AppliedW11

May 7, 2024

[3]:
```
from IPython.display import Image, display
from matplotlib import pyplot as plt
import numpy as np
```

1 Question 1

Discuss your assignment with your Applied Class demonstrators.

2 Question 2

- Solution representation: Can leave the same – the search space is not changing.
- fitness function: Can leave the same – the problem is not changing.
- fitness evaluation: Can evaluate the fitness of a population by the maximum of the fitness of all members in the population.
- Crossover and mutation: Mutation the same as perturb in EA case. To mutate, we can a block of the bitlist from one parent and the remainder of the bitlist to obtain the correct length from the second parent.

2.1 Evolutionary Algorithm

[80]:
```
class EA_knapsack:
    def __init__(self, W, V, k,pop_size, generations, n_elite = 1, p_mut=0.1,␣
 ↪p_cros=0.1, random_state=None):
        self.W = W
        self.V = V
        self.k = k
        self.pop_size = pop_size
        self.generations = generations
        self.n_elite = n_elite
        self.p_mut = p_mut
        self.p_cros = p_cros
        self.random_state = random_state
        np.random.seed(self.random_state)
```

```
    def mutate(self, x):
        flip = np.random.choice(len(x))
        x_pert = x.copy()
        x_pert[flip] = 1 - x_pert[flip]
        return x_pert

    def crossover(self, x1, x2):
        c = np.random.randint(len(x1))
        x3 = np.hstack([x1[:c], x2[c:]])
        return x3

    def sample(self, x):
        population = np.zeros(shape=(self.pop_size, len(x)))
        for i in range(self.pop_size): # sample pop_size many candidate solutions
            sol_size = np.random.randint(low=0, high=len(x)+1) # randomly select␣
↪the number of elements in solution
            sol_idxs = np.random.randint(low=0, high=len(x), size=sol_size) #␣
↪randomly select correspoinding items
            sol = np.zeros_like(x) # 0 means not selected
            sol[sol_idxs] = 1 # 1 means selected
            population[i, :] = sol
        return population

    def fitness(self, x):
        valid = np.dot(x, self.W) <= self.k # vector where True means solution␣
↪does not exceed capacity
        value = np.dot(x, self.V) # vector of solutions values
        f = np.where(valid, value, valid) # when capacity is exceeded, set value␣
↪to False (i.e: 0) otherwise, sum of item values
        return f

    def solve(self, verbose=False):
        pop = self.sample(self.W)

        for i in range(self.generations):
            f = self.fitness(pop)
            f_total = np.sum(f)
            p = f/f_total
            pop_new = np.zeros_like(pop)
            ranking = np.argsort(f)[::-1]
            pop_new[:self.n_elite] = pop[ranking[:self.n_elite]].copy()

            if verbose: print('Generation: {0}, Best: {1}, f={2}'.format(i,␣
↪pop_new[0], f[ranking[0]]))

            for j in range(self.pop_size - self.n_elite):
                idx = np.random.choice(self.pop_size, p=p)
```

```
                x = pop[idx].copy()
                if np.random.random() < self.p_mut:
                    x = self.mutate(x)
                if np.random.random() < self.p_cros:
                    idx2 = np.random.choice(self.pop_size, p=p)
                    x2 = pop[idx2].copy()
                    x = self.crossover(x, x2)
                pop_new[j+self.n_elite] = x

            pop = pop_new.copy()

        f = self.fitness(pop)
        best_idx = np.argmax(f)
        best = pop[best_idx]

        if verbose: print('{0} solutions checked'.format(self.pop_size*self.
↪generations))

        return best, f[best_idx]
```

[81]:
```
EA = EA_knapsack(W=W, V=V, k=k, pop_size=10, generations=200, n_elite=1,␣
↪random_state=random_state)
EA.solve(verbose=False)
```

[81]:
```
(array([0., 0., 0., 1., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 1., 1.,
        1., 0., 0.]),
 48.0)
```

[ ]:

3 Question 3

Both games can be handled in a similar way, it is recommended you check out the paper alongside the Super Mario Bros question. They can be encoded as combinations of input moves. Each game has a maximum number of inputs per second, and in the case of Super Mario Bros, there is a time limit. This gives a maximum number of moves per game. Pac-Man doesn't have a time limit so the length of the input moves can be arbitrary - best to set a fixed length though.

The fitness function can be how high the combination of inputs score. Can be perturbed by changing a subset of input moves. For genetic algorithms, crossover can occur by taking two games, splitting at the same point and swapping components.

4 Question 4

Assume that 99/100 all agents are using the upper road. The 100th driver is still not incentivised to use the upper road, as both choices now have a travel time of 100. In equilibrium, all drivers take the lower road and have a travel time of 100.

Total time spent on road is: F(x) = xˆ2 + 100*(100-x). 100 minuites for each person on the upper road, and x minutes for each of the x people on the lower road. A the global optima exists either at F′(x) = 0 or at one of the endpoints.

F′(x) = 2x - 100 and F′(x) = 0 at x=50.

F(0) = 10000, F(100) = 10000, F(50)=7500.

The optimal distribution is to have half the drivers on each road.

5 Question 5

[22]:
```
display(Image(filename='game_dominance.png'))
```

	a	b	c	d	e
A	$63, -1$	$28, -1$	$-2, 0$	$-2, 45$	$-3, 19$
B	$32, 1$	$2, 2$	$2, 5$	$33, 0$	$2, 3$
C	$54, 1$	$95, -1$	$0, 2$	$4, -1$	$0, 4$
D	$1, -33$	$-3, 43$	$-1, 39$	$1, -12$	$-1, 17$
E	$-22, 0$	$1, -13$	$-1, 88$	$-2, -57$	$-3, 72$

Start with the row player: - Row A is not dominated, as there are situations where no strategy can beat is (i.e: when the column player plays column a) - Rows B and C are similarly not dominated - Row D is dominated: there is always a better option than it (when column player plays a, row player should play A. when column player plays b, row player should play C. when column player plays c, row player should play C. when column player plays d, row player should play B. when column player plays e, row player should play B). Thus, we remove Row D from the row player's viable strategies - Row E can be removed for similar reasons

Now move on to the column player, keeping in mind that we will be ignoring rows D and E as they have alreayd been eliminated. - Column a is dominated: when row player plays A, column player should play d. when row player plays B, column player should play c. when row player plays C, column player should play e. - Column b is also dominated

Move back to the row player, now ignoring rows D and E and columns a and b - Now rows A and C are dominated since the column player would never play stategy a or b and we therefore ignore those columns in the matrix.

Move again to the column player, noting that the row player now will only play row B - column C has the best payoff at this point

we have now reached a stage where each player has only one strategy that is never dominated. Thus, by iterative removal of dominated strategies, the players should use profile (B, c) with payoffs (2, 5).

[ ]: