Plik lista\_1\_v2.py

**Genom**: Zakodowane potencjalne rozwiązanie. Gdzie jedynka reprezentuje to że bierzemy jakiś przedmiot a zero że go nie bierzemy. Przykładowo chromosom [1 0 1 0 1] oznacza, że bierzemy pierwszy, trzeci i piąty przedmiot.

**Funkcja „rozparsuj\_linie”** służy do podzielenia linii z pliku csv, która jest w formie stringa na tablice numpy reprezentujące wagi przedmiotów, wartości przedmiotów oraz maksymalną pojemność plecaka (maksymalna waga). Dzielimy linie/ strona na części według średników na trzy elementy. Waga: usuwamy nawiasy kwadratowe i zmieniamy używając wyrażenie map stringi na inty. Później tworzymy z tego liste i tą listę wykorzystujemy jako argument w np.array. który tworzy nam wektor

**Funkcja „generuj\_poczatkowa\_populacje”** służy do wygenerowania początkowej populacji genomów,Korzysta z funkcji numpy random.choice. Generuje populację jako macierz, gdzie każdy wiersz to chromosom

**Funkcja** „**funkcja\_dostosowania**” służy do oceny dostosowania do środowiska „fitness” poszczególnych genomów. W przypadku naszego problemu plecakowego, równa się ona sumie wartości przedmiotów dla danego genomu o ile mieszczą się one w maksymalnej dozwolonej wadze, jeśli nie przyznajmy wartość zero. Do obliczania sumarycznej wartości oraz wagi przedmiotów, dokonujemy mnożenia wektorów odpowiednio wartości, wag z wektorem genomów.  
wykorzystujemy do tego funkcję numpy np.dot *macierz populacji mnożymy z wektorem wartości, dostajemy sume wartości, analogicznie mnoży populację z wektorem wag i tak dostajemy sume wag. W tych przypadkach gdzie suma wag jest większa niż max. Pojemność, przyedzialmy zero, w przeciwnym przypadku, sumę wartości*

**1. Cel funkcji oceny**

Funkcja oceny ma za zadanie:

* **Maksymalizować sumę wartości przedmiotów** zawartych w plecaku.
* **Uwzględnić ograniczenie wagowe**, tj. suma wag przedmiotów nie może przekraczać pojemności plecaka. W przeciwnym wypadku rozwiązanie powinno zostać ukarane.
* def funkcja\_dostosowania(chromosom, wartosci, wagi, maks\_pojemnosc):
* suma\_wartosci = 0
* suma\_wag = 0
* for i in range(len(chromosom)):
* if chromosom[i] == 1:
* suma\_wartosci += wartosci[i]
* suma\_wag += wagi[i]
* if suma\_wag > maks\_pojemnosc:
* return 0
* return suma\_wartosci
* w zmiennej suma\_wartości sumuje wartości wszystkich przedmiotów branych w tym chromosomie, analogicznie w zmiennej suma\_wag sumuje wagi przedmiotów
* w pętli iteruje każdy genom chromosomu jeżeli jest równy 1, wyciągam odpowiednio z tablicy wartości oraz tablicy wagi, wartość oraz wagę przedmiotu i dodaje je do zmiennych suma\_wag oraz suma\_wartości
* po przejściu pętli sprawdzam czy suma wag nie przekroczyła maksymalnej pojemności, jeżeli przekroczyła zwracam 0, jeśli nie to zwracam sume\_wartości

**3. Struktura funkcji oceny**

Funkcja oceny dla problemu plecakowego powinna działać zgodnie z poniższymi zasadami:

1. **Obliczenie sumy wartości przedmiotów w plecaku**:
   * Dla danego rozwiązania (chromosomu), wybieramy przedmioty oznaczone jako 1 w wektorze binarnym.
   * Sumujemy wartości tych przedmiotów.
2. **Sprawdzenie ograniczenia wagowego**:
   * Obliczamy sumę wag dla wybranych przedmiotów.
   * Jeśli suma wag przekracza maksymalną pojemność plecaka, stosujemy karę.
3. **Przyznawanie punktów za dopasowanie**:
   * Rozwiązania spełniające ograniczenie wagowe otrzymują jako wartość dopasowania sumę wartości wybranych przedmiotów.
   * Rozwiązania przekraczające ograniczenie wagowe otrzymują **karę**, np. ich dopasowanie można ustawić na 0 lub zmniejszyć proporcjonalnie do przekroczenia wagi.

W **funkcji main()** otwieramy plik z danymi, zawierającym przypadki

**Analiza obecnej strategii**

1. **Zalety**:
   * Prosta i bezpośrednia implementacja. Rozwiązania, które nie spełniają ograniczenia wagowego, są natychmiast odrzucane.
   * Funkcja oceny efektywnie eliminuje nieprawidłowe rozwiązania, co przyspiesza wyszukiwanie dopuszczalnych rozwiązań.
2. **Wady**:
   * **Brak elastyczności**: Chromosomy, które minimalnie przekraczają dopuszczalną wagę, są traktowane tak samo jak te, które znacząco przekraczają limit. To może prowadzić do zbyt wczesnego odrzucenia potencjalnie dobrych rozwiązań.
   * **Mniejsze możliwości eksploracji przestrzeni rozwiązań**: Odrzucając wszystkie "przeciążone" chromosomy, algorytm ogranicza swoją zdolność do eksploracji rozwiązań bliskich optymalnemu.
   * **Ryzyko pustych populacji**: Jeśli wiele chromosomów przekracza ograniczenie wagowe (np. na początku ewolucji), algorytm może stracić populację lub działać na bardzo ograniczonej liczbie rozwiązań.

**2. Krzyżowanie**

* krzyżowanie jednopunktowe
* def krzyzowanie\_a(chromosom1, chromosom2):
* if len(chromosom1) != len(chromosom2):
* raise ValueError("Chromosomy muszą mieć taką samą długość")
* dlugosc = len(chromosom1)
* punkt\_podzialu = dlugosc // 2
* return chromosom1[:punkt\_podzialu] + chromosom2[punkt\_podzialu:]
* najpierw sprawdzamy czy chromosomy, które chcemy skrzyżować mają tą samą długość ponieważ w przeciwnym wypadku nie możemy ich skrzyżować, kiedy ich długośc jest różna wyrzucamy wyjątek (raise ValueError…)
* następnie wyznaczamy punkt podziału chromosomów czyli indeks tablicy według, którego będziemy dzielić obydwa chromosomy na dwie części jest to długość tablicy, podzielona na dwa, bez reszty (7/2=3, dzielenie całkowite) (długość =…, punkt\_podziału = …)
* następnie zwracamy nowy chromosom potomny, który składa się z elementów chromosomu 1 od indeksu 0 do indeksu punktu podziału (wyłącznie) oraz elementów chromosomu 2, od indeksu punktu podziału włącznie do ostatniego indeksu
* Testy krzyżowania:
* chromosom1: [1, 0, 1, 0, 1], chromosom2: [0, 1, 0, 1, 1], potomostwo: [1, 0, 0, 1, 1]
* krzyżowane jednorodne:
* def krzyzowanie\_b(chromosom1, chromosom2):
* if len(chromosom1) != len(chromosom2):
* raise ValueError("Chromosomy muszą mieć taką samą długość")
* potomstwo = []
* for gen1, gen2 in zip(chromosom1, chromosom2):
* potomstwo.append(random.choice([gen1, gen2]))
* return potomstwo
* najpierw sprawdzamy czy chromosomy, które chcemy skrzyżować mają tą samą długość ponieważ w przeciwnym wypadku nie możemy ich skrzyżować, kiedy ich długośc jest różna wyrzucamy wyjątek (raise ValueError…)
* tworzymy pustą tablice potomstwo, do której będziemy dodawać kolejne genomy chromosomu potomnego
* tworzymy tablice par genomów z chromosomu 1 i 2, w taki sposób, że parujemy ze sobą 1 genom z chromosomu 1 z 1 genomem z chromosomu 2, 2 genom z chromosomu 1 z 2 genomem z chromosomu 2 itd…
* for gen1, gen2 in zip(chromosom1, chromosom2): zip robie z 2 list, listępar
* w pętli przechodzimy wszystkie pary genomów losujemy 1 z nich i dodajemy do tablicy potomstwo. Po przejściu pętli mamy chromosom potomny składający się z losowych genomów z chromosomu 1 i chrom. 2
* przykład: Testy krzyżowania:

[(1, 0), (0, 1), (1, 0), (0, 1), (1, 1)]

chromosom1: [1, 0, 1, 0, 1], chromosom2: [0, 1, 0, 1, 1], potomostwo: [1, 1, 0, 1, 1]

**4. Selekcja**

* **Turniejowa:**
* def selekcja\_turniejowa(populacja, wyniki\_dostosowania\_dict, rozmiar\_turnieju):
* zwyciezcy = []
* for \_ in range(len(populacja)):
* uczestnicy = random.sample(list(populacja), rozmiar\_turnieju)
* najlepszy\_uczestnik = uczestnicy[0]
* najlepsze\_dostosowanie = wyniki\_dostosowania\_dict[tuple(najlepszy\_uczestnik)]
* for uczestnik in uczestnicy[1:]:
* dostosowanie = wyniki\_dostosowania\_dict[tuple(uczestnik)]
* if dostosowanie > najlepsze\_dostosowanie:
* najlepszy\_uczestnik = uczestnik
* najlepsze\_dostosowanie = dostosowanie
* zwyciezcy.append(najlepszy\_uczestnik)
* return zwyciezcy
* selekcja turniejowa to metoda selekcji chromosomów wybranych do krzyżowania poprzez organizowanie mini turnieju, ustalony rozmiar turnieju mówi nam ile losowych chromosomów z populacji wybieramy do turnieju, sam turniej polega na wybraniu chrom. O najlepszym dostosowaniu z wylosowanych chromosomów
* organizujemy tyle turnieji ile potrzebne nam chromosomów do krzyżowania. Ilość wybranych chromosomów jest równa rozmiarowi populacji z bieżącego pokolenia.
* Funkcja selekcji turniejowej przyjmuje trzy argumenty: listę chromosomów w bieżącej populacji, słownik, w którym kluczem jest krotka reprezentująca chromosom, a wartością jest jego dostosowanie, oraz liczbę osobników biorących udział w każdym turnieju. Funkcja rozpoczyna od utworzenia pustej listy, która będzie przechowywać zwycięzców poszczególnych turniejów. Następnie iteruje przez populację, przeprowadzając tyle turniejów, ile wynosi liczba osobników w populacji. W każdej iteracji losowo wybierana jest grupa uczestników o określonym rozmiarze z populacji. Pierwszy uczestnik z listy uczestników jest początkowo uznawany za najlepszego, a jego dostosowanie jest pobierane ze słownika. Następnie funkcja iteruje przez pozostałych uczestników turnieju. Dla każdego uczestnika pobierane jest jego dostosowanie. Jeśli dostosowanie bieżącego uczestnika jest lepsze niż dotychczasowego najlepszego uczestnika, to bieżący uczestnik staje się nowym najlepszym uczestnikiem. Po zakończeniu wewnętrznej pętli, najlepszy uczestnik turnieju jest dodawany do listy zwycięzców. Po zakończeniu wszystkich turniejów, funkcja zwraca listę zwycięzców, która zawiera wybranych osobników do dalszej ewolucji.
* **ruletkowa – opis kodu**
* Liczymy sumaryczną wartość dostosowania w całej populacji poprzez  
  ***suma\_dostosowania = sum(wyniki\_dostosowania\_dict.values())***wyniki\_dostosowania\_dict to słownik pythonowy zawierający wyniki dostosowania dla poszczególnych chromosomów.  
  Wykorzystujemy go ponieważ obliczyliśmy już wcześniej fitness dla chromosomów i nie chcemy go wyliczać za każdym razem od nowa
* Tworzymy nową pustą listę wybrane = []. Będziemy dodawać do tej listy kolejne chromosomy wybrane w selekcji
* Uruchamiamy zewnętrzną pętle:  
  ***for \_ in range(len(populacja))***Range zwróci listę wszystkich licz całkowitych od zera do ilości chromosomów w populacji (wyłącznie) w ten sposób wykonamy dokładnie tyle iteracji ile jest chromosomów. Ponieważ w każdej iteracji, chcemy wybrać dokładnie jeden chromosom, więc po zakończeniu działania pętli będziemy mieć wybrane dokładnie tyle samo chromosomów co w obecnej populacji.
* Wewnątrz każdej iteracji zewnętrznej losujemy dowolną liczbę pomiędzy 0 a suma\_dostosowania poprzez ***prog = random.uniform(0, suma\_dostosowania)***  
  Wylosowana liczba reprezentuje losowanie ruletki.  
  Dodatkowo inicjalizujemy zmienną ***akumulacja = 0***Będziemy do tej zmiennej dodawać po kolei fintess każdego z chromosomów.  
  Chromosom, przy którego iteracji w pętli wewętrznej akumulacja przekroczy, wylosowaną liczbę ***prog,*** zostaje wybrany w selekcji
* Przechodzimy teraz po kolej każdy chromosom w populacji w pętli wewnętrznej poprzez  
  ***for* *chromosom in populacja:***Zwiększamy zmienną ***akumulacja*** o fintness chromosu z bieżacej iteracji.  
  Sprawdzamy warunek czy akumulacja przekorzyczyła ***prog,*** jeżeli tak wybieramy chromosom poprzez dodanie do listy ***wybrane*** oraz przerywamy pętlę zewnętrzną poprze ***break***
* Pod koniec działania, mamy w zmiennej ***wybrane*** listę wszystkich wybranych chromosomów w selekcji ruletkowej, zwracamy tą listę jako wynik działania funkcji.

Mutacja – opis kodu

* Przechodzimy w pętli kolejne indeksy chromosomu
* W każdej iteracji losujemy liczbę od 0 do 1 ( ***random.random()*** ), jeżeli liczba jest mniejsza niż prawdopodobieństwo mutacji, dokonujemy mutacji. Zmieniamy 0 na 1 w chromosomie lub 1 na 0 (mutacja dwustronna) poprzez ***chromosom[i] = 1 – chromosom[i]***
* Pod koniec działania petli zwracamy ewentualnie zmutowany chromosom

**1. Porównanie metod selekcji**

**Analiza wyników:**

**Selekcja turniejowa:**

* Ma większą kontrolę nad presją selekcyjną (poprzez rozmiar turnieju). Im większy rozmiar turnieju, tym większa presja selekcyjna, co może prowadzić do szybszej konwergencji, ale mniejszej różnorodności w populacji.
* Czas działania może być większy, zwłaszcza przy dużych populacjach, ponieważ wymaga porównywania wielu chromosomów.
* **Selekcja ruletki:**
* Jest szybsza w wykonaniu, ale mniej precyzyjna, szczególnie gdy wartości dostosowania chromosomów w populacji są bardzo podobne. Może prowadzić do stagnacji, jeśli różnice w dostosowaniu nie są wyraźne.
* Wymaga normalizacji funkcji przystosowania, aby uniknąć problemów z bardzo małymi lub bardzo dużymi wartościami.

Wnioski:

* Jeśli zależy Ci na szybkiej selekcji w dużych populacjach, selekcja ruletki jest lepsza.
* Jeśli priorytetem jest eksploracja i znalezienie lepszych rozwiązań w dłuższym czasie, selekcja turniejowa daje więcej kontroli nad presją selekcyjną.

**Wpływ mutacji na różnorodność populacji**

Analiza wyników:

Różnorodność:

* Mutacja punktowa pozwala na utrzymanie różnorodności w populacji. Zależność różnorodności od prawdopodobieństwa mutacji jest kluczowa:
* Niskie prawdopodobieństwo mutacji (np. 0.01–0.05) pozwala na subtelne zmiany, ale różnorodność szybko maleje.
* Wysokie prawdopodobieństwo mutacji (np. 0.1–0.2) może prowadzić do bardziej losowych zmian, utrzymując różnorodność, ale zwiększając ryzyko zniszczenia dobrych rozwiązań.

Stabilność:

* Przy zbyt niskiej różnorodności populacja może stagnować (lokalne minima), co wymaga większego nacisku na mutację.

Wnioski:

* Optymalne prawdopodobieństwo mutacji zależy od problemu:
* W przypadku bardziej złożonych problemów zaleca się utrzymanie umiarkowanego poziomu mutacji (np. 0.1), aby uniknąć stagnacji.
* Dla prostych problemów niskie prawdopodobieństwo (np. 0.01–0.05) może wystarczyć.
* Mutacja jest kluczowa dla utrzymania różnorodności, co zapobiega stagnacji w algorytmie genetycznym. Nadmierna różnorodność może jednak spowalniać konwergencję do dobrego rozwiązania.

Podsumowanie ogólne:

Która selekcja jest lepsza?

Jeśli priorytetem jest szybkość — selekcja ruletki.

Jeśli jakość i kontrola — selekcja turniejowa.

Wpływ mutacji:

Umiarkowane prawdopodobieństwo mutacji (np. 0.1) jest kluczowe dla utrzymania różnorodności i unikania lokalnych minimów.

* Niska różnorodność prowadzi do szybkiej konwergencji, ale może ograniczać jakość końcowego rozwiązania.
* Czy chcesz, abym przygotował wizualizacje (np. wykresy porównujące różnorodność, czas działania czy sumę dostosowania dla metod selekcji i mutacji)?