Evolutionary Algorithms - Project Report

EL KHADIR Bachir
Ecole Polytechnique

May 30, 2014

Abstract

This paper presents the results of the application of Evolutionary algorithms on various problems. The problems are different, but the modelling remane the same. The first section presents a generic class EA implementing the base mechanism of genetic algorithm. For each problem, we must find a suitable representation (tuple, a matrix ...), then we must implement a fitness function and a mutation/crossover operator.

I choose *Python* as a programming langage.

Contents

1 Implementation of the GA algorithm (file "ga.py")

The EA class implmenets the $(1 + (\lambda, \lambda))$ GA in a generic form. The constructor is:

```
def __init__(self, fitness, mutation=None, crossover=None)
```

It requires:

- a fitness function to evaluate individuals,
- a mutation operator, by default, this operator is used:

```
def default_mutation(l, x):
        bits_to_change = random.sample(range(len(x)), l)
        x_mut = list(x)
        for b in bits_to_change:
                x_mut[b] = 1 - x_mut[b]
        return x_mut
```

- a crossover operator

```
def default_crossover(c, x, xx):
        parent = [x, xx]
        parent_choice = bernoulli.rvs(c, size=len(x))
        return [ parent[p][i] for i, p in enumerate(parent_choice) ]
```

Then, the *run* function does all the work. It generate the children, select the best one, does the crossover and mutation phase, and then repeat the process *n generations* times.

```
def run(self, n, x_init, offspring_size=5, n_generations=10, p=None, c=None, self_adapt=False, max_fitness=No
```

1.1 The One-Max problem (file "onemax.py")

The implmentation of the one-max problem is straight forward. We start with a random vector in $\{0,1\}^n$. Default mutation and crossover operator work fine. For the fitness function, we sum all the bits in x (there is a builtin function *sum* provided with *Python*). The code is as follow:

```
ea_algo = ga.EA(fitness=sum)
x_init          = np.random.random_integers(0, 1, size=n)
best_x          = ea_algo.run(n, x_init, offspring_size=5, n_generations=100)
```

1.1.1 Stats

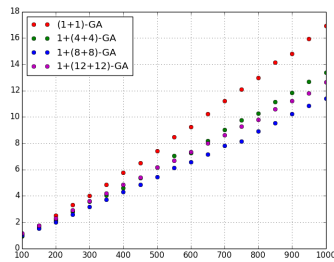

Figure 1: Number of fitness evaluations for $\lambda \in \{1, 4, 8, 12\}$

1.2 The Maximum Matching (file "matching.py")

We represent an undirected graph by it's adjacency list.

```
vertices = range(n)
edges = [(1, 2), (3, 4)]
```

The function reponsible for calculating the degree of a vertex v in ine the subgraph consisting of the edges of M:

```
def deg(M):
    deg_m = np.zeros(n)
    for i, (e, f) in enumerate(edges):
        if M[i]:
            deg_m[e] += 1
            deg_m[f] += 1
    return sum([max(0, d-1) for d in deg_m])
```

And the fitness function:

```
def fitness(M):
    return sum(M) - m * deg(M)
```

Here is the result (edges of the best matching are coloured in red):

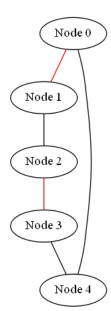

Figure 2: Best matching

1.2.1 Stats

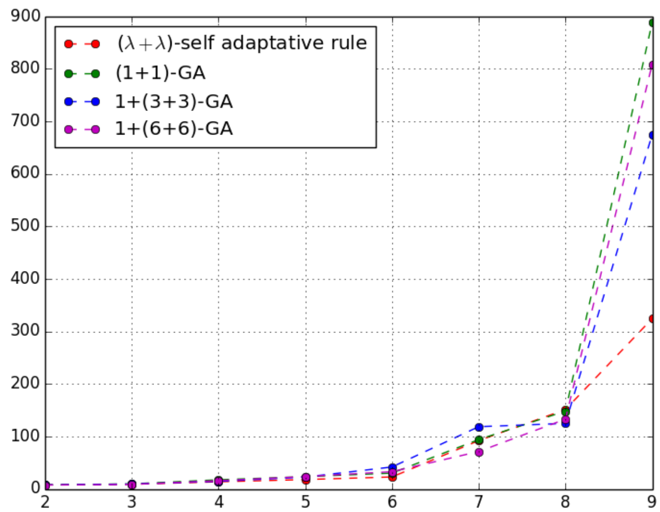

Figure 3: Number of fitness evaluations for $\lambda \in \{1, 3\}$ and for the self adaptative rule

1.3 Euler cycles (file eulerian_cycle.py)

We represent graphes by an adjacency matrix. The function responsible for calculating the degree of a vertex v in the graph is:

```
def deg(v):
    return sum([edges[v][i] for i in vertices if i != v])
```

We can use the default *crossover* operator. The implementation of the mutation operator as suggested:

```
def find_match_and_remove(Mv, e):
    f = None
    k = len(Mv)-1

    # Look for an edge f matched to e
    while k >= 0 and f is None:
                        i, j = Mv[k]
                        if i == e:
                                f = j
                        if j == e:
                                f = i
                        k -= 1
    # Remove the matching (if possible)
    if f != None:
                        Mv.pop(k+1)
    return f

def mutation(l, x):
        x_mut = deepcopy(x)
        for _ in range(l):
                # Choose a random vertex v
                v = random.randint(0, n-1)
                # if deg(v) == 2, a trivial perfect matching is [ (i, j) ]
                if deg(v) == 2:
                        i, j = [w for w in vertices if edges[v][w] and v != w]
                        x_mut[v] = [(i,j)]
```

```
            else:
                    # Choose a random edge e incident to v,
                    e = random.choice([i for i in vertices if i != v and edges[v][i]])
                    # See if it is matched. If it's the case, remove the match
                    f = find_match_and_remove(x_mut[v], e)

                    # Same thing for (ee, ff)
                    ee = random.choice([i for i in vertices if (not i in[ v, e, f]) and edges[v][i]])
                    ff = find_match_and_remove(x_mut[v], ee)

                    # match (e, ee) and (f, ff) if possible
                    x_mut[v].append( (e, ee) )
                    if f and ff:
                            x_mut[v].append( (f, ff) )
    return x_mut
```

And the fitness function:

```
def fitness(x):
        cycles_penality = nb_edges - sum(map(len, x))
        return  - cycles_penality
```

After the algorithm has finished, we can reconstruct the path from the maximum matching:

```
def reconstruct_path(xx):
        x = deepcopy(xx)
        v = 0
        try:
                (i,j) = x[v][0]
        except IndexError:
                return []
        path = [i, v]

        while j != None:
                j = find_match_and_remove(x[v], i)
                path.append(j)
                i, v = v, j
        return path[:-2]
```

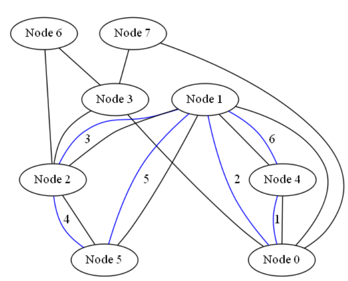

Figure 4: Euler cycle

1.3.1 Stats

1.4 Own Idea

1.4.1 The N-Queen's Problem (file "chess.py")

In chess, a queen can move horizontally, vertically, or diagonally. Given a chess board with N rows and N columns, N-Queen's problem asks how to place N queens on the chess board so that none of them can hit any other in one move.

Since the number of queens is equal to the number of rows/columns, in each row there is exactly one queen, and two different queens are placed in two different columns. Therefore a configuration (position of the queens on the chess board) is represented with a permutation of $\{0, ..., N-1\}$. An individual is a N-tuple $x = (x_1, ..., x_N)$, where $0 \leq x_i < N$ is the column of the queen placed on the row i. We start with the N-tuple $(0, ..., N-1)$:

```
x_init = range(n)
```

With this representation, there will be no queens on the same row/column. We only have to check for possible collisions diagonally. The fitness function counts the number of those collisions:

```
def fitness(t):
    x = np.zeros((n , n))
    for i, j in enumerate(t):
        x[i][j] = 1

    s = 0
    # diags
    for k in range(-n + 1, n):
        # diag (i, i+k) (n-i-1, i+k)
        q_diag = sum([ x[i][i + k] for i in range(max(0, -k), min(n, n - k))])
        s += max(0, q_diag - 1)
        q_diag = sum([ x[n - i - 1][i + k] for i in range(max(0, -k), min(n, n - k))])
        s += max(0, q_diag - 1)
    return -s
```

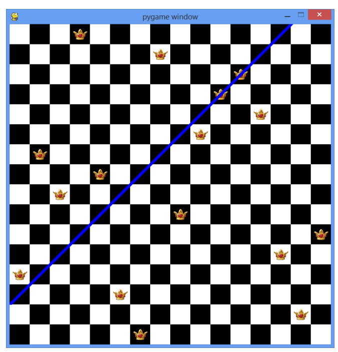

Figure 5: Valid configuration

1.4.2 2D Image reconstruction (file ”2d$_{r}econstruct.py$”)

This program tries to rebuild a 2D grey scale image using only disks. Individuals are strings representing lists of 256 circles $\{(x, y, radius, color, alpha)\}$ in binary format where:

- x and y are the position of the center of the circle
- *color* is an integer between 0 and 255 giving the tone of grey the circle must be filled with
- *alpha* is the transparency

The fitness function is a per-pixel RGB comparison:

```
def fitness(x):
        draw_spheres(x)
        pixels = array2d(srf)
        return -np.linalg.norm((reference_pixels - pixels) / M)
```

Since we use a string based representation, we can use the default crossover and mutation operators as in the first algorithm.

Here is the result:

Figure 6: Source image

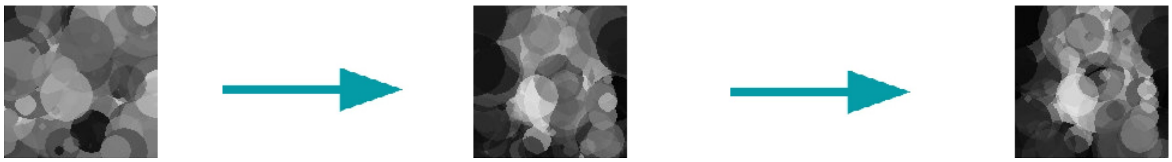

Figure 7: Evolution

1.4.3 Finance