Metody obliczeniowe optymalizacji

Optymalizacja przy użyciu metod metaheurystycznych

Wstęp

Do tej pory na zajęciach laboratoryjnych rozwiązywaliśmy problemy metodami deterministycznymi, które działają w ustalony matematycznie sposób i nie zawierają elementów losowości. Istnieją również metody metaheurystyczne, które polegają na niedeterministycznym (zawierającym losowość) poszukiwaniu rozwiązań. Wiele z nich, tzw. metody populacyjne, jest inspirowanych naturą i polega na tworzeniu populacji osobników, które reprezentują rozwiązanie problemu, a następnie modyfikowanie ich w kolejnych epokach w celu ulepszenia i zbliżenia się do rozwiązania optymalnego.

W metodach deterministycznych możliwe jest wprowadzenie tzw. twardych ograniczeń, które nie mogą zostać naruszone. Korzystając z metod metaheurystycznych takie ograniczenia możemy nałożyć jedynie na zmienne decyzyjne, natomiast nie ma możliwości podawania ograniczeń funkcyjnych w stylu macierzy A, b, Aeq, Beq znanych z funkcji *linprog* w Matlabie. Zamiast tego stosuje się tzw. miękkie ograniczenia, które polegają na modyfikacji funkcji celu, tak aby „karała" za wychodzenie poza dopuszczalny obszar.

Najbardziej znane, klasyczne metody metaheurystyczne to Algorytm Genetyczny (ang. Genetic Algorithm, GA) oraz Optymalizacja Rojem Cząstek (ang. Particle Swarm Optimization, PSA). W tym ćwiczeniu zajmiemy się jednak mniej znanymi, nowszymi metodami.

Ogólne założenia

Zadanie polega na przetestowaniu trzech stochastycznych (zawierających elementy losowe) metod optymalizacji znajdujących się w module MEALPY języka Python. Moduł ten należy pobrać i zainstalować przed rozpoczęciem pracy (najlepiej w najnowszej wersji lub co najmniej 3.03).

Do testów należy wybrać trzy metody spośród 12 według skryptu przedstawionego na stronie 2.

Metody w MEALPY posiadają dwa główne parametry (*epoch* – liczba epok oraz *pop_size* – rozmiar populacji). Parametr *epoch* powinien być ustawiony na 1000, przy czym należy zastosować mechanizm wczesnego zatrzymania (MSC – Maximum Stagnation Count – dodatkowy parametr zakończenia), który przerywa działanie algorytmu, jeśli w ciągu ostatnich 20 epok nie nastąpi poprawa wyników.

Każdą metodę należy uruchomić dla parametru *pop_size* zmieniające się z w zakresie [5-100] z krokiem 5. Dla każdego testu należy wyznaczyć wartość funkcji celu (zwanej również funkcją przystosowania, ang. fitness function, w przypadku metod metaheurystycznych) oraz czas optymalizacji. Wartości i czasy należy przedstawić na wykresie. W testach należy ustawić ziarno losowości (ang. seed), aby wyniki były powtarzalne mimo losowego charakteru metod.

Dla każdej metody należy wyznaczyć najlepszy wynik (pod względem funkcji celu) i sprawdzić:

1. dla jakiej wartości parametru *pop_size* został uzyskany;
2. czas optymalizacji dla tego przypadku;
3. liczbę epok, po których zakończył się algorytm.

Na koniec należy utworzyć ranking (na podstawie wartości funkcji celu), tzn. określić, która metoda wygrała, która zajęła drugie i trzecie miejsce. We wnioskach należy opisać jak duże były różnice oraz jak różniły się czasy optymalizacji i liczby epok dla zwycięskich przypadków.

Na ostatniej stronie przedstawione są dwa problemy optymalizacyjne. Na jednym z nich należy przetestować wybrane metody. Jeśli Twoje nazwisko ma parzystą liczbę liter, to wybierz problem 1; w przeciwnym wypadku wybierz problem 2. Prowadzący dostarczy pliki z danymi obu problemów.

Na podstawie niniejszego ćwiczenia należy wykonać i przesłać prowadzącemu (dawwar@prz.edu.pl) przed kolejnymi zajęciami. Przekroczenie tego terminu spowoduje zaniżenie oceny końcowej z laboratorium o 0.5 stopnia. W sprawozdaniu należy przedstawić kod (na podstawie którego przeprowadzono testy), wszystkie wyniki (liczby, tabele, wykresy) oraz wnioski. Dodatkowo, należy zamieścić krótki opis (4-6 zdań) metody, która okazała się zwycięska.

Wybór metody

Skrypt (Python) wyboru metod:

```
while len(ind := input('Podaj swój numer indeksu:')) < 6:
    print('Niepoprawny numer indeksu.')

method_1 = (int(ind[2]) + int(ind[3])) // 4 + 1
method_2 = (int(ind[3]) + int(ind[4])) // 4 + 5
method_3 = (int(ind[4]) + int(ind[5])) // 4 + 9

print(f'Twoje metody, to: {method_1, method_2, method_3}.')
```

Numery metod:

1. **SSA** (*Salp Swarm Algorithm*) – Inspirowany zachowaniem stadnym i sposobem poruszania się salp (morskich bezkręgowców), które tworzą w oceanach charakterystyczne łańcuchy w celu lepszej nawigacji i żerowania.
2. **HHO** (*Harris Hawks Optimization*) – Inspirowany unikalnym, kooperatywnym zachowaniem jastrzębi Harrisa, które stosują różne warianty taktyki „zaskakującego ataku” (ang. *surprise pounce*) w celu osaczenia ofiary.
3. **GWO** (*Grey Wolf Optimizer*) – Oparty na ścisłej hierarchii społecznej oraz stadnym mechanizmie polowania wilków szarych, gdzie proces decyzyjny i poszukiwanie celu są sterowane przez liderów (osobniki alfa, beta i delta).
4. **SCSO** (*Sand Cat Swarm Optimization*) – Inspirowany zachowaniem kotów pustynnych (wykrywanie niskoczęstotliwościowych drgań generowanych przez ofiary).
5. **ALO** (*Ant Lion Optimizer*) – Inspirowany mechanizmem polowania mrówkolwa.
6. **MFO** (*Moth Flame Optimization*) – Oparty na sposobie nawigacji ciem w stosunku do źródła światła.
7. **FOA** (*Fruit-fly Optimization Algorithm*) – Inspirowany zmysłem węchu i wzroku muszek owocówek.
8. **MPA** (*Marine Predators Algorithm*) – Inspirowany strategiami przetrwania drapieżników morskich.
9. **HBA** (*Honey Badger Algorithm*) – Oparty na inteligentnym zachowaniu miodożera podczas szukania pożywienia.
10. **TSO** (*Tuna Swarm Optimization*) – Modelujący zachowanie stadne tuńczyków.
11. **MGO** (*Mountain Gazelle Optimizer*) – Inspirowany zachowaniem społecznym gazeli górskich.
12. **WOA** (*Whale Optimization Algorithm*) – Inspirowany techniką polowania wielorybów (sieć bąbelkowa).

1. Problem plecakowy

Opis problemu

Przy podanym zbiorze elementów o podanej wadze i wartości, należy wybrać taki podzbiór by suma wartości była możliwie jak największa, a suma wag była nie większa od danej pojemności plecaka.

Format danych wejściowych

```
<pojemność-plecaka>

<objętość-przedmiotu-1> <wartość-przedmiotu-1>
<objętość-przedmiotu-2> <wartość-przedmiotu-2>
<objętość-przedmiotu-3> <wartość-przedmiotu-3>
…
<objętość-przedmiotu-N> <wartość-przedmiotu-N>
```

2. Bin-packing

Opis problemu

Mając dowolną ilość kubełków o jednakowych pojemnościach oraz określoną ilość przedmiotów, każdy o określonym rozmiarze, należy znaleźć taki sposób umieszczenia przedmiotów w kubełkach, aby użyć do tego celu jak najmniejszej ilości kubełków.

Przykładowy format danych wejściowych

```
<pojemność-kubełków>
<ilość-przedmiotów>
<wielkość-przedmiotu-1>
<wielkość-przedmiotu-2>
…
<wielkość-przedmiotu-N>
```