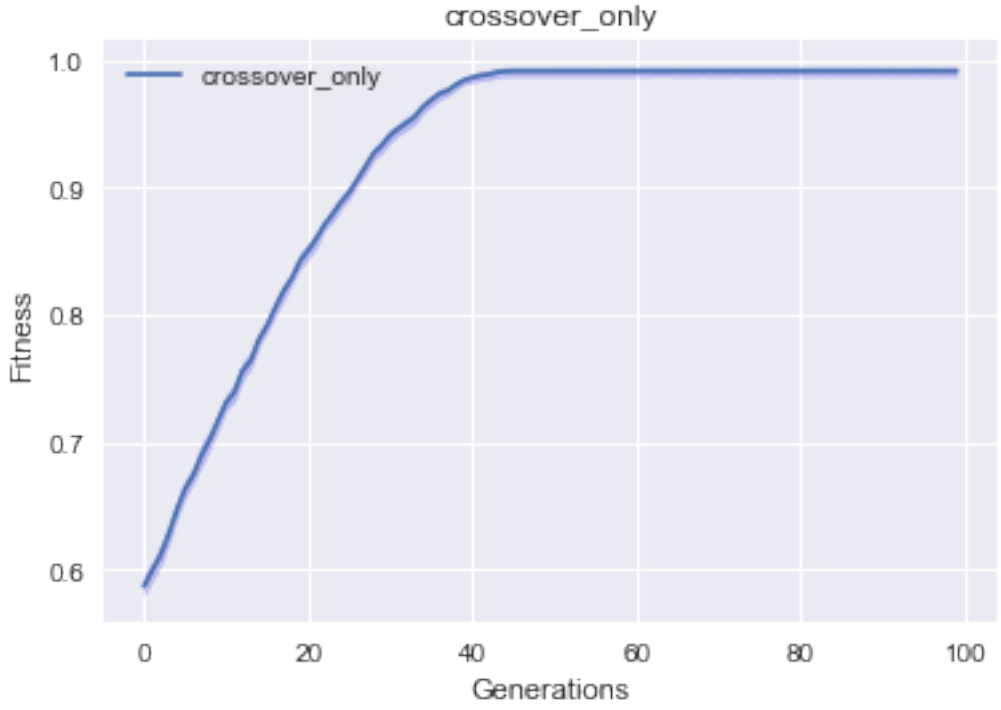
In [7]:
```
# experimentation
# num_runs = 20
```

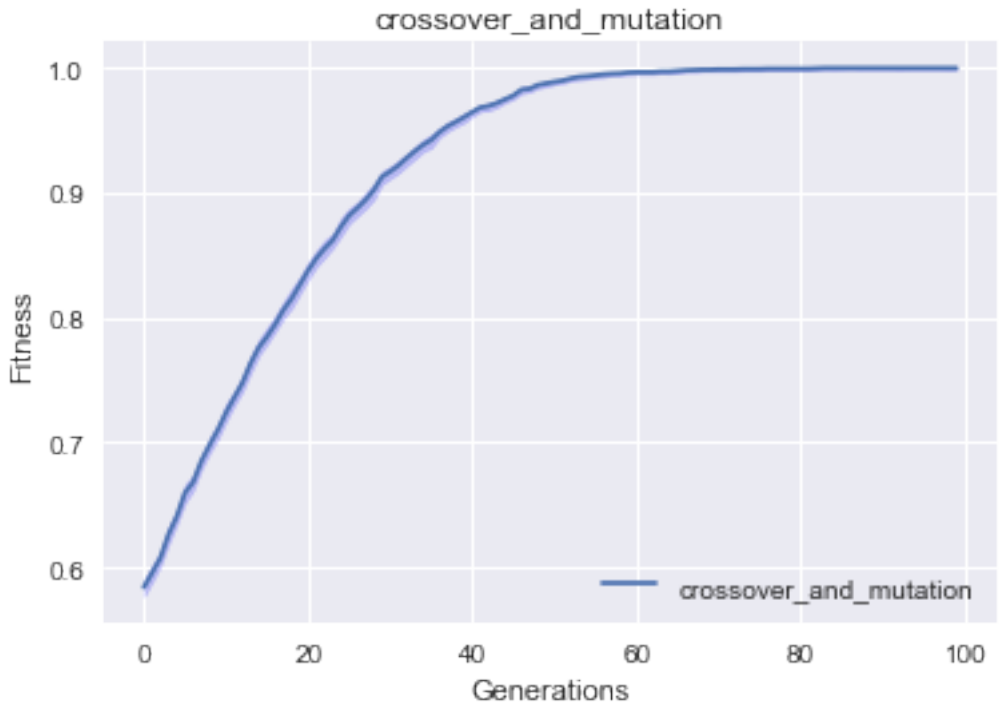
```
        # total_generations = 100
        # num_elements_to_mutate = 1
        # bit_string_length = 200
        # num_parents = 50
        # num_children = 50
        # fitness_function = np.mean
        # experiment_results = {}

        experiment_results['mutation_only'] = []
        experiment_results['crossover_only'] = []
        experiment_results['crossover_and_mutation'] = []

        for i in range(num_runs):
            # mutation only
            m_only = evolutionary_algorithm(fitness_function, total_generations, num_parents, n
            experiment_results['mutation_only'].append(m_only)

            # crossover only
            c_only = evolutionary_algorithm(fitness_function, total_generations, num_parents, n
            experiment_results['crossover_only'].append(c_only)

            # both crossover and mutation
            c_and_m = evolutionary_algorithm(fitness_function, total_generations, num_parents,
            experiment_results['crossover_and_mutation'].append(c_and_m)
```

In [8]:
```
# visualization
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_res
```

```
[array([0.5715, 0.583 ]), array([0.573  , 0.58575]), array([0.57425, 0.5865 ]), array([0.57675
[array([0.58  , 0.5965]), array([0.592  , 0.60625]), array([0.6035, 0.6195]), array([0.62075, 0
[array([0.577, 0.591]), array([0.58875, 0.60225]), array([0.6015, 0.613 ]), array([0.61925, 0.6
```


mutation_only
mutation_only
0.80
0.75
0.70
0.65
0.60
Fitness
0
20
40
60
80
100
Generations


crossover_only
crossover_only
1.0
0.9
0.8
0.7
0.6
Fitness
0
20
40
60
80
100
Generations



1.0.8 Q6: Well... What happened?

Describe the effect of your approach. If it did not work out as expected, please hypotheize as to why this is the case. If it did work out well, please comment on how broadly you think this finding might apply (or in what experimental conditions you might expect to come to a different conclusion).

This worked out better than I thought it would. Crossover alone was able to find the optimal solution after about 50 generations. This means that the uniform crossover worked very well in maximizing sum of the bits. We have a decent chance to make fairly large jumps in our fitness landscape compared to 2-point crossover. This is because potentially, uniform crossover can produce a bit string of all ones on the first try, whereas, this isn't even a possibility for something like 2-point crossover in most cases. 2-point crossover can only work with a slice of the alleles that exist within the two parents that it is looking at. Uniform crossover can choose from either parent on any single bit allowing for more possibilities. This makes the uniform crossover a bit more exploratory in this regard, meaning that sometimes it will find better solutions. This fitness landscape may have been an advatageous place for uniform crossover because of it's simplicity. Interestingly, it looks like mutation + crossover does worse than just crossover. This, I believe, is due to the fact that as we get closer to a solution for this particular problem, mutation is more likely to change a 1 to a 0 than a 0 to a 1. This would push us slightly further from the optimal solution.

1.0.9 Q7: Implementing Tournament Selection

Aside from crossover, including populations also gives us the opportunity to explore alternate selection mechanisms. As mentioned in class, tournament selection is one of my go-to methods for parent selection, so let's implement it here. The tournament should rely on input parameters such as the `tournament_size` to determine how many solutions will compete in a given tournament or `num_tournament_winners` to determine how many individuals from each tournament will be selected to move on as parents of the next generation. Touraments can be selected from the population with or without replacement (specifically I'm referring to making sure all individuals appear in at least one tournament before any individual partakes in one for a second time), and here feel free to use whichever version is simpler for you to implement and understand (which I expect will be the case with replacement).

```
In [9]: def evolutionary_algorithm(fitness_function=None, total_generations=100, num_parents=10
            """ Evolutinary Algorithm (copied from the basic hillclimber in our last assignmen

                parameters:
                fitness_function: (callable function) that return the fitness of a genome
                                  given the genome as an input parameter (e.g. as defined in
                total_generations: (int) number of total iterations for stopping condition
                num_parents: (int) the number of parents we downselect to at each generation (
                num_children: (int) the number of children (note: parents not included in this
                bit_string_length: (int) length of bit string genome to be evoloved
                num_elements_to_mutate: (int) number of alleles to modify during mutation (0 =
                crossover (bool): whether to perform crossover when generating children

                returns:
                fitness_over_time: (numpy array) track record of the top fitness value at each
            """

            # initialize record keeping
            fitness_over_time = []

            # the initialization proceedure
            parents = []
            children = []
            for i in range(num_parents):
                parents.append(Individual(fitness_function, bit_string_length))

            # get population fitness
            for i in range(num_parents):
                parents[i].eval_fitness()

            for i in range(total_generations): # repeat

                # the modification procedure
                children = []

                # inheretance
```

```
if not crossover:
    children = copy.deepcopy(parents)

# crossover
if crossover:
    while len(children)<(num_children):
        random.shuffle(parents)
        for j in range(0, num_parents, 2):

            # select both parents
            first_parent = parents[j]
            second_parent = parents[j+1]

            # randomly select two points for crossover
            crossover_points = random.sample(range(0,bit_string_length), 2)

            # Do we want to disallow crossover from higher to lower numbers?
            # TODO: Remove below line and edit crossover to work when higher n
            crossover_points = sorted(crossover_points)

            # switch genes between these two points between the two individual
            child1 = Individual(fitness_function, bit_string_length)
            child1.genome = parents[j].genome[:crossover_points[0]] + parents[
            child2 = Individual(fitness_function, bit_string_length)
            child2.genome = parents[j+1].genome[:crossover_points[0]] + parents

            # Or do we just keep the children no matter what
            children.append(child1)
            children.append(child2)


    if len(children) > num_children:
        children = children[:num_children]

# mutation
if num_elements_to_mutate > 0:
    # loop through each child
    for j in range(len(children)):

        # create storage array for elements to mutate
        elements_mutated = []
        elements_mutated = random.sample(range(0,bit_string_length), num_eleme

        # loop through the array of indices to be mutated
        for k in range(len(elements_mutated)):
            # set the bit to the opposite of what it currently is
            if children[j].genome[elements_mutated[k]] == 0:
                children[j].genome[elements_mutated[k]] = 1
```

```
                else:
                    children[j].genome[elements_mutated[k]] = 0

        # the assessement procedure
        for j in range(num_children):
            children[j].eval_fitness()

        # selection procedure
        parents += children.copy()
        random.shuffle(parents)
        new_parents = []

        # loop for the number of parents that we want
        while len(new_parents) < num_parents:
            random.shuffle(parents)
            for j in range(0, len(parents), tournament_size):
                winners = []
                for k in range(0, tournament_size):
                    if len(parents) <= j+k:
                        winners.append(parents[random.randint(0,len(parents)-1)])
                    else:
                        winners.append(parents[j+k])
                winners = sorted(winners, key=lambda x: x.fitness, reverse=True)
                winners = winners[:num_tournament_winners]
                new_parents += winners

        parents = sorted(new_parents, key=lambda x: x.fitness, reverse=True)[:num_pare

        # record keeping
        fitness_over_time.append(parents[0].fitness)

    return fitness_over_time
```

1.0.10 Q8: Run and Plot

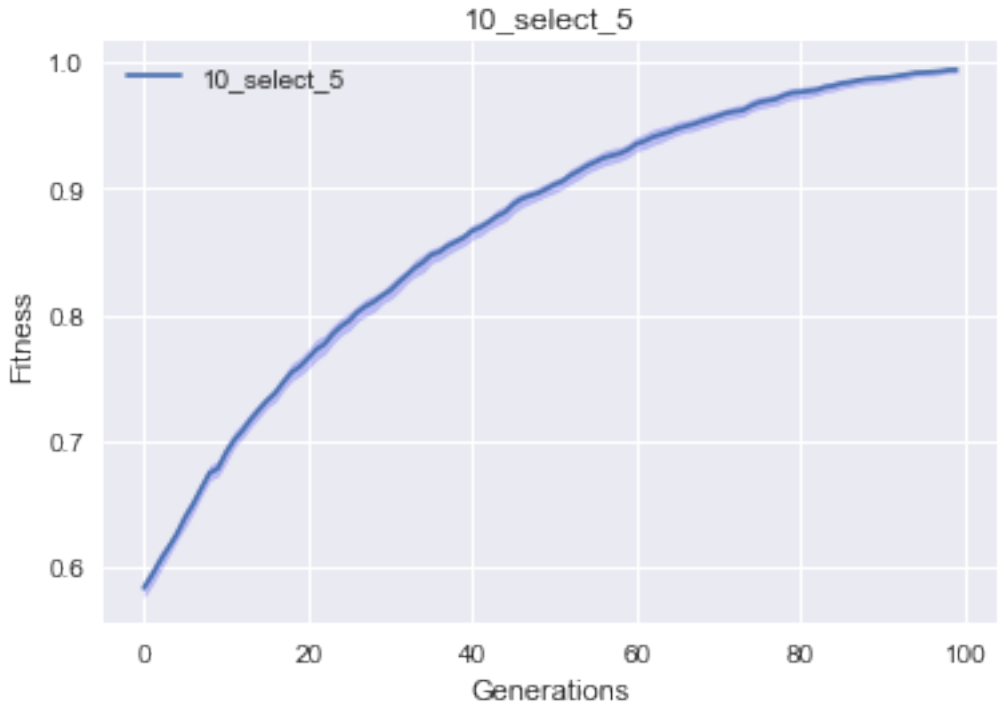
We discussed in class that the number of individuals participating in a tournament affects the amount of selection pressure it produces, presumably the same is true for the number of individuals selected from that tournament. So let's play around and generate some data to try and get to the bottom of it! In particular, let's run the following four experimental conditions: `10 select 5`, `20 select 10`, `20 select 5`, `50 select 10` (where the first number is how many individuals are in a tournament, and the second number is how many are selected from that tournament). Let's run these on the full-fledged evolutionary_algorithm including both mutation and crossover (for consistency and ease of grading please the original evolutionary algorithm implementation from `Q2` rather than your new implementation in `Q5` by either rerunning the prior code block, or by coping and pasting it in a new code block below). As above, please visualize the resulting fitnes over time and their boostrapped confidence intervals as well.

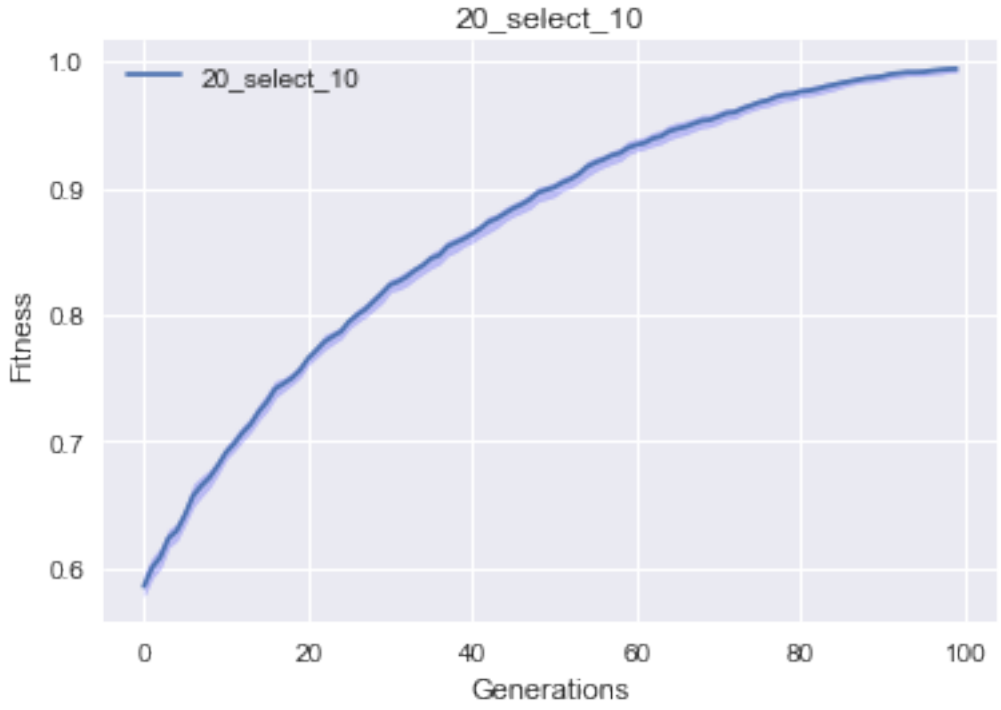
In [10]:
```
# if wanting to copy the original evolutionary_algorithm implementation here (e.g. so
...
```

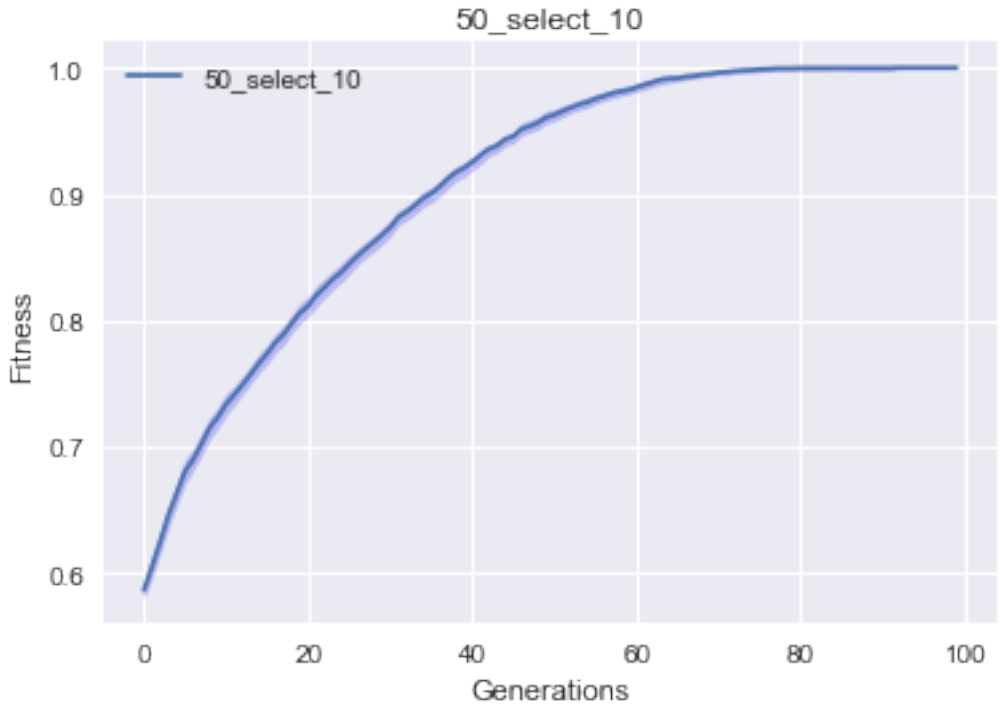
In [11]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
bit_string_length = 200
num_parents = 50
num_children = 50
fitness_function = np.mean

experiment_results['10_select_5'] = []
experiment_results['20_select_10'] = []
experiment_results['50_select_10'] = []

for i in range(num_runs):
    # 10 select 5
    ten_select_5 = evolutionary_algorithm(fitness_function, total_generations, num_pa
    experiment_results['10_select_5'].append(ten_select_5)

    # 20 select 10
    twenty_select_10 = evolutionary_algorithm(fitness_function, total_generations, nu
    experiment_results['20_select_10'].append(twenty_select_10)

    # both crossover and mutation
    fifty_select_10 = evolutionary_algorithm(fitness_function, total_generations, num
    experiment_results['50_select_10'].append(fifty_select_10)
```

In [12]:
```
# plotting
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_re
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_re
plot_mean_and_bootstrapped_ci_over_time(input_data=np.transpose(np.array(experiment_re
```

```
[array([0.577  , 0.59025]), array([0.587, 0.6  ]), array([0.59875, 0.61225]), array([0.61025, 0
[array([0.579  , 0.59625]), array([0.59325, 0.60925]), array([0.601  , 0.61675]), array([0.617
[array([0.582 , 0.5925]), array([0.59825, 0.61375]), array([0.6185, 0.6325]), array([0.63825, 0
```


10_select_5
10_select_5
1.0
0.9
0.8
0.7
0.6
Fitness
0
20
40
60
80
100
Generations


20_select_10
20_select_10
1.0
0.9
0.8
0.7
0.6
Fitness
0
20
40
60
80
100
Generations



1.0.11 Q9: Analysis

What do these results suggest matter about the values of the tournnament size and the number of winners selected? Is this suprising?

It looks like the smaller the ratio is between the tournament size and the number of winners, the more exploitative it is. I assume this due to the simplicity of the search space for this problem and the graphs presented. The two graphs that have a 2 to 1 ratio for these variables doe very similarly in performance, but the one that has a 5 to 1 ratio converges much faster to the optimal solution. There seems to only be a slight difference in the two that have a 2 to 1 ratio with the one having a smaller tournament size having a bit more exploration judging by the confidence interval line thickness.

1.0.12 Q10: Future Work

Again, we've just scratched the tip of the iceberg in terms of understanding or efficiently employing populations in evolutionary algorithms. If you were to run one more experiment here (i.e. another question in this assignment) what would you test next? If you were to deeply investigate some phenomenon around populations/selection/crossover (i.e. spend 6 weeks on a course project) what broader topic might you dig into?

I would want to play around with the fitness landscape / problem space. I think that having such a simple fitness landscape makes it difficult to understand the effects of these different selection and variation techniques. We can see some of this if we look back at the last assignment. If I was to deeply investigate some phenomenon herein for a number of weeks, I think I would want to investigate selection and/or crossover when applied to a dynamic fitness

landscape. What might be the best technique(s) to use in a scenario where our solutions are coevolving along with the fitness landscape. Should we use a dynamic technique for the dynamic landscapes? Is there a "best" way to go about it? Does it depend on the coevolution that is happening?

1.0.13 Congratulations, you made it to the end!

Nice work -- and hopefully you're starting to get the hang of these!

Please save this file as a .ipynb, and also download it as a .pdf, uploading **both** to blackboard to complete this assignment.

For your submission, please make sure that you have renamed this file (and that the resulting pdf follows suit) to replce `[netid]` with your UVM netid. This will greatly simplify our grading pipeline, and make sure that you receive credit for your work.

Academic Integrity Attribution During this assignment I collaborated with:

Just me for the most part. I asked to see Alican's graphs to compare mine to them, but that was about it.

In [ ]: