**Raport**

Przemysław Lis 229940

Wojciech Majchrzak 229947

1. **Zasada działania programu**

Program wczytuje dane z pliku dane.csv. Następnie przypisuje dane do atrybutu items w klasie Backpack. Po podaniu rozmiaru plecaka wczytywane są parametry podstawowe takie jak pojemność plecaka, wielkość populacji, wielkość osobnika oraz ilość generacji. Następnie poprzez odpowiednie funkcje opisane poniżej program wykonuje symulacje tworzenia się nowych generacji. Wynikiem programu jest populacja która posiada pewne obiekty w plecaku i ich waga nie przekracza maksymalnej wagi plecaka. Z tych osobników można wytypować najlepszego za pomocą funkcji get\_the\_best\_individual().

1. **Wybrane miejsca implementacji rozwiązania**

Diagram UML:

Obraz zawierający tekst

Opis wygenerowany automatycznie

Funkcja \_\_generate\_initial\_population generuje początkową populacje. Losowo ustawia geny 0 lub 1 w chromosomie osobnika.

def \_\_generate\_initial\_population(self):  
 population = []  
 for individual in range(self.population\_size):  
 current\_individual = Individual(self.individual\_size)  
 for gen in range(self.individual\_size):  
 random\_gen = random.randint(0, 1) # generate value 0 or 1  
 current\_individual.gens.append(random\_gen)  
 population.append(current\_individual)  
 return population

Funkcja \_\_calculate\_fitness\_of\_the\_individual oblicza sumę wartości każdego osobnika która jest potem przekazywana jako fitness. W przypadku gdy waga poszczególnych rzeczy z plecaka jest większa niż maksymalna jego pojemność, to wtedy do fitness danego osbnika przypisywane jest 0.

def \_\_calculate\_fitness\_of\_the\_individual(self, individual: Individual):  
 sum\_of\_weight = 0.0  
 sum\_of\_values = 0.0  
 for i, gen in enumerate(individual.gens):  
 if gen == 1:  
 sum\_of\_weight += self.backpack.items['Waga'][i]  
 sum\_of\_values += self.backpack.items['Wartosc'][i]  
 if sum\_of\_weight > self.backpack.max\_capacity:  
 individual.fitness = 0  
 return 0  
  
 individual.fitness = sum\_of\_values  
 return sum\_of\_values # fitness of the individual

Funkcja \_\_set\_probability\_in\_population ustawia prawdopodobieństwo wybrania danego osbnika podczas metody ruletki.

def \_\_set\_probability\_in\_population(self):  
 sum\_of\_fitness = 0  
 for individual in self.population:  
 sum\_of\_fitness += individual.fitness  
  
 for individual in self.population:  
 try:  
 individual.choose\_probability = individual.fitness / sum\_of\_fitness  
 except ZeroDivisionError:  
 individual.choose\_probability = 0

Funkcja roulette\_wheel\_selection na początku ustawia prawdopodobieństwo wyboru osobnika funkcją opisaną powyżej a następnie losuje osobnika.

Zwracany jest indeks wylosowanego osbnika.

def roulette\_wheel\_selection(self, number\_of\_chosen):  
 self.\_\_set\_probability\_in\_population()  
 intervals = []  
 sum\_ = 0  
  
 for i in range(number\_of\_chosen):  
 intervals.append([i, sum\_, sum\_ + self.population[i].choose\_probability])  
 sum\_ += self.population[i].choose\_probability  
  
 random\_number = random.random()  
 result = [interval for interval in intervals if interval[1] < random\_number <= interval[2]]  
 return result

funkcja one\_point\_crossing krzyżuje dwóch rodziców i w wyniku tego powstaje dwójka dzieci które następnie są dodawane do populacji.

Punkt mutacji jest losowany za każdym razem.

def one\_point\_crossing(self, parent\_a: Individual, parent\_b: Individual, crossing\_point):  
 children = []  
 children\_1 = Individual(self.individual\_size)  
 children\_2 = Individual(self.individual\_size)  
 for i, gen in enumerate(parent\_a.gens):  
 if i <= crossing\_point:  
 children\_1.gens.append(copy(parent\_a.gens[i]))  
 else:  
 children\_1.gens.append(copy(parent\_b.gens[i]))  
  
 for i, gen in enumerate(parent\_a.gens):  
 if i <= crossing\_point:  
 children\_2.gens.append(copy(parent\_b.gens[i]))  
 else:  
 children\_2.gens.append(copy(parent\_a.gens[i]))  
  
 children.append(children\_1)  
 children.append(children\_2)  
  
 return children

funkcja individual\_mutation mutuje wybrany losowo gen w podanym osobniku na przeciwny.

def individual\_mutation(self, individual: Individual):  
 random\_gen = random.randint(0, self.individual\_size - 1)  
 if individual.gens[random\_gen] == 1:  
 individual.gens[random\_gen] = 0  
 else:  
 individual.gens[random\_gen] = 1  
 return individual

Funkcja simulation używa powyższych funkcji aby przeprowadzić symulacje. W funkcji wykonywane są generacje określone na początku i w każdej generacji wykonywana jest metoda ruletki a następnie krzyżowany i mutowany jest gen w dzieciach powstałych przy krzyżowaniu.

def simulation(self):  
 the\_best\_global\_adaptation = 0  
 for generation in range(self.generation\_numbers):  
 the\_best\_current\_adaptation = 0  
 self.delete\_weak\_individual()  
 self.\_\_calculate\_fitness\_of\_population()  
 for individual in self.population:  
 if individual.fitness > the\_best\_current\_adaptation:  
 the\_best\_current\_adaptation = individual.fitness  
  
 if the\_best\_current\_adaptation > the\_best\_global\_adaptation:  
 the\_best\_global\_adaptation = the\_best\_current\_adaptation  
  
 parent\_1 = self.roulette\_wheel\_selection(self.population\_size)  
 parent\_2 = self.roulette\_wheel\_selection(self.population\_size)  
  
 crossing\_point = random.randint(0, self.individual\_size - 1)children = self.one\_point\_crossing(self.population[parent\_1[0][0]], self.population[parent\_2[0][0]], crossing\_point)  
  
 for child in children:  
 child = self.individual\_mutation(child)  
 self.connect\_population\_and\_children(child)