# HOMEWORK 3 FOR MODERN OPTIMIZATION METHODS

R09946006 | 何青儒 | HO, Ching-Ru | 10/06/2020

## Knapsack Problem via GA.

1. Gut Knife, Desert Eagle Magnum, AK-47 Rifle, Gas Mask and Tactical Shield.
   - $weightList(w_i) = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.2, 122.4, 247.6, 352, 24.2, 32.1, 42.5]$
   - $pointList(c_i) = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485, 9, 12, 15]$
   - $itemList(x_i) = [0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1]^T$
   - $weightSum = \sum_{i=1}^{15} w_i x_i = 257.6$
   - $pointSum = \sum_{i=1}^{15} c_i x_i + h(x) = 346 + 0 = 346$
2. Optimization problem:
   - Notations:
     - There are totally $n$ items can be carried, in this case, $n = 15$.
       - There are totally $m$ types, in this case, $m = 4$, which refers to knife, pistols, primary, and equipment mentioned in the second column.
       - The $i-th$ item can get $c_i$ points, and its weight is $w_i$.
       - The maximum weight of inventory bag is $W$, we can not carry items over this upper bound.
       - If we carry the $i-th$ items, then make $x_i = 1$. If not, $x_i = 0$.
       - We have bonus point function $h(x)$, refer to rule $(b), (c), (d)$.
   - max $f(x) = \sum_{i=1}^{n} c_i x_i + h(x)$
   - Bonus Point: $h(x) = \begin{cases} 5, (x_1 \wedge x_6) = 1 \\ 15, (x_4 \wedge (x_9 \vee x_{10})) = 1 \\ 25, ((x_8 \vee x_{11}) \wedge (x_6 \vee x_{15})) = 1 \\ 70, (x_{13} \wedge x_{14} \wedge x_{15}) = 1 \end{cases}$
   - subject to
     - $g(x) = \sum_{i=1}^{15} w_i x_i \leq W$,
     - and rule $(a)$ means $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_{13} \vee x_{14} \vee x_{15}) = 1$
   - for $x_i \in \{0, 1\}, i = 1, 2, 3, \cdots, 15$
3. There are 10342 combinations (see `combinations.txt` file), including an empty knapsack.
4. GA1 Operators:
   - Selection: roulette-wheel selection
   - Crossover: 2-point crossover
   - Mutation: multi bit flip mutation (random 2 locations)

```
# Selection: roulette-wheel selection
def selection_roulette_wheel(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    sum_fitness = sum(fitness) # as an int
    roulette_table = np.empty(sum_fitness)
    start = 0
    for i in range(len(fitness)):
        end = start + fitness[i]
        roulette_table[start : end] = i
        start = end
    for j in range(num_parents):
        pick = randint(0, sum_fitness - 1)
        k = int(roulette_table[pick])
        parents[j] = population[k]
    return parents

# Crossover: 2-point crossover
def crossover_2point(parents, num_offsprings):
    offsprings = np.zeros((num_offsprings, parents.shape[1]))
    crossover_point = [0,0]     # random 2 points
    while(crossover_point[0] == crossover_point[1]):
        crossover_point = [rd.randint(0, parents.shape[1]-1), rd.randint(0,
parents.shape[1]-1)]
    crossover_point = sorted(crossover_point) # sorted(increasing)
    crossover_rate = 1 # we assume that it would and should croosover in anytime
    i = 0
    while (i < num_offsprings):  # until fullfill
        parent1_index = i % parents.shape[0]
        parent2_index = (i+1) % parents.shape[0]
        x = rd.random()
        if x > crossover_rate: # if probabilty bigger than cross_rate, it will crossover
            continue
        parent1_index = i % parents.shape[0]
        parent2_index = (i+1) % parents.shape[0]
        offsprings[i, 0:crossover_point[0]] = parents[parent1_index, 0 : crossover_point[0]]
        offsprings[i, crossover_point[0] : crossover_point[1]] = parents[parent2_index,
crossover_point[0] : crossover_point[1]]
        offsprings[i, crossover_point[1] : ] = parents[parent1_index, crossover_point[1] : ]
        i=i+1
    return offsprings

# Mutation: multi bit flip mutation (random 2 locations)
def mutation_rand2point(offsprings): # random 2 points mulyibit-flip technique
    mutants = np.empty((offsprings.shape))
    for i in range(mutants.shape[0]): # for each in in generations
        mutation_point = [0,0] # random 2 points
        while(mutation_point[0] == mutation_point[1]):
            mutation_point = [rd.randint(0, offsprings.shape[1]-1), rd.randint(0,
offsprings.shape[1]-1)]
        mutation_point = sorted(mutation_point)
        mutants[i,:] = offsprings[i,:] # copy as a list
        # for the 1st and 2nd mutation points
        for k in range(len(mutation_point)):
            if (mutants[i, mutation_point[k]] == 0):
                mutants[i, mutation_point[k]] = 1
            elif(mutants[i, mutation_point[k]] == 1):
                mutants[i, mutation_point[k]] = 0
            else:
                print("Something error happened in mutation_rand2point!")
    return mutants
```

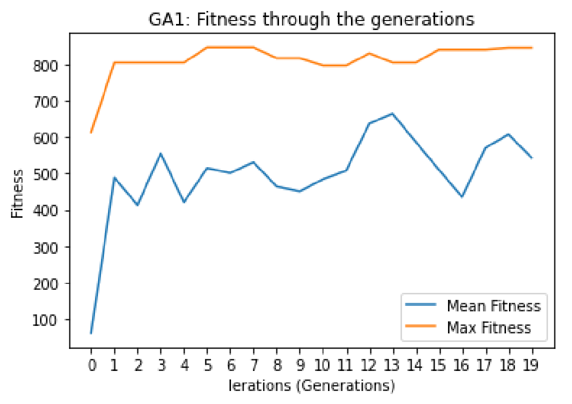
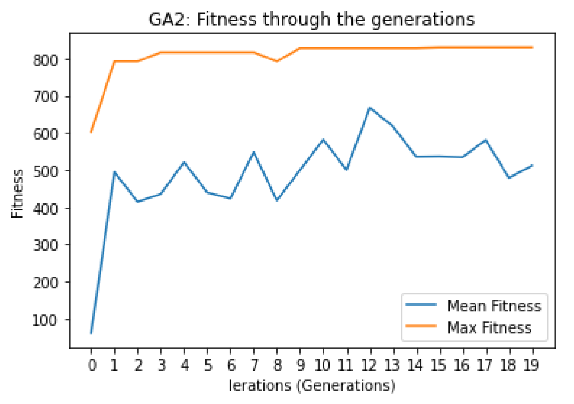
5. GA2 Operators:
    - Selection: roulette-wheel selection
    - Crossover: uniform crossover ($crossover\_probability = 0.1$)
    - Mutation: multi bit flip mutation (consecutive based on item types)
        - Sorry, I'm not very sure what it means actually, so I assume 4-point ($= m$) consecutive mutation.

```
# Selection: roulette-wheel selection
# Omitted, as stated above.

# Crossover: uniform crossover (crossover probability = 0.1)
def crossover_uniform(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    probabilty_uni = 0.1
    i = 0
    while (parents.shape[0] < num_offsprings):
        parent_index = i % parents.shape[0]
        for j in range(len(offspring)): # for i-th chromosome, j-th gene
            x = rd.random()
            if x > probabilty_uni: # compare the probability
                offsprings[i,j] = parents[parent_index, j] # no crossover
            else: # crossover
                if(parents[parent_index, j] == 0):
                    offsprings[i, j] = 1
                elif(parents[parent_index, j] == 1):
                    offsprings[i, j] = 0
                else:
                    print("Something error happened in crossover!")
        i=i+1
    return offsprings

# Mutation: multi bit flip mutation (consecutive based on item types)
def mutation_consecutive(offsprings): #bit-flip technique
    types = 4
    mutants = np.empty((offsprings.shape))
    for i in range(mutants.shape[0]):
        mutants[i,:] = offsprings[i,:]
        start = randint(0,offsprings.shape[1]-1)
        for i in range(types):
            indicator = (start + i) % offsprings.shape[1]
            if mutants[i,indicator] == 0 :
                mutants[i,indicator] = 1
            else :
                mutants[i,indicator] = 0
    return mutants
```

6. See `HW3_GA1.ipynb` and `HW3_GA2.ipynb` (format: jupyter notebook).
    - GA1 result:
        - MAX Points: 845 with raw point 800 and bonus point 45 (apply rule $(b)$, $(c)$, $(d)$)
        - Total weight: $528.1 < 529$ (most weight we can carry)
        - Item index (start from 1): $[1, 3, 4, 5, 6, 7, 8, 9, 14]$
        - Items: Shadow Daggers (Knife), Gut Knife (Knife), 228 Compact Handgun (Pistols), Night Hawk (Pistols), Ingram MAC-10 SMG (Primary), Leone YG1265 Auto Shotgun

        (Primary), M4A1 Carbine (Primary), Night-Vision Goggle (Equipment)
    - GA2 result:
        - MAX Points: 830 with raw point 785 and bonus point 45 (apply rule $(b)$, $(c)$, $(d)$)
        - Total weight: $512.7 < 529$ (most weight we can carry)
        - Item index (start from 1): $[1, 3, 4, 6, 8, 9, 10, 14]$
        - Items: Shadow Daggers (Knife), Gut Knife (Knife), 228 Compact Handgun (Pistols), Desert Eagle Magnum (Pistols), Leone YG1265 Auto Shotgun (Primary), M4A1 Carbine (Primary), AK-47 Rifle (Primary), Night-Vision Goggle (Equipment)

7. Progress diagrams of GA1 and GA2:
    - Selection: Both GAs use roulette-wheel selection. The disadvantage of roulette-wheel selection method is that though the best solution in parent generation has the largest probability, but it might not be chosen in this section, making the `Max Fitness` may be smaller between different generation.
    - Crossover & Mutation: GA1 uses 2-point crossover with random 2-point flip mutation, and GA2 uses uniform crossover with consecutive mutation. It seems GA2 has more "change and diversity" between generation than GA1. We can observe `Mean Fitness` line between two figures and find that line of GA2 is more violent than another one.
    - I think method appiled in GA2 can generate more diverse offsprings than method applied in GA1. In the crossover section, the uniform method can generate offsprings more randomly because it doesn't depend on the original fragment of parents generation, but 2-point crossover will copy the fragment from parents generation, making offsprings is similar to parents. In the mutation section, flipping 4 $(= m)$ bits can generate offsprings more randomly, too, because it mutates more bits than 2-point.

| **GA1** | **GA2** |
|---|---|
|  |  |