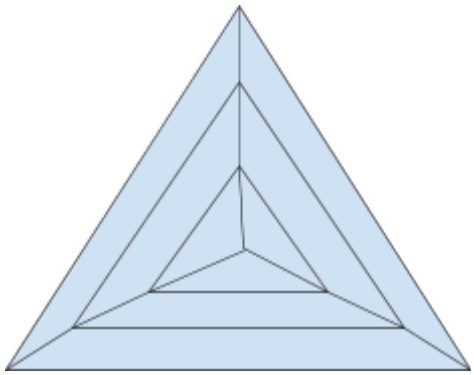
Ryan Tanaka
September 17, 2018
ICS674

Using Genetic Algorithms for Layouts of Planar Graphs

Idea. Given a planar graph, can a genetic algorithm find a layout such that no (or only a minimum) number of edges intersect? For this project, I used a simple planar graph depicted below. It contains 10 nodes and 18 links. Each of the nodes can be placed in any of the points within an 800x800 pixel SVG element. This means that each node could be in one of 800^2 positions, so there are $(800^2)^{10}$ possible layouts since nodes may end up on "top" of each other. Edges are limited to straight edges.



Search Space. Vector of 10 (x,y) coordinates.

Objective Function. The number of edge pairs in the graph is used to determine the fitness of an individual. For each individual, the code checks each pair of edges (which amounts to $\frac{num\ edges!}{2(2 - num\ edges)!}$), then does some checks to compute the orientation of those points with respect to each other to see if they intersect.

```
def compute_fitness(individual):
   link_crossings = 0
   link_pairs = combinations(links, 2)
   num_link_pairs = len(list(combinations(links, 2)))

   for link_pair in link_pairs:
       if (do_links_intersect(
               link_pair[0], individual.chromosome[link_pair[0].source],
individual.chromosome[link_pair[0].target],
               link_pair[1], individual.chromosome[link_pair[1].source],
individual.chromosome[link_pair[1].target])):
           link_crossings += 1

   # the number of pairs of edges in the graph that don't intersect
```

```
    individual.fitness = num_link_pairs - link_crossings
```

Recombination. The recombination operator was implemented exactly as it was in the slides from our initial lecture. This uses single point crossover, where the crossover point is chosen randomly. Another detail about this variation operator is that everyone gets an equal chance to for an offspring together. This could have been done by giving preference to those with higher fitness scores.

```
def recombination(population):

    recombination_rate = 0.5
    population_xo = [None] * len(population)

    for index, individual in enumerate(population):
        if random.random() > recombination_rate:
            offspring = individual

        else:
            offspring = individual
            parent_2_index = random.randint(0, len(population) - 1)

            parent_2 = population[parent_2_index]
            xo_point = random.randint(0, len(individual.chromosome) - 1)

            for i in range(len(offspring.chromosome)):
                if i > xo_point:
                    offspring.chromosome[i].x = parent_2.chromosome[i].x
                    offspring.chromosome[i].y = parent_2.chromosome[i].y

        population_xo[index] = offspring

    return population_xo
```

Mutation. The mutation operator was taken directly from the slides. The only difference here is that instead of changing an alphabet, the mutation operator changes the (x,y) coordinates of a single unit in the chromosome.

```
def mutation(population):

    mutation_rate = 0.05
    population_mut = [None] * len(population)

    for index, individual in enumerate(population):
```

```
        if random.random() > mutation_rate:
            offspring = deepcopy(individual)
        else:
            offspring = deepcopy(individual)
            locus = random.randint(0, len(individual.chromosome) - 1)
            offspring.chromosome[locus].x = random.randint(0, CANVAS_DIMENSION)
            offspring.chromosome[locus].y = random.randint(0, CANVAS_DIMENSION)

        population_mut[index] = deepcopy(offspring)

    return population_mut
```

Selection. Fitness proportional selection was used for this implementation. The higher an individual's fitness, the higher the probability that it is selected to take part in the next generation. Those with low fitness scores have a non-zero chance of continuing on as well.

```
def selection(population):

    population_sel = [None] * len(population)

    fitness_sum = sum([individual.fitness for individual in population])

    for index, individual in enumerate(population):
        random_stopping_point = random.randint(0, fitness_sum)
        partial_sum = 0

        for individual_x in population:
            partial_sum += individual_x.fitness

            if partial_sum >= random_stopping_point:
                population_sel[index] = deepcopy(individual_x)
                break

    return population_sel
```

Termination Criterion. The current implementation terminates after either going through 1500 generations or finding a layout such that no edges intersect. First, recombination is done on the population, followed by mutation. Then the population is evaluated so that fitness scores can be calculated. Finally, selection is done to determine who will be a part of the next generation. The algorithm as a whole looks as follows:

```
while current_generation < num_generations and current_max_fitness <
```

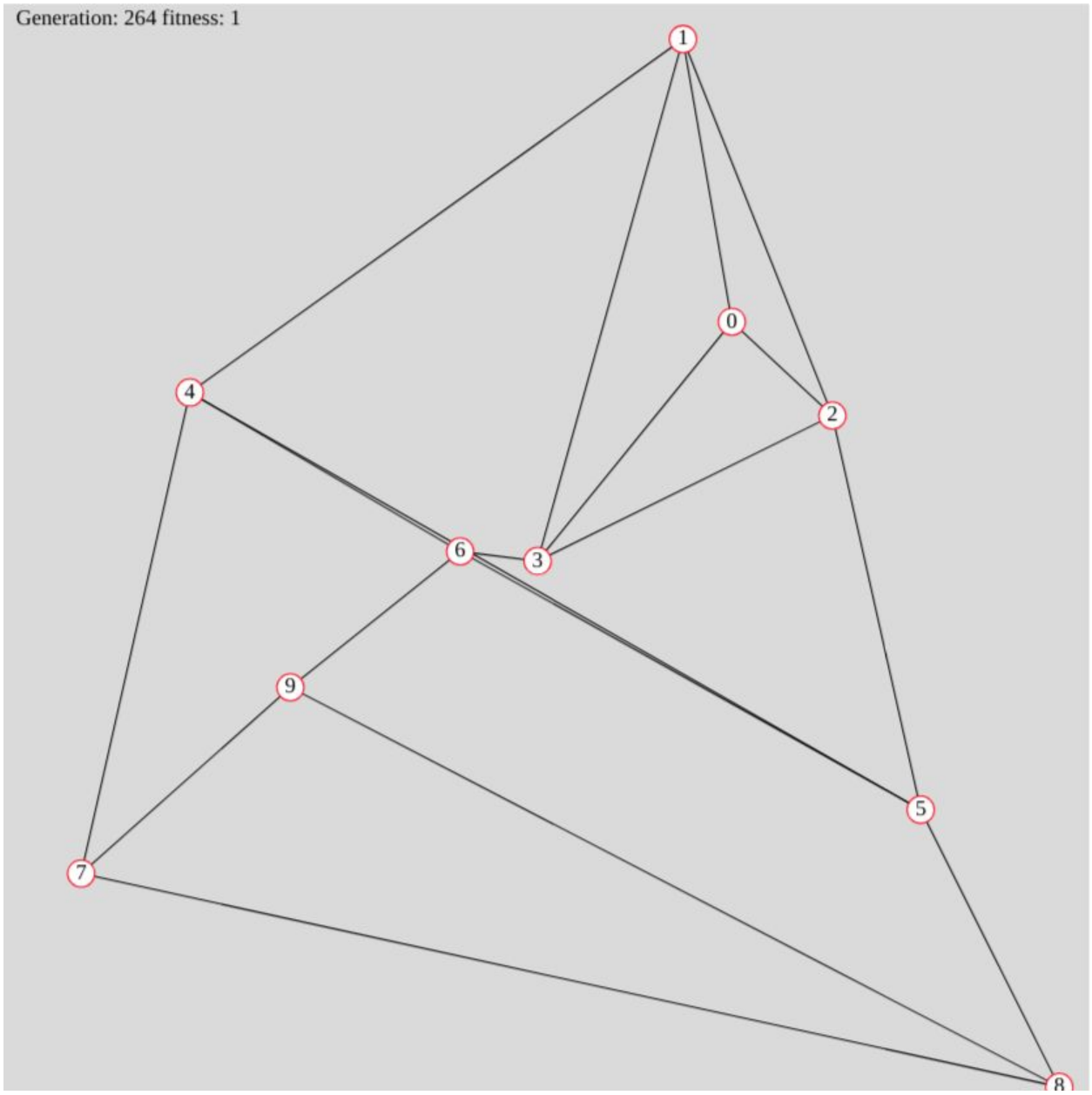
```
max_fitness:

    population = recombination(population)
    population = mutation(population)
    evaluate(population)
    population = selection(population)
    current_generation += 1
```

Trial Run. Data for this graph was obtained by running the algorithm with the following properties.

- Recombination Rate: 0.5
- Mutation Rate: 0.05
- Population: 300
- Max Generations: 1500

In many instances, the best individual in the population only needed to fix 1 intersecting link pair, however this was very hard to obtain. With a lower recombination rate, the max fitness was fixed at 151 or 152 (almost perfect), but individuals would rarely hit a fitness score of 153. The recombination operator may need to be redesigned because two parents with high fitness scores may produce an offspring with a low fitness score depending on how their graphs are structured. Aditionionally, two individuals with the same fitness score could have entirely different layouts. With one individual, you may only need to move one point over by a pixel, whereas the other individual may require a point to be moved over a 100 pixels to obtain a perfect fitness score.

Example Layout Produced By the Algorithm.



In this example, I set the target fitness to be 152 (ie. out of the 153 link pairs, 1 pair can have an intersection). Since my target was only 1 intersection, the fitness shows 1 since 152/152 = 1. The intersection here occurs between the link pairs (4, 5) and (6, 3). To view this, open the html file located in the folder “graph_for_evolutionary_algorithm” using Chrome.

Source Code: