miniProject

December 8, 2020

[1]:
```python
import numpy
import random
random.seed()
```

[2]:
```python
Nd = 9

class Population(object):
    """ A set of candidate solutions to the Sudoku puzzle. These candidates are␣
↪also known as the chromosomes in the population. """

    def __init__(self):
        self.candidates = []
        return

    def seed(self, Nc, given):
        self.candidates = []

        # Determine the legal values that each square can take.
        helper = Candidate()
        helper.values = [[[] for j in range(0, Nd)] for i in range(0, Nd)]
        for row in range(0, Nd):
            for column in range(0, Nd):
                for value in range(1, 10):
                    if((given.values[row][column] == 0) and not (given.
↪is_column_duplicate(column, value) or given.is_block_duplicate(row, column,␣
↪value) or given.is_row_duplicate(row, value))):
                        # Value is available.
                        helper.values[row][column].append(value)
                    elif(given.values[row][column] != 0):
                        # Given/known value from file.
                        helper.values[row][column].append(given.
↪values[row][column])
                        break

        # Seed a new population.
        for p in range(0, Nc):
            g = Candidate()
```

```python
            for i in range(0, Nd): # New row in candidate.
                row = numpy.zeros(Nd)

                # Fill in the givens.
                for j in range(0, Nd): # New column j value in row i.

                    # If value is already given, don't change it.
                    if(given.values[i][j] != 0):
                        row[j] = given.values[i][j]
                    # Fill in the gaps using the helper board.
                    elif(given.values[i][j] == 0):
                        row[j] = helper.values[i][j][random.randint(0, len(helper.values[i][j])-1)]

                # If we don't have a valid board, then try again. There must be no duplicates in the row.
                while(len(list(set(row))) != Nd):
                    for j in range(0, Nd):
                        if(given.values[i][j] == 0):
                            row[j] = helper.values[i][j][random.randint(0, len(helper.values[i][j])-1)]

                g.values[i] = row

            self.candidates.append(g)

        # Compute the fitness of all candidates in the population.
        self.update_fitness()
        print("Seeding complete.")
        return

    def update_fitness(self):
        """ Update fitness of every candidate/chromosome. """
        for candidate in self.candidates:
            candidate.update_fitness()
        return

    def sort(self):
        """ Sort the population based on fitness. """
        self.candidates.sort(reverse = True,key = lambda x: x.fitness)
        return

    def sort_fitness(self, x, y):
        """ The sorting function. """
        if(x.fitness < y.fitness):
            return 1
        elif(x.fitness == y.fitness):
```

```
            return 0
        else:
            return -1
```

[3]:
```
class Candidate(object):
    """ A candidate solutions to the Sudoku puzzle. """
    def __init__(self):
        self.values = numpy.zeros((Nd, Nd), dtype=int)
        self.fitness = None
        return

    def update_fitness(self):
        """ The fitness of a candidate solution is determined by how close it␣
↪is to being the actual solution to the puzzle. The actual solution (i.e. the␣
↪'fittest') is defined as a 9x9 grid of numbers in the range [1, 9] where␣
↪each row, column and 3x3 block contains the numbers [1, 9] without any␣
↪duplicates (see e.g. http://www.sudoku.com/); if there are any duplicates␣
↪then the fitness will be lower. """

        row_count = numpy.zeros(Nd)
        column_count = numpy.zeros(Nd)
        block_count = numpy.zeros(Nd)
        row_sum = 0
        column_sum = 0
        block_sum = 0

        for i in range(0, Nd):  # For each row...
            for j in range(0, Nd):  # For each number within it...
                row_count[self.values[i][j]-1] += 1  # ...Update list with␣
↪occurrence of a particular number.

            row_sum += (1.0/len(set(row_count)))/Nd
            row_count = numpy.zeros(Nd)

        for i in range(0, Nd):  # For each column...
            for j in range(0, Nd):  # For each number within it...
                column_count[self.values[j][i]-1] += 1  # ...Update list with␣
↪occurrence of a particular number.

            column_sum += (1.0 / len(set(column_count)))/Nd
            column_count = numpy.zeros(Nd)


        # For each block...
        for i in range(0, Nd, 3):
            for j in range(0, Nd, 3):
                block_count[self.values[i][j]-1] += 1
```

```
                block_count[self.values[i][j+1]-1] += 1
                block_count[self.values[i][j+2]-1] += 1

                block_count[self.values[i+1][j]-1] += 1
                block_count[self.values[i+1][j+1]-1] += 1
                block_count[self.values[i+1][j+2]-1] += 1

                block_count[self.values[i+2][j]-1] += 1
                block_count[self.values[i+2][j+1]-1] += 1
                block_count[self.values[i+2][j+2]-1] += 1

                block_sum += (1.0/len(set(block_count)))/Nd
                block_count = numpy.zeros(Nd)

        # Calculate overall fitness.
        if (int(row_sum) == 1 and int(column_sum) == 1 and int(block_sum) == 1):
            fitness = 1.0
        else:
            fitness = column_sum * block_sum

        self.fitness = fitness
        return

    def mutate(self, mutation_rate, given):
        """ Mutate a candidate by picking a row, and then picking two values␣
↪within that row to swap. """

        r = random.uniform(0, 1.1)
        while(r > 1): # Outside [0, 1] boundary - choose another
            r = random.uniform(0, 1.1)

        success = False
        if (r < mutation_rate):  # Mutate.
            while(not success):
                row1 = random.randint(0, 8)
                row2 = random.randint(0, 8)
                row2 = row1

                from_column = random.randint(0, 8)
                to_column = random.randint(0, 8)
                while(from_column == to_column):
                    from_column = random.randint(0, 8)
                    to_column = random.randint(0, 8)

                # Check if the two places are free...
                if(given.values[row1][from_column] == 0 and given.
↪values[row1][to_column] == 0):
```

```
                    # ...and that we are not causing a duplicate in the rows'␣
↪columns.
                    if(not given.is_column_duplicate(to_column, self.
↪values[row1][from_column])
                       and not given.is_column_duplicate(from_column, self.
↪values[row2][to_column])
                       and not given.is_block_duplicate(row2, to_column, self.
↪values[row1][from_column])
                       and not given.is_block_duplicate(row1, from_column, self.
↪values[row2][to_column])):

                        # Swap values.
                        temp = self.values[row2][to_column]
                        self.values[row2][to_column] = self.
↪values[row1][from_column]
                        self.values[row1][from_column] = temp
                        success = True

        return success
```

[4]:

```
class Given(Candidate):
    def __init__(self, values):
        self.values = values
        return

    def is_row_duplicate(self, row, value):
        for column in range(0, Nd):
            if(self.values[row][column] == value):
               return True
        return False

    def is_column_duplicate(self, column, value):
        for row in range(0, Nd):
            if(self.values[row][column] == value):
               return True
        return False

    def is_block_duplicate(self, row, column, value):
        i = 3*(int(row/3))
        j = 3*(int(column/3))

        if((self.values[i][j] == value)
           or (self.values[i][j+1] == value)
           or (self.values[i][j+2] == value)
           or (self.values[i+1][j] == value)
           or (self.values[i+1][j+1] == value)
```

```
            or (self.values[i+1][j+2] == value)
            or (self.values[i+2][j] == value)
            or (self.values[i+2][j+1] == value)
            or (self.values[i+2][j+2] == value)):
             return True
        else:
            return False
```

[5]:
```
class Tournament(object):
    """ The crossover function requires two parents to be selected from the␣
↪population pool. The Tournament class is used to do this.

    Two individuals are selected from the population pool and a random number␣
↪in [0, 1] is chosen. If this number is less than the 'selection rate' (e.g.␣
↪0.85), then the fitter individual is selected; otherwise, the weaker one is␣
↪selected.
    """
    def __init__(self):
        return

    def compete(self, candidates):
        c1 = candidates[random.randint(0, len(candidates)-1)]
        c2 = candidates[random.randint(0, len(candidates)-1)]
        f1 = c1.fitness
        f2 = c2.fitness

        if(f1 > f2):
            fittest = c1
            weakest = c2
        else:
            fittest = c2
            weakest = c1

        selection_rate = 0.85
        r = random.uniform(0, 1.1)
        while(r > 1):
            r = random.uniform(0, 1.1)
        if(r < selection_rate):
            return fittest
        else:
            return weakest
```

[6]:
```
class CycleCrossover(object):
    """ Crossover relates to the analogy of genes within each parent candidate␣
↪mixing together in the hopes of creating a fitter child candidate. Cycle␣
↪crossover is used here """
```

```
    def __init__(self):
        return

    def crossover(self, parent1, parent2, crossover_rate):
        child1 = Candidate()
        child2 = Candidate()

        child1.values = numpy.copy(parent1.values)
        child2.values = numpy.copy(parent2.values)

        r = random.uniform(0, 1.1)
        while(r > 1):
            r = random.uniform(0, 1.1)

        if (r < crossover_rate):
            crossover_point1 = random.randint(0, 8)
            crossover_point2 = random.randint(1, 9)
            while(crossover_point1 == crossover_point2):
                crossover_point1 = random.randint(0, 8)
                crossover_point2 = random.randint(1, 9)

            if(crossover_point1 > crossover_point2):
                temp = crossover_point1
                crossover_point1 = crossover_point2
                crossover_point2 = temp

            for i in range(crossover_point1, crossover_point2):
                child1.values[i], child2.values[i] = self.crossover_rows(child1.
↪values[i], child2.values[i])

        return child1, child2

    def crossover_rows(self, row1, row2):
        child_row1 = numpy.zeros(Nd)
        child_row2 = numpy.zeros(Nd)

        remaining = [*range(1, Nd+1)]
        cycle = 0

        while((0 in child_row1) and (0 in child_row2)):
            if(cycle % 2 == 0):  # Even cycles.
                # Assign next unused value.
                index = self.find_unused(row1, remaining)
                start = row1[index]
                remaining.remove(row1[index])
                child_row1[index] = row1[index]
                child_row2[index] = row2[index]
```

```
                next = row2[index]

                while(next != start):  # While cycle not done...
                    index = self.find_value(row1, next)
                    child_row1[index] = row1[index]
                    remaining.remove(row1[index])
                    child_row2[index] = row2[index]
                    next = row2[index]

                cycle += 1

            else:  # Odd cycle - flip values.
                index = self.find_unused(row1, remaining)
                start = row1[index]
                remaining.remove(row1[index])
                child_row1[index] = row2[index]
                child_row2[index] = row1[index]
                next = row2[index]

                while(next != start):  # While cycle not done...
                    index = self.find_value(row1, next)
                    child_row1[index] = row2[index]
                    remaining.remove(row1[index])
                    child_row2[index] = row1[index]
                    next = row2[index]

                cycle += 1

        return child_row1, child_row2

    def find_unused(self, parent_row, remaining):
#         print(parent_row[2])
        for i in range(0, len(parent_row)):
            if(parent_row[i] in remaining):
                return i

    def find_value(self, parent_row, value):
        for i in range(0, len(parent_row)):
            if(parent_row[i] == value):
                return i
```

[7]:
```
class Sudoku(object):
    """ Solves a given Sudoku puzzle using a genetic algorithm. """

    def __init__(self):
        self.given = None
        return
```

```python
def load(self, path):
    with open(path, "r") as f:
        values = numpy.loadtxt(f).reshape((Nd, Nd)).astype(int)
        self.given = Given(values)
    return

def save(self, path, solution):
    with open(path, "w") as f:
        numpy.savetxt(f, solution.values.reshape(Nd*Nd), fmt='%d')
    return

def solve(self):
    Nc = 1000  # Number of candidates (i.e. population size).
    Ne = int(0.05*Nc)  # Number of elites.
    Ng = 1000  # Number of generations.
    Nm = 0  # Number of mutations.

    # Mutation parameters.
    phi = 0
    sigma = 1
    mutation_rate = 0.06

    # Create an initial population.
    self.population = Population()
    self.population.seed(Nc, self.given)

    stale = 0
    for generation in range(0, Ng):
        print("Generation %d" % generation)

        best_fitness = 0.0
        for c in range(0, Nc):
            fitness = self.population.candidates[c].fitness
            if(fitness == 1):
                print("Solution found at generation %d!" % generation)
                print(self.population.candidates[c].values)
                return self.population.candidates[c]

            if(fitness > best_fitness):
                best_fitness = fitness

        print("Best fitness: %f" % best_fitness)

        next_population = []
```

```
            # Select elites (the fittest candidates) and preserve them for the␣
↪next generation.
            self.population.sort()
            elites = []
            for e in range(0, Ne):
                elite = Candidate()
                elite.values = numpy.copy(self.population.candidates[e].values)
                elites.append(elite)

            # Create the rest of the candidates.
            for count in range(Ne, Nc, 2):
                t = Tournament()
                parent1 = t.compete(self.population.candidates)
                parent2 = t.compete(self.population.candidates)

                ## Cross-over.
                cc = CycleCrossover()
                child1, child2 = cc.crossover(parent1, parent2,␣
↪crossover_rate=1.0)

                # Mutate child1.
                child1.update_fitness()
                old_fitness = child1.fitness
                success = child1.mutate(mutation_rate, self.given)
                child1.update_fitness()
                if(success):
                    Nm += 1
                    if(child1.fitness > old_fitness):
                        phi = phi + 1

                # Mutate child2.
                child2.update_fitness()
                old_fitness = child2.fitness
                success = child2.mutate(mutation_rate, self.given)
                child2.update_fitness()
                if(success):
                    Nm += 1
                    if(child2.fitness > old_fitness):
                        phi = phi + 1

                # Add children to new population.
                next_population.append(child1)
                next_population.append(child2)

            # Append elites onto the end of the population. These will not have␣
↪been affected by crossover or mutation.
            for e in range(0, Ne):
```

```
                next_population.append(elites[e])

            # Select next generation.
            self.population.candidates = next_population
            self.population.update_fitness()

            # Calculate new adaptive mutation rate (based on Rechenberg's 1/5 success rule). This is to stop too much mutation as the fitness progresses towards unity.
            if(Nm == 0):
                phi = 0
            else:
                phi = phi / Nm

            if(phi > 0.2):
                sigma = sigma/0.998
            elif(phi < 0.2):
                sigma = sigma*0.998

            mutation_rate = abs(numpy.random.normal(loc=0.0, scale=sigma, size=None))
            Nm = 0
            phi = 0

            # Check for stale population.
            self.population.sort()
            if(self.population.candidates[0].fitness != self.population.candidates[1].fitness):
                stale = 0
            else:
                stale += 1

            # Re-seed the population if 100 generations have passed with the fittest two candidates always having the same fitness.
            if(stale >= 100):
                print("The population has gone stale. Re-seeding...")
                self.population.seed(Nc, self.given)
                stale = 0
                sigma = 1
                phi = 0
                Nm = 0
                mutation_rate = 0.06

        print("No solution found.")
        return None
```

[10]:
```python
s = Sudoku()
s.load("puzz.txt")
solution = s.solve()
if(solution):
    s.save("solution.txt", solution)
```

```
Seeding complete.
Generation 0
Best fitness: 0.308642
Generation 1
Best fitness: 0.308642
Generation 2
Best fitness: 0.349794
Generation 3
Best fitness: 0.349794
Generation 4
Best fitness: 0.473251
Generation 5
Best fitness: 0.473251
Generation 6
Best fitness: 0.473251
Generation 7
Best fitness: 0.536351
Generation 8
Best fitness: 0.536351
Generation 9
Best fitness: 0.536351
Generation 10
Best fitness: 0.599451
Generation 11
Best fitness: 0.599451
Generation 12
Best fitness: 0.599451
Generation 13
Best fitness: 0.662551
Generation 14
Best fitness: 0.777778
Generation 15
Best fitness: 0.777778
Generation 16
Best fitness: 0.851852
Generation 17
Best fitness: 0.851852
Generation 18
Best fitness: 0.851852
Generation 19
Solution found at generation 19!
```

```
[[8 3 9 2 7 4 6 5 1]
 [5 2 4 1 3 6 7 8 9]
 [7 6 1 5 8 9 4 2 3]
 [1 9 7 8 5 3 2 6 4]
 [6 8 3 4 9 2 5 1 7]
 [2 4 5 6 1 7 9 3 8]
 [3 5 2 7 4 1 8 9 6]
 [9 7 8 3 6 5 1 4 2]
 [4 1 6 9 2 8 3 7 5]]
```

[ ]:

```

```