Metaheurystyki i ich zastosowania 2023/24

Zadanie 3 – algorytm genetyczny

Autorzy:
Michał Ferdzyn 242383
Artur Grzybek 242399

1. Zasada działania programu

Program opiera się na implementacji algorytmu genetycznego dla problemu plecakowego.

Działanie programu:

1. **Określenie parametrów algorytmu:**
    - *rozmiar_populacji*: Określa liczbę osobników w populacji początkowej.
    - liczba_iteracji: Określa, ile iteracji (pokoleń) algorytm genetyczny ma przeprowadzić.
    - *prawdopodobienstwo_krzyzowania*: Określa prawdopodobieństwo krzyżowania (crossover) między osobnikami.
    - *prawdopodobienstwo_mutacji:* Określa prawdopodobieństwo mutacji genów w populacji. Pozwala zwiększyć różnorodność potomstwa oraz rozwiązań.
    - *ruletka:* Decyduje, czy selekcja rodziców ma być ruletkowa (True) czy elitarna (False).
    - *czy_jednopunktowe:* Decyduje, czy krzyżowanie genów ma być jednopunktowe (True) czy dwupunktowe (False).
2. **Inicjalizacja populacji:**
    - Populacja początkowa składa się z losowych osobników reprezentujących rozwiązania problemu. Każdy osobnik to ciąg genów (0 lub 1), gdzie długość ciągu odpowiada liczbie przedmiotów (26). To kodowanie odpowiada za to, które przedmioty są wybrane (1) lub niewybrane (0) do umieszczenia w plecaku.
3. **Funkcja przystosowania:**
    - Funkcja *oblicz_przystosowanie(osobnik)* ocenia wartość przystosowania danego osobnika. Wartość przystosowania to suma wartości przedmiotów, które mieszczą się w ograniczeniach wagowych.
4. **Selekcja rodziców:**
    - Funkcje *selekcja_ruletkowa* i *selekcja_elitarna* wybierają rodziców na podstawie przystosowania. Ruletka stosuje selekcję opartą na prawdopodobieństwie, a selekcja elitarna wybiera lepszą połowę populacji.
5. **Krzyżowanie genów:**
    - Funkcja *krzyzowanie_genow* przeprowadza krzyżowanie genów dwóch rodziców, tworząc potomstwo. W celu realizacji programu na ocenę 5 wprowadziliśmy możliwość krzyżowania jednopunktowego jak i dwupunktowego.
6. **Mutacja:**

- Funkcja mutuj_populacje losowo mutuje geny osobników z pewnym prawdopodobieństwem.

7. **Nowe pokolenie:**
   - Nowa populacja powstaje poprzez krzyżowanie genów i mutację.
8. **Warunek zakończenia działania programu:**
   - Ilość Iteracji: Algorytm wykonuje określoną liczbę iteracji *(liczba_iteracji).* Jeśli warunek zakończenia nie zostanie spełniony w żadnej z iteracji, algorytm kończy działanie po przeprowadzeniu wszystkich iteracji.
9. **Tworzenie wykresów:**
   - Po zakończeniu pracy programu tworzy się wykres ukazujący najlepsze, najgorsze i średnie wyniki dla badanych parametrów.

2. Wybrane miejsca implementacji rozwiązania

Algorytm genetyczny – implementacja

```
def algorytm_genetyczny(rozmiar_populacji=30,
                        liczba_iteracji=30,
                        prawdopodobienstwo_krzyzowania=0.8,
                        prawdopodobienstwo_mutacji=0.4,
                        ruletka=True,
                        jednopunktowe=False):
    populacja = []
    for i in range(rozmiar_populacji):
        populacja.append(losowy_osobnik())
    for i in range(liczba_iteracji):
        ocalali, rodzice = wybierz_rodzicow(populacja, prawdopodobienstwo_krzyzowania, ruletka)
        pary = wybierz_pary(rodzice)
        dzieci = mutuj_populacje(nowa_generacja(pary, jednopunktowe), prawdopodobienstwo_mutacji)
        populacja = ocalali + dzieci
    return populacja
```

- Tworzymy funkcję *algorytm_genetyczny* z określonymi przez nas parametrami, takimi jak rozmiar populacji, liczba iteracji, prawdopodobieństwo krzyżowania, itp.
- Tworzy początkową populację o rozmiarze *rozmiar_populacji*. Każdy osobnik w populacji jest generowany przez funkcję *losowy_osobnik().*
- Następnie w głównej pętli algorytmu wywołujemy funkcję *wybierz_rodzicow*, aby dokonać selekcji rodziców na podstawie przystosowania populacji. Parametry *prawdopodobienstwo_krzyzowania* i *ruletka* wpływają na sposób selekcji.
- Wywołujemy funkcję *wybierz_pary* do utworzenia par rodziców z wybranych rodziców.
- Następnie korzystamy z funkcji *nowa_generacja* do krzyżowania genów i tworzenia nowej generacji na podstawie par rodziców. Następnie mutuje tę nową generację na podstawie określonego prawdopodobieństwa mutacji.
- Aktualizujemy populację, dodając do niej ocalałych osobników (jeżeli są) i nowo utworzone dzieci.
- Na końcu algorytmu implementujemy warunki zakończenia programu: poprzez skończoną liczbę iteracji, zwraca populację jako wynik.

Generowanie losowego osobnika

```
def losowy_osobnik():
    return urandom(26)
```

- Funkcja *urandom* pochodząca z modułu *bitarray.util* generuje losową sekwencję bitów o określonej długości. W tym przypadku, generuje 26 losowych bitów, co odpowiada liczbie przedmiotów (genów), z których będzie składał się każdy osobnik.

Obliczenie przystosowanie

```
def oblicz_przystosowanie(osobnik):
    suma_wagi = 0
    suma_wartosci = 0
    for i in range(len(osobnik)):
        if osobnik[i]:
            suma_wagi += DATA[i][1]
            suma_wartosci += DATA[i][2]
    if suma_wagi > BAG_MAX_WEIGHT:
        return 0
    else:
        return suma_wartosci
```

- Zmienne *suma_wagi i suma_wartosci* są inicjalizowane na wartość 0. Używamy ich do śledzenia sumy wag i wartości przedmiotów, które są uwzględnione w danym osobniku.
- Pętla iteruje przez każdy gen (bit) w sekwencji osobnik. Jeśli dany gen ma wartość 1, co oznacza, że przypisuje przedmiot do plecaka, to aktualizowane są sumy wag i wartości na podstawie danych przedmiotów znajdujących się w DATA. (*DATA[i][1]* to waga i-tego przedmiotu natomiast *DATA[i][2]* to wartość i-tego przedmiotu)
- Następnie sprawdzamy, czy suma wag przypisanych przedmiotów przekracza maksymalną wagę plecaka (BAG_MAX_WEIGHT). Jeśli tak, to funkcja zwraca 0, co oznacza, że osobnik ten nie jest dopuszczalny z uwagi na przekroczenie limitu wagi.
- Jeżeli osobnik spełnia ograniczenia wagi plecaka, funkcja zwraca sumę wartości przypisanych przedmiotów, co jest miarą przystosowania tego osobnika. Wartość ta będzie używana do oceny jakości danego rozwiązania.

Obliczenie średniego, minimalnego oraz maksymalnego przystosowania

```python
def oblicz_sume_przystosowania_populacji(populacja):
    suma = 0
    for osobnik in populacja:
        suma += oblicz_przystosowanie(osobnik)
    return suma

# Funkcja zwracająca średnie przystosowanie osobnika z populacji
michalf1703
def oblicz_srednie_przystosowanie_populacji(populacja):
    return oblicz_sume_przystosowania_populacji(populacja) / len(populacja)

michalf1703
def oblicz_maksymalne_przystosowanie_populacji(populacja):
    maksimum = 0
    for osobnik in populacja:
        if maksimum < oblicz_przystosowanie(osobnik):
            maksimum = oblicz_przystosowanie(osobnik)
    return maksimum

michalf1703
def oblicz_minimalne_przystosowanie_populacji(populacja):
    if not populacja:
        return None
    return min(oblicz_przystosowanie(osobnik) for osobnik in populacja)
```

- Te cztery funkcje są związane z oceną przystosowania populacji, czyli z badaniem, jak dobrze osobniki w populacji radzą sobie w rozwiązaniu problemu plecakowego.
- Funkcja *oblicz_sume_przystosowania_populacji*, odpowiada za obliczanie sumy przystosowań wszystkich osobników w danej populacji. Zaimplementowana jest pętla, która iteruje przez każdego osobnika w populacji i następnie wywołuje funkcję *oblicz_przystosowanie(osobnik),* która zwraca wartość przystosowania tego konkretnego osobnika. Po zakończeniu pętli zwracana jest wartość, która jest obliczoną sumą przystosowań wszystkich osobników w populacji.
- Funkcja *oblicz_srednie_przystosowanie_populacji* zwraca średnie przystosowanie osobnika w populacji. Wykorzystuje funkcję *oblicz_sume_przystosowania_populacji,* aby uzyskać sumę przystosowań w populacji, a następnie dzieli tę sumę przez liczbę osobników w populacji.
- Funkcja *oblicz_maksymalne_przystosowanie_populacji* zwraca maksymalne przystosowanie wśród osobników w populacji. Iteruje przez wszystkie osobniki w populacji, używając funkcji *oblicz_przystosowanie* do obliczenia przystosowania każdego osobnika, a następnie zapisuje maksymalną wartość.
- Funkcja *oblicz_minimalne_przystosowanie_populacji* zwraca minimalne przystosowanie wśród osobników w populacji. Podobnie jak w poprzedniej funkcji, iteruje przez wszystkie osobniki, używając funkcji *oblicz_przystosowanie* do obliczenia przystosowania każdego osobnika, a następnie zapisuje minimalną wartość.

Selekcja ruletkowa

```
def selekcja_ruletkowa(populacja):
    suma_przystosowania = oblicz_sume_przystosowania_populacji(populacja)
    if suma_przystosowania == 0:
        return populacja
    tabela_prawdopodobienstw = {}
    for i in range(len(populacja)):
        tabela_prawdopodobienstw[i] = oblicz_przystosowanie(populacja[i]) / suma_przystosowania
    indeksy = random.choices(list(tabela_prawdopodobienstw.keys()),
                             weights=list(tabela_prawdopodobienstw.values()),
                             k=len(populacja))
    return [populacja[indeks] for indeks in indeksy]
```

- *suma_przystosowania*: Oblicza sumę przystosowań wszystkich osobników w populacji za pomocą funkcji *oblicz_sume_przystosowania_populacji*.
- *if suma_przystosowania == 0*: Jeśli suma przystosowania wynosi zero, oznacza to, że wszystkie osobniki w populacji mają zerowe przystosowanie. W takim przypadku funkcja zwraca populację niezmienioną.
- *tabela_prawdopodobienstw*: Tworzymy słownik, w którym kluczami są indeksy osobników, a wartościami są ich prawdopodobieństwa wyboru.
- *Pętla for i in range(len(populacja)):* Iteruje przez wszystkie osobniki w populacji.
- *tabela_prawdopodobienstw[i] = oblicz_przystosowanie(populacja[i]) / suma_przystosowania*: Obliczamy prawdopodobieństwo wyboru każdego osobnika i zapisuje je w tabeli.
- *indeksy = random.choices(list(tabela_prawdopodobienstw.keys()), weights=list(tabela_prawdopodobienstw.values()), k=len(populacja)):* Wybieramy indeksy osobników zgodnie z ich prawdopodobieństwami wyboru przy użyciu funkcji *random.choices.*
- *return [populacja[indeks] for indeks in indeksy]:* Zwracamy nową populację, w której osobnicy są wybierani za pomocą ruletki na podstawie obliczonych prawdopodobieństw.

Selekcja elitarna

```
def selekcja_elitarna(populacja):
    tabela_rankingowa = {}
    for i in range(len(populacja)):
        tabela_rankingowa[i] = oblicz_przystosowanie(populacja[i])
    indeks = sorted(tabela_rankingowa.items(), key=lambda item: -item[1])
    lepsza_polowa = [populacja[indeks[i][0]] for i in range(int(len(populacja) / 2))]
    return lepsza_polowa + lepsza_polowa
```

- *tabela_rankingowa:* Tworzymy słownik, w którym kluczami są indeksy osobników, a wartościami są ich przystosowania obliczone za pomocą funkcji *oblicz_przystosowanie*.

- *for i in range(len(populacja)):* Iterujemy przez wszystkie osobniki w populacji.
- *tabela_rankingowa[i] = oblicz_przystosowanie(populacja[i]):* Oblicza przystosowanie dla każdego osobnika i zapisuje je w tabeli rankingowej.
- *indeks = sorted(tabela_rankingowa.items(), key=lambda item: -item[1]):* Sortujemy indeksy osobników w tabeli rankingowej według malejącego przystosowania, dzięki czemu najlepszy osobnik ma najwyższy indeks.
- *lepsza_polowa:* Tworzymy listę zawierającą lepszą połowę osobników (o najwyższym przystosowaniu) na podstawie posortowanych indeksów.
- *return lepsza_polowa + lepsza_polowa*: Zwracamy nową populację, w której lepsza połowa osobników została podwójnie powielona. Ta operacja ma na celu zwiększenie szansy na przetrwanie najlepszych osobników w populacji.

Wybór rodzica

```python
def wybierz_rodzicow(populacja, prawdopodobienstwo_krzyzowania, czy_ruletka):
    random.shuffle(populacja)
    liczba_do_wyboru = int(prawdopodobienstwo_krzyzowania * len(populacja))
    wybrani_osobnicy = populacja[:liczba_do_wyboru]
    pozostali_osobnicy = populacja[liczba_do_wyboru:]
    if czy_ruletka:
        return pozostali_osobnicy, selekcja_ruletkowa(wybrani_osobnicy)
    else:
        return pozostali_osobnicy, selekcja_elitarna(wybrani_osobnicy)
```

- *random.shuffle(populacja):* Przetasowujemy populację, aby kolejność osobników była losowa. Jest to istotne, aby uniknąć wprowadzania błędów związanych z ewentualną uporządkowaną strukturą populacji.
- *liczba_do_wyboru = int(prawdopodobienstwo_krzyzowania * len(populacja)):* Obliczamy liczbę osobników, które zostaną wybrane do krzyżowania na podstawie zadanej wartości *prawdopodobienstwo_krzyzowania*.
- *wybrani_osobnicy = populacja[:liczba_do_wyboru]:* Wybieramy osobniki do krzyżowania, biorąc pierwsze *liczba_do_wyboru* elementów z przetasowanej populacji.
- *pozostali_osobnicy = populacja[liczba_do_wyboru:]:* Pozostałe osobniki, które nie wezmą udziału w krzyżowaniu.
- *if czy_ruletka::* Sprawdzamy, czy należy użyć selekcji ruletkowej. Jeśli tak, wybieramy rodziców za pomocą funkcji *selekcja_ruletkowa* na podstawie wcześniej wybranych osobników do krzyżowania.
- else:: Jeśli *czy_ruletka* nie jest spełnione, używa selekcji elitarnej, wybierając rodziców za pomocą funkcji *selekcja_elitarna* na podstawie wcześniej wybranych osobników do krzyżowania.
- *return pozostali_osobnicy, selekcja_ruletkowa(wybrani_osobnicy):* Zwracamy pozostałe osobniki oraz wybranych rodziców na podstawie selekcji ruletkowej lub elitarnej, w zależności od wartości *czy_ruletka.*

Krzyżowanie genów

```
def krzyzowanie_genow(rodzice, czy_jednopunktowe):
    punkt_krzyzowania = random.randint(1, 24)
    punkt_krzyzowania_2 = random.randint(1, 24)
    if punkt_krzyzowania > punkt_krzyzowania_2:
        punkt_krzyzowania, punkt_krzyzowania_2 = punkt_krzyzowania_2, punkt_krzyzowania
    if czy_jednopunktowe:
        return [rodzice[0][:punkt_krzyzowania] + rodzice[1][punkt_krzyzowania:],
                rodzice[1][:punkt_krzyzowania] + rodzice[0][punkt_krzyzowania:]]
    else:
        return [rodzice[0][:punkt_krzyzowania] + rodzice[1][punkt_krzyzowania:punkt_krzyzowania_2] + rodzice[0][punkt_krzyzowania_2:],
                rodzice[1][:punkt_krzyzowania] + rodzice[0][punkt_krzyzowania:punkt_krzyzowania_2] + rodzice[1][punkt_krzyzowania_2:]]
```

- *punkt_krzyzowania = random.randint(1, 24):* Losujemy punkt krzyżowania od 1 do 24. Oznacza to, że geny od 1 do punktu krzyżowania będą pochodziły od pierwszego rodzica, a geny od punktu krzyżowania do końca będą pochodziły od drugiego rodzica. (wybierając punkt krzyżowania od 1 do 24, unika się sytuacji, w której całe geny są zamieniane między rodzicami)
- *punkt_krzyzowania_2 = random.randint(1, 24):* Podobnie jak powyżej, losujemy drugi punkt krzyżowania.
- *if punkt_krzyzowania > punkt_krzyzowania_2::* Upewniamy się, że punkty krzyżowania są posortowane rosnąco.
- *if czy_jednopunktowe::* Sprawdzamy, czy krzyżowanie ma być jednopunktowe. Jeśli tak, to geny są wymieniane tylko na jednym punkcie krzyżowania.
- *return [rodzice[0][:punkt_krzyzowania] + rodzice[1][punkt_krzyzowania:], rodzice[1][:punkt_krzyzowania] + rodzice[0][punkt_krzyzowania:]]:* Zwracamy dwójkę potomków, którzy powstały w wyniku krzyżowania genów. W przypadku krzyżowania jednopunktowego, jedna połowa genów jest od jednego rodzica, a druga połowa od drugiego rodzica. W przypadku krzyżowania dwupunktowego, dwie części genów są wymieniane między rodzicami.

Tworzenie par rodziców z populacji

```
def wybierz_pary(populacja):
    pary = []
    random.shuffle(populacja)

    i = 0
    while i < len(populacja) - 1:
        pary.append([populacja[i], populacja[i + 1]])
        i += 2

    if len(populacja) % 2 == 1:
        pary.append([populacja[len(populacja) - 1], populacja[0]])
    return pary
```

- *random.shuffle(populacja):* Tasujemy populację, aby zmieszać kolejność jej osobników, co jest przydatne, aby losowo wybierać pary.

- *while i < len(populacja) - 1:* Iterujemy przez populację w krokach po 2, tworząc pary kolejnych osobników.
- *pary.append([populacja[i], populacja[i + 1]]):* Dodajemy parę osobników do listy par.
- *if len(populacja) % 2 == 1:* Sprawdzamy, czy populacja ma nieparzystą liczbę osobników. Jeśli tak, dodajemy parę z ostatniego osobnika i pierwszego osobnika (po tasowaniu, dlatego nie są one już kolejnymi osobnikami).
- Funkcja zwraca listę par, gdzie każda para zawiera dwa osobniki przeznaczone do krzyżowania.

Tworzenie nowej generacji

```
def nowa_generacja(pary, czy_jednopunktowe):
    dzieci = []
    for para in pary:
        dzieci += krzyzowanie_genow(para, czy_jednopunktowe)
    return dzieci
```

- Funkcja *nowa_generacja* służy do generowania nowej populacji na podstawie par rodziców.
- *pary*: Lista par rodziców, gdzie każda para zawiera dwa osobniki przeznaczone do krzyżowania.
- *czy_jednopunktowe:* Parametr określający, czy użyć jednopunktowego czy dwupunktowego krzyżowania genów.
- *dzieci:* Lista, która będzie przechowywać potomstwo uzyskane po krzyżowaniu.
- *for para in pary*: Iterujemy przez wszystkie pary rodziców.
- *dzieci += krzyzowanie_genow(para, czy_jednopunktowe):* Dla każdej pary rodziców wywołuje funkcję *krzyzowanie_genow*, która zwraca dwójkę potomków (dzieci) na podstawie rodziców i wybranej metody krzyżowania (jednopunktowego lub dwupunktowego). Otrzymane dzieci są dodawane do listy dzieci.
- *return dzieci:* Funkcja zwraca listę dzieci, która stanowi nową generację populacji uzyskaną poprzez krzyżowanie genów.

Mutacja

```
def mutuj_populacje(populacja, prawdopodobienstwo):
    random.shuffle(populacja)
    for i in range(int(prawdopodobienstwo * len(populacja))):
        populacja[i].invert(random.randint(0, 25))
    return populacja
```

- Funkcja *mutuj_populacje* służy do mutacji populacji, czyli losowej zmiany genów w niektórych osobnikach.
- *populacja:* Lista osobników, która ma zostać poddana mutacji.
- *prawdopodobienstwo:* Prawdopodobieństwo mutacji dla każdego osobnika w populacji.
- *random.shuffle(populacja):* Tasujemy populację, aby kolejność osobników nie miała wpływu na proces mutacji.

- *for i in range(int(prawdopodobienstwo * len(populacja))):* Iterujemy przez pewną liczbę osobników, które zostaną poddane mutacji. Ta liczba jest proporcjonalna do całkowitej liczby osobników w populacji i prawdopodobieństwa mutacji.
- *populacja[i].invert(random.randint(0, 25)):* Dla każdego wybranego osobnika (o indeksie i) losowo wybiera jedną pozycję (gen) i zmienia jego wartość na przeciwną (1 na 0 lub 0 na 1).
- *return populacja:* Funkcja zwraca zmutowaną populację. Mutacja polega na odwróceniu losowego genu w wybranych osobnikach, co wprowadza różnorodność genetyczną w populacji.

3. Założenia podstawowe

Metody i operatory genetyczne:

- Losowy osobnik (losowy_osobnik): Generuje osobnika z losowymi 26 genami (reprezentującymi przedmioty).
- Obliczanie przystosowania (oblicz_przystosowanie): Sumuje wartości i wagi przedmiotów, a następnie zwraca wartość przystosowania, uwzględniając ograniczenie wagowe. Określa, jak dobre są poszczególne rozwiązania.
- Selekcja ruletkowa (selekcja_ruletkowa): Wybiera osobniki do krzyżowania zgodnie z metodą ruletki, gdzie prawdopodobieństwo wyboru danej jednostki jest proporcjonalne do jej przystosowania.
- Selekcja elitarna (selekcja_elitarna): Wybiera połowę najlepszych osobników na podstawie przystosowania.
- Krzyżowanie genów (krzyzowanie_genow): Tworzy dwójkę dzieci z dwóch rodziców, wymieniając ich geny w miejscu krzyżowania.
- Wybór par (wybierz_pary): Tworzy pary z populacji przez tasowanie i łączenie kolejnych osobników.
- Nowa generacja (nowa_generacja): Tworzy nową populację na podstawie par rodziców poprzez krzyżowanie genów.
- Mutacja populacji (mutuj_populacje): Losowo zmienia geny wybranych osobników, odwracając jeden z genów.

Warunki stopu:

- Algorytm zakończy działanie po skończonej liczbie pokoleń (iteracji)

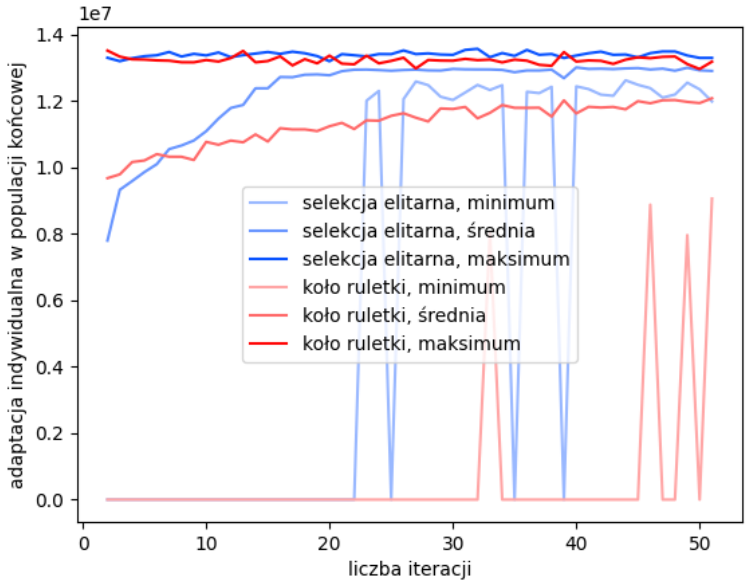
4. Wyniki i analiza

Domyślne ustawienia algorytmu

1 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



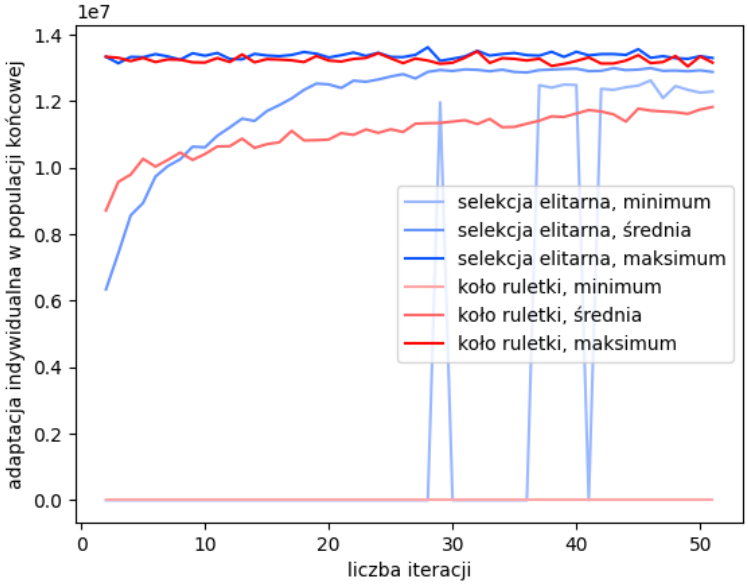
Czas trwania programu: **12 sekund**

Jaki wpływ na uzyskiwane wyniki ma prawdopodobieństwo krzyżowania?

2 uruchomienie programu

Określone parametry:

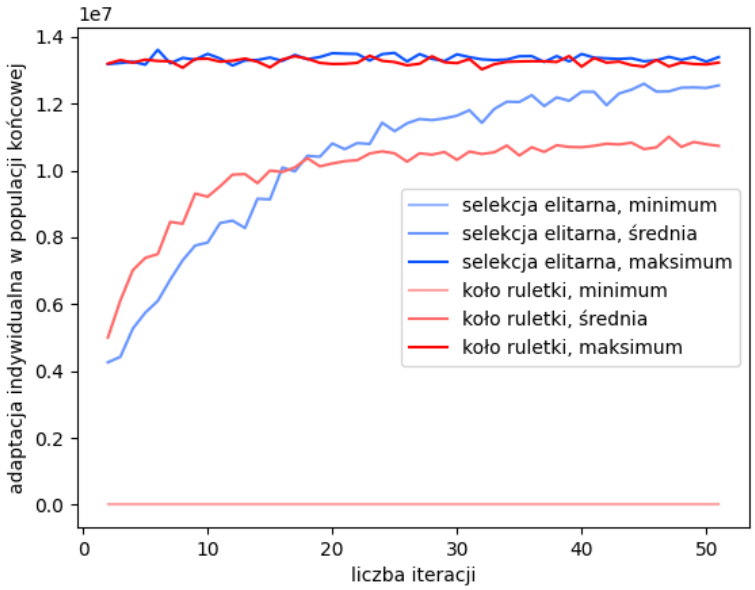
- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.6
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



3 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.2
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



Wnioski

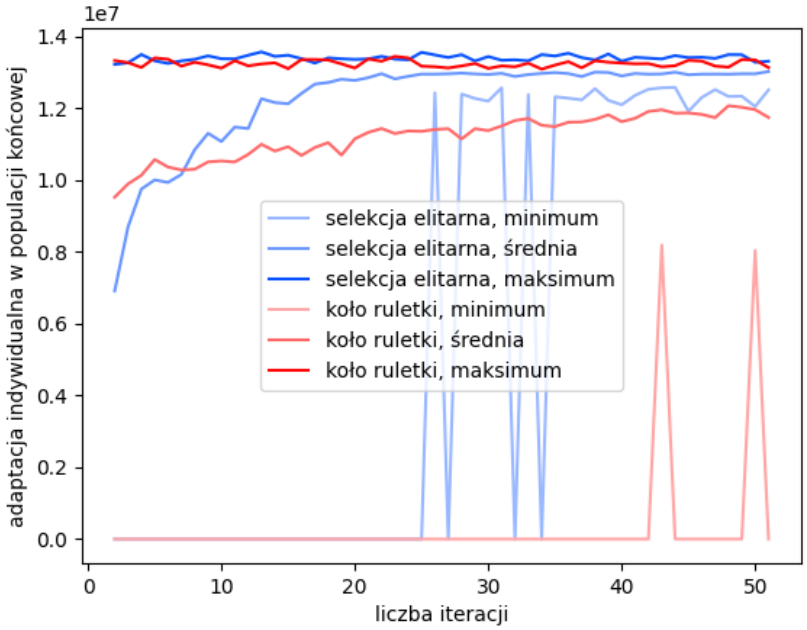
Zauważamy, że wraz zmniejszanie prawdopodobieństwa krzyżówki wpływa na zróżnicowanie rozwiązań w populacji. Im większa wartość tej zmiennej tym zauważalne są „gwałtowniejsze” zmiany na wykresie. Dostrzegamy, też że wartość średniego rozwiązania dla populacji jest bardziej zbliżona do wartości maksymalnych, przy wyższych prawdopodobieństwach. Rozwiązanie minimalne w przypadku *uruchomienia 3* jest równe 0, w każdej iteracji (zarówno dla koła ruletki jak i selekcji elitarnej). Natomiast wraz z zwiększaniem prawdopodobieństwa krzyżowania *(uruchomienie 1 oraz 2)* rozwiązanie minimalne stawało się bardziej zróżnicowane. Stwierdzamy, więc że ten współczynnik powinien mieć stosunkowo wysoką wartość, ponieważ pozytywnie to wpływa na zróżnicowanie genotypów w populacji, zmniejsza szansę na utknięcie algorytmu w optimum lokalnym. Unikamy stagnacji wyników.

Jaki wpływ na uzyskiwane wyniki ma prawdopodobieństwo mutacji?

4 uruchomienie programu

Określone parametry:

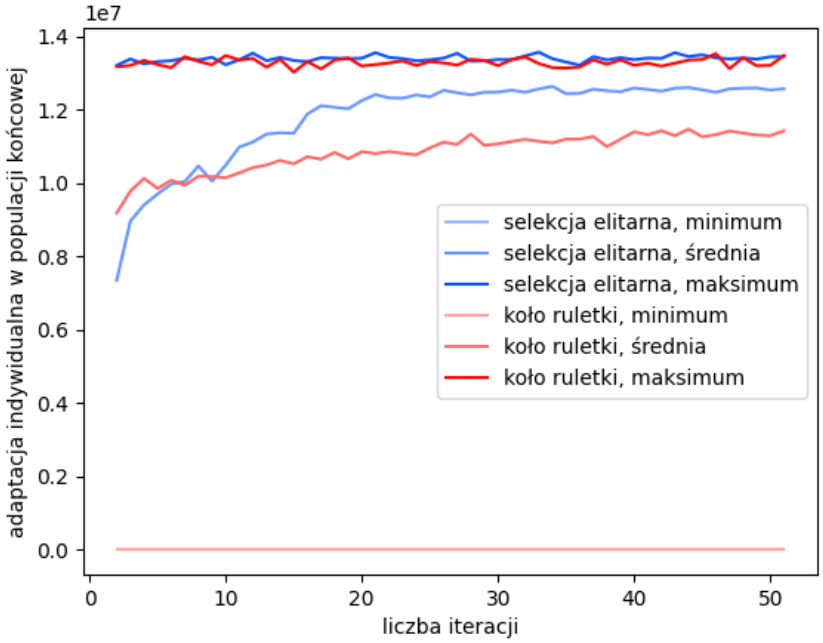
- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.05
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



5 uruchomienie programu

Określone parametry:

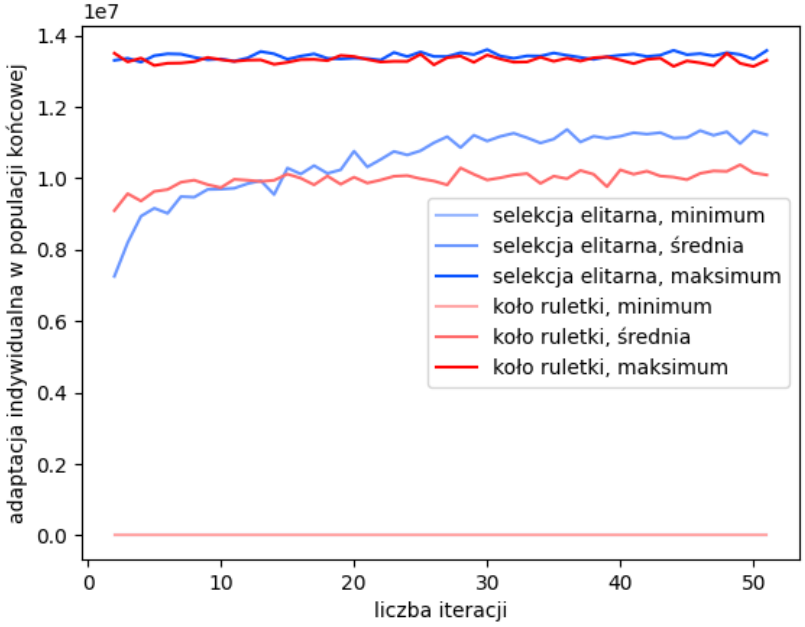
- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.1
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



6 uruchomienie programu

Określone parametry:

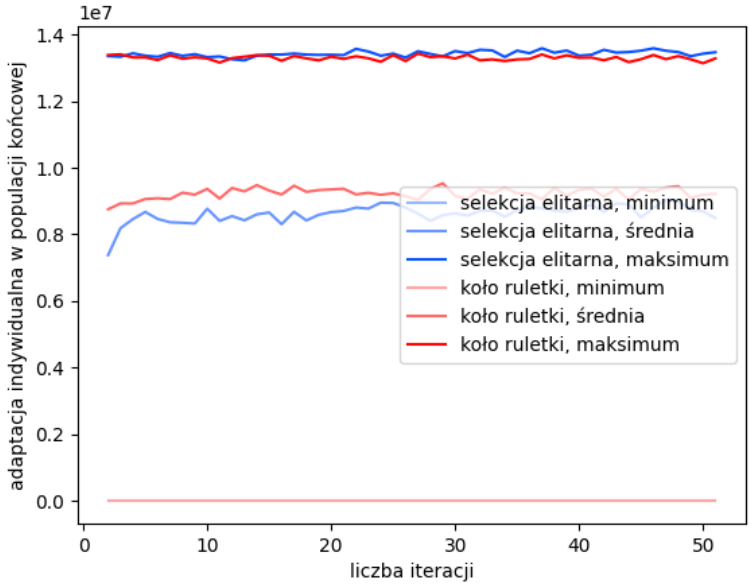
- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.3
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



7 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 20
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.6
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



Wnioski

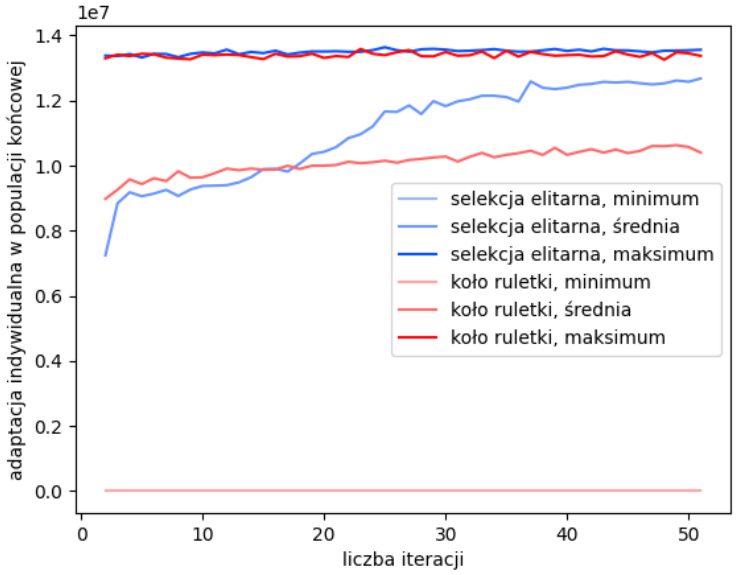
Zauważamy, że wraz ze wzrostem prawdopodobieństwa mutacji stabilizują się minimalne, średnie oraz maksymalne rozwiązania dla populacji. Warto jednak podkreślić, iż nasz program działa w ten sposób, że algorytm jest uruchamiany 50 razy (dla każdego uruchomienia następuje 30 pokoleń) i rezultaty dla każdego kolejnego uruchomienia przedstawiamy na naszych wykresach (jako liczba iteracji). W celu prześledzenia zachowania rozwiązań dla kolejnych pokoleń lepszym sposobem byłoby sporządzenie wykresu dla jednego uruchomienia całego algorytmu. Wówczas bardziej zauważalne byłby ogromy wpływ prawdopodobieństwa mutacji na rozwiązania maksymalne (następowałyby nagłe zmiany najlepszego rezultatu w kolejnych pokoleniach). Na podstawie sporządzonych przez nas wykresów możemy jednak stwierdzić, że wraz zwiększaniem wartości tego współczynnika wyniki każdego uruchomienia są coraz bardziej zbliżone do siebie.

Jaki wpływ na uzyskiwane wyniki ma wielkość populacji?

8 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 60
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50

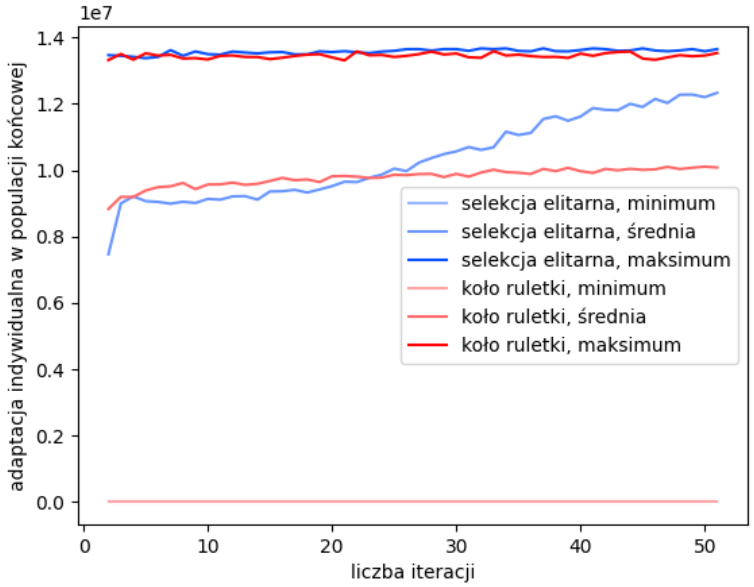


Czas trwania programu: **34 sekund**

9 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 150
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie jednopunktowe
- Liczba uruchomień algorytmu = 50



Czas trwania programu: **1 minuta 20 sekund**

Wnioski

Zwiększanie wielkości populacji w bardzo dużym stopniu wpływa na czas pracy algorytmu. Z pewnością potrzeba więcej zasobów obliczeniowych. Przy uruchomieniu programu z domyślnymi wartościami (wielkość populacji = 20) czas trwania wynosił 12 sekund. Przy ilości populacji równej 60 czas ten wydłużył się do 34 sekund. Natomiast przy ustawieniu wielkości populacji na 150 ten czas wyniósł aż 1 minutę oraz 20 sekund. Porównując rezultaty na wykresach nie dostrzegamy drastycznych zmian związanych ze zwiększeniem tego parametru. Warto jednak podkreślić, iż większa populacja wiąże się oczywiście z lepszą eksploracją przestrzeni, bo istnieje większa szansa na różnorodne genotypy w naszej populacji. Z naszych obserwacji jednak zauważamy, że wartość ta powinna być odpowiednio dostosowana do specyfikacji problemu – przy użyciu zbyt dużej populacji nasze wyniki nie ulegną polepszeniu, a tylko wydłużą czas działania programu. Działa to jednak też w druga stronę – przy zbyt małej populacji możemy nie uzyskać odpowiednio dobrych rezultatów.

Jaki wpływ na uzyskiwane wyniki mają metody selekcji?

Wnioski

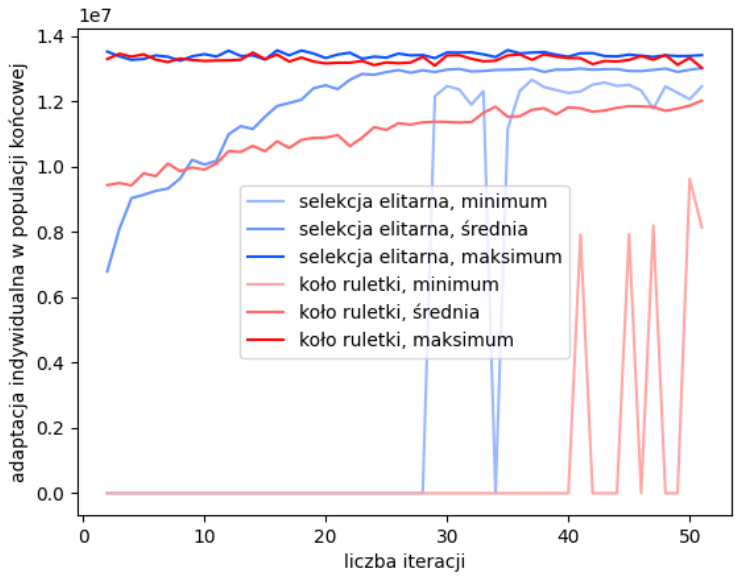
Na każdym przedstawionym przez nas wykresie metoda elitarna oraz koło ruletki zostały ze sobą zestawione, więc na potrzeby analizy tego zagadnienia nie musimy generować nowych rozwiązań – przeanalizujemy poprzednie rezultaty (w szczególności *uruchomienie 1* z domyślnymi wartościami). Selekcja elitarna wiąże się, z tym że wybierana jest pewna liczba najlepiej przystosowanych osobników, a następna grupa reprodukuje się, tworząc kolejną generacje. W przypadku koła ruletki osobniki wybierane są z prawdopodobieństwem proporcjonalnym do ich przystosowania, tym większe szanse na zostanie wybranym. Zauważamy, iż większa stabilizacja najlepszego wyniku występuje w przypadku selekcji elitarnej. Ten sposób z pewnością zwiększa poziom przystosowania w populacji, ale może prowadzić do mniejszej różnorodności genetycznej. Koło ruletki wykazuje większe wahania poziomu przystosowania, lecz pozwala na poszukiwanie bardziej zróżnicowanych rozwiązań oraz jest w stanie uniknąć zbyt szybkiej zbieżności. Ostatecznym wnioskiem w tym aspekcie jest stwierdzenie, że należy odpowiednio i bardzo ostrożnie wybierać metodę selekcji, tak aby była ona jak najbardziej przystosowana do rozwiązywanego problemu.

Jaki wpływ na uzyskiwane wyniki mają metody krzyżowania?

10 uruchomienie programu

Określone parametry:

- Rozmiar populacji = 150
- Liczba iteracji(pokoleń) = 30
- Prawdopodobieństwo krzyżowania = 0.9
- Prawdopodobieństwo mutacji = 0.01
- Krzyżowanie **dwupunktowe**
- Liczba uruchomień algorytmu = 50



Wnioski

Analizując wykres z *uruchomienia 10* oraz wykres z *uruchomienia 1* (domyślne wartości), zauważamy że w przypadku krzyżowania dwupunktowego rezultaty gwałtowniej się zmieniają. Stosując ją jesteśmy w stanie zwiększyć różnorodność genetyczna, wprowadzając większą zmienność w strukturze dzieci (korzystny wpływ na unikanie lokalnych optimów). Natomiast krzyżówka jednopunktowa skutkuje szybsza zbieżnością algorytmu (może prowadzić do utknięcia w optimum lokalnym).