UNIVERSITY OF PRETORIA

COS 314

Project 2 (Genetic Algorithm)

Author
Kyle WOOD - u16087993
u16087993@tuks.co.za

May 23, 2018

Contents

Installation:

Software Requirements

- Python Recommended version 2.7.x
- Tkinter Recommended version 8.6
- OpenCV

Installation steps:

To test if your system has all requirements run the following script:
Run: $ python systemTest.py

If all the tests pass then the program should be able to run correctly, Otherwise follow the following installation instructions:

Try run (install.sh)
1. Type: $ sudo chmod +x install.sh
2. Type: $ ./install.sh
If install.sh fails to install then visit the following resources:
Python: `http://docs.python-guide.org/en/latest/starting/install/linux/`
Tkinter: `https://www.techinfected.net/2015/09/how-to-install-and-use-tkinter-in-ubuntu-debia`
`html`
OpenCV: `https://docs.opencv.org/3.4.1/d2/de6/tutorial_py_setup_in_`
`ubuntu.html`

Running the program

Once your system is able to run systemTest.py without errors, you should now be able to run the program using the following command:

To run program:
$ python mosaic.py "./imageLibraryPath" "imagePath" NumRows NumCols

The program will initialise and create the necessary folders and resize and create a new scaled image library in directory imageLib. Once this is completed the GUI will display. May take some time to show GUI due to image library creation

Options available:
1. Population Size
2. Tournament Size
3. Cross-over Probability
4. Mutation Probability
5. Mutation Size

Click the run button to start the GA with the settings you chose, you can stop at any time by clicking the stop button, this will save the final image as shown in the directory structure below, simply click open folder to view it.

Directory Setup

The following directory structure has been implemented in this project:

```
Main folder
├── Project lib *(Generated)
│   ├── image lib *(Scaled image library)
│   ├── Output
│   │   ├── Generations
│   │   └── filenameMosaic.jpg *(Final Output after n generations)
│   └── Segmented Image *(Original image broke into segments)
├── Documentation
├── Collection of Mosaics
│   ├── Original.jpg
│   ├── generated.jpg
│   └── settings.txt
├── Image library (Supplied by user)
├── install.sh *(Install necessary libraries)
├── systemTest.py *(Test configuration of system)
├── mosaic.py *(Program)
├── proj2support.py *(support program)
└── OriginalImage.jpg *(Supplied by user)
```

Design Choices:

Chromosomes:

Each chromosome is most often represented as a 2D matrix where each element in the matrix is an output pixel (Image from image library) which makes up the final image.

Initial Generation (Gen 0):

This creates a population of *PopSize* chromosomes where each chromosome is created by randomly choosing an element in the image library and placing it in the chromosome until an image is created.

Selection:

Tournament Selection is implemented as a default, it works by for the tournament size it loops and randomly each loop selects an element from the population, it then compares the element based on its fitness to the previously best found element, if it has a lower fitness value (smaller average distance) which is better in this implementation then it becomes the fittest element found so far. After the loop has completed we return the winner (element in the tournament with the best fitness) .

```python
def tournamentSelection(self,pop, popFitnesses,tournamentsize):
        index  = randint(0,len(pop)-1)
        bestFitness = popFitnesses[index]
        bestIndx= index
        i=1
        for i in range (0,tournamentsize):
            index  = randint(0,len(pop)-1)
            if (popFitnesses[index] < bestFitness):
                bestIndx = index
                bestFitness = popFitnesses[index]
        return bestIndx
```

Cross-over:

Cross-over has been implemented as a variation of two-point cross-over. Firstly a random choice decides whether the cross over will be vertical or horizontal, meaning that the cross over will ether only affect columns (if vertical) or rows (if horizontal). Once this choice has been made the two offspring are initialised, offspring1 = parent1 and offspring2 = parent2, we then randomly choose two rows/columns and then replace those rows/columns selected with the same rows/columns of the opposite parent (ie. crossing over the rows/columns selected of parent1 with offspring2 (which was based on parent2 and the opposite for offspring 2). Once the two offspring are created they are then returned. The parents used in the cross-over are chosen using 2 tournament selections, also cross over only occurs if a randomly chosen operatorProb (in newGeneration creation method) is less than or equal to the cross-over probability selected by the user.

```
def crossOver (self,p1, p2):
        #choose horizontal vs vertical crossover
        operator = randint(0, 1)
        offspring1 = deepcopy(p1)
        offspring2 = deepcopy(p2)
        if (operator == 0):  # vertical crossover
            crossOverPoint1 = randint(0, cols-1)
            crossOverPoint2 = randint(0, cols-1)
            while crossOverPoint2 == crossOverPoint1:
                crossOverPoint2 = randint(0, cols-1)

            #swap columns of p1 and p2 to create 2 offspring
            offspring1 [crossOverPoint1,:] = p2[crossOverPoint1,:]
            offspring1 [crossOverPoint2,:] = p2[crossOverPoint2,:]
            offspring2 [crossOverPoint1,:] = p1[crossOverPoint1,:]
            offspring2 [crossOverPoint2,:] = p1[crossOverPoint2,:]
        else:  # horizontal crossover
            crossOverPoint1 = randint(0, rows-1)
            crossOverPoint2 = randint(0, rows-1)

            while crossOverPoint2 == crossOverPoint1:
                crossOverPoint2 = randint(0, rows-1)

            #swap rows of p1 and p2 to create 2 offspring
            offspring1 [:,crossOverPoint1] = p2[:,crossOverPoint1]
            offspring1 [:,crossOverPoint2] = p2[:,crossOverPoint2]
            offspring2 [:,crossOverPoint1] = p1[:,crossOverPoint1]
            offspring2 [:,crossOverPoint2] = p1[:,crossOverPoint2]
        output = [offspring1,offspring2]
        return output
```

Mutation:

Mutation loops for the a random number of times between zero and the chosen mutation size, for each iteration the loop a random row and column index are chosen (ie a random pixel in the image is chosen) and replaced with a randomly selected new image from the image library. The offspring is then returned. Mutation only occurs if the randomly chosen operatorProb (in newGeneration creation method) is less than or equal to the mutation probability selected by the user.

```
def mutation (self,p1):
        #choose horizontal vs vertical crossover
        mutationAmt = randint(0,mutationAmount)
        i = 0
        offspring = deepcopy(p1)
        # choose and place n random images
        for i in range(0,mutationAmt):
            xIndx = randint(0,cols-1)
            yIndx = randint(0,rows-1)
```

```
          shuffle(imageLib)
          offspring[xIndx][yIndx]= imageLib[0]
      return offspring
```

Creation of new populations:

To create a new generation, the entire old population of *popSize* is copied over the the new generation, then *popSize* additional new chromosomes are created by randomly choosing an operator and operator probability to decide whether an operator will be used. Operators implemented (Crossover, Mutation and Reproduction) parents for these new offspring are chosen using tournament selection. After the additional chromosomes are created, the resulting population is of size 2*popSize, the population is then sorted according to their fitness values and the top popSize chromosomes become the new generation, this new generation is then returned. The chromosome with the best fitness in this population is then shown on the GUI.

```
#Creates a new generation of chromosomes
  def createNewGeneration(self,genNumber,size,pop,popFitness,reproductionProb):
      global population,fitnessPopulation

      #Copy all elements from old population to new population
      newPop = deepcopy(pop)


      while len(newPop) < 2*len(pop):
          #select parents using tournament selection
          parent1 = newPop[self.tournamentSelection(pop,popFitness,tournament_size)]
          #choose an operator and prob randomly
          operator = randint(1,3)
          operatorProb = randint(0,100)/100.0

          if (operator == 0): #crossover
              if (operatorProb <= crossOverProb):
                  parent2 =newPop[ self.tournamentSelection(pop,popFitness,tournament_size)]
                  offspring = self.crossOver (parent1,parent2)
                  newPop.append(offspring[0])
                  newPop.append(offspring[1])
          elif (operator ==1 ): #mutation
              if (operatorProb <= mutationProb):
                  #choose chromosome with worst fitness and mutate that
                  offspring = self.mutation(parent1)
                  newPop.append(offspring)
          else: # randomly create new image
              if (operator <= reproductionProb):
                  newPop.append(self.createRandomImage(imageOrig))
                  #newPop.append(population[0])

      #calculate fitness of population, sort population by fitness
```

```
        fitnessPopulation = np.array(self.fitness(imageOrig,newPop,
    segImage))
        sortedFitnessIndx = np.argsort(fitnessPopulation)
        newPop = np.array(newPop)
        newPop[sortedFitnessIndx]
        fitnessPopulation =fitnessPopulation[sortedFitnessIndx]
        newPop = newPop[sortedFitnessIndx]

        #New gen = top pop_size elements of population (reduce size
    back to pop_size)
        i=0
        outputPop =[]
        for i in range(0,pop_size):
            outputPop.append(deepcopy(newPop[i]))
        imageCount =0
        for chromosome in outputPop:
            img = self.createImageFromMatrix(chromosome)
            #cv2.imwrite(os.path.join(outputFolder,'chromosome'+str(
    imageCount)+'.jpg'),img)
            imageCount+=1
        cv2.imwrite(os.path.join(outputFolder,'gen'+str(genNumber)+'.
    jpg'),self.createImageFromMatrix( outputPop[0]))
        print("Gen "+str(genNumber)+" created")
        return outputPop
```

Fitness function:

The fitness function implemented makes use of the Riemersma's formula which will calculates the distance of one pixel to the appropriate location of the corresponding spot on original image by adding up the squares of the distances of every sub-pixel to the spot. The total distance is then divided by the dimensions of the image to give the overall average of the generated image compared to the original image. The objective is to try reduce this to be as small as possible.

```
def calculateDifference(self,newVector, origVector):
        rnew = newVector[0]
        bnew = newVector[1]
        gnew = newVector[2]

        rorig =origVector[0]
        borig =origVector[1]
        gorig =origVector[2]

        rD = (rnew+rorig)/2.0
        R = rorig- rnew
        G = gnew-gorig
        B = bnew-borig
        # Riemersma difference
        dif = math.sqrt((2+(rD/256))*math.pow(R,2)+4*math.pow(G,2)
    +(2+((255-rD)/256))*math.pow(B,2))
        #dif = math.sqrt (math.pow(R,2)+math.pow(G,2)+math.pow(B,2))
        return dif

```

```
def fitness (self,imageOrig,population,segImage):
        fitnesses=[]
        i =0
        for chromosome in population:
            distances =0
            i=0
            for i in range(0,rows):
                j=0
                for j in range (0,cols):
                    img = self.createImageFromMatrix(chromosome)
                    avgColorNew = np.array(chromosome[i][j]).mean(axis
    =(0,1))
                    avgColorOrig = np.array(segImage[i][j]).mean(axis
    =(0,1))
                    distances += self.calculateDifference(avgColorNew,
    avgColorOrig)
            amount = rows*cols
            average_dist = distances/amount
            fitnesses.append (average_dist)

        return fitnesses
```