Ćwiczenie 2

Algorytmy ewolucyjne i genetyczne

Adam Nowakowski (300428)

1. Zadanie

Zadaniem była implementacja metody realizującej *algorytm ewolucyjny bez krzyżowania z selekcją turniejową i sukcesją elitarną.* Następnie należało znaleźć parametry umożliwiające odpowiednio *minimalizację i maksymalizację dwuwymiarowych funkcji celu* f_1 *i* f_2. Trzeba było także zbadać jak zachowuje się algorytm w przypadku zmiany wartości parametru *k sukcesji elitarnej*.

Parametry i funkcje zdefiniowane w zadaniu:

$$f_1(x) = \varphi(x, \mu_1, \Sigma_1) + \varphi(x, \mu_2, \Sigma_2) + \varphi(x, \mu_3, \Sigma_3)$$

$$\varphi(x, \mu, \Sigma) = \frac{\exp\left(-0{,}5(x-\mu)^T \Sigma^{-1}(x-\mu)\right)}{\sqrt{(2\pi)^{\dim(x)}|\Sigma|}}$$

$$f_2(x) = -20\exp\left[-0.2\sqrt{0.5x^T x}\right] - \exp[0.5(\cos 2\pi x_1 + \cos 2\pi x_2)] + e + 20$$

	Rozkład punktów początkowych	Próg	Budżet
f_1(x)	$[N(0,1), N(0,1)]^T$	0.15	10^6
f_2(x)	$[N(3,1), N(3,1)]^T$	1	10^5

gdzie:

- próg – średnia wartość funkcji przy minimalizacji / maksymalizacji musi być lepsza od tej wartości
- budżet – maksymalna ilość wywołań funkcji celu

$$\mu_1 = \begin{bmatrix} 14 \\ -11 \end{bmatrix}, \mu_2 = \begin{bmatrix} 10 \\ -10 \end{bmatrix}, \mu_1 = \begin{bmatrix} 7 \\ -14 \end{bmatrix},$$

$$\Sigma_1 = \begin{bmatrix} 1.3 & -0.5 \\ -0.5 & 0.8 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 1.7 & 0.4 \\ 0.4 & 1.2 \end{bmatrix}, \Sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1.5 \end{bmatrix}$$

2. Wprowadzenie teoretyczne

Algorytmy ewolucyjne to grupa metod optymalizacji inspirowanych ewolucją naturalną, tak samo według działania, jak i nazewnictwa używanego do ich opisywania. Aby zrozumieć algorytm nad którym się pochylany, należy wyjaśnić sobie pojęcia, które będą się powtarzać w tym dokumencie:

- osobnik – jest to najmniejsza jednostka w naszym algorytmie, która ma przypisany pewien zbiór informacji stanowiących jego genotyp. Określa on cechy osobnika, które są poddawane ocenie, tak aby silniejszy (z lepszymi cechami) przeżył, a ten słabszy zginął.
- populacja – zbiór osobników, które istnieją w środowisku zdefiniowanym przez problem podlegający optymalizacji.
- krzyżowanie – proces który zachodzi pomiędzy dwoma osobnikami (rodzicami) aby połączyć niektóre, wybrane losowo, genotypy w jeden potomny. Służy bardziej do eksploatacji przestrzeni przeszukiwań (jak najdokładniejszego zlokalizowania ekstremum), niż do jej eksploracji (znajdowania innego niż bieżące ekstremum lokalnego).
- mutacja – proces, który zachodzi na pojedynczym osobniku rodzicielskim. Polega na małej modyfikacji jego genotypu, aby osobnik potomny był dosyć podobny do rodzicielskiego. Służy to do eksploracji przestrzeni przeszukiwań.
- selekcja – proces zachodzący na całej populacji. Polega na wybraniu tych osobników, które mają najlepsze cechy, aby przeżyły do kolejnej iteracji lub się krzyżowały / mutowały. Ocena cech zachodzi poprzez funkcję oceny, która na podstawie genotypu określa, który osobnik jest lepszy, a który gorszy.

W przypadku naszych funkcji celu $f_1(\mathrm{x})$ i $f_2(\mathrm{x})$, które będziemy badać, naszymi osobnikami będą punkty na przestrzeni trójwymiarowej. Ich genotypem będzie dwuwymiarowy zestaw ich współrzędnych x, a cechą będzie obliczana z funkcji wartość, którą będziemy w zależności od funkcji maksymalizować i minimalizować. Z tego wynika, że nasza funkcja celu będzie w przypadku maksymalizacji „promować" coraz to większe wartości tej funkcji, a w przypadku minimalizacji te mniejsze. Na tej podstawie będziemy decydować o selekcji i mutacji tych osobników.

W naszym algorytmie będziemy implementować selekcję turniejową, która jest rodzajem selekcji rangowej. Selekcja rangowa charakteryzuje się tym, że populację sortuje się ze względu na funkcję oceny a pozycja osobnika to jego ranga. Prawdopodobieństwo wyboru osobnika w takim podejściu ma to do siebie, że osobnik najgorszy ma niezerową szansę, aby przeżyć, a ten najlepszy może nie zostać wybrany i nie przeżyje. Jest to o tyle dobre rozwiązanie, że czasem się może zdarzyć tak, że ten osobnik o słabych cechach może być rodzicem dla jakiegoś świetnego potomka, który przeskoczy ekstremum lokalne do innego, lepszego. Zwiększa to eksplorację przestrzeni przeszukiwań. Podejście turniejowe do selekcji zachowuje cechy rangowego i polega na wstępnym losowaniu z powtórzeniami s osobników do turnieju (gdzie s to rozmiar turnieju) i wybraniu spośród nich tego o najlepszych cechach. Tak wyłonione osobniki zostaną poddane procesowi mutacji.

Osobniki, które przeżyją do następnej iteracji algorytmu wybieramy za pomocą sukcesji elitarnej. Polega ona na wybraniu z populacji sprzed mutacji k osobników najlepszych, którym pozwolimy przeżyć oraz wyrzucenie z populacji zmutowanej k osobników najgorszych. To podejście zwiększa eksploatację przestrzeni przeszukiwań. W takim przypadku k powinno być małą liczbą, aby algorytm nie utknął w ekstremum lokalnym.

3. Implementacja

W katalogu `/implementacja` znajdują się trzy skrypty w kodzie Pythona:

- *`functions.py`* – zawiera implementacje obu funkcji, które będziemy badać, a także funkcję pomocniczą do znalezienia funkcji przeciwnej do danej.
- *`evolution.py`* – zawiera klasę, w której zaimplementowany jest badany algorytm.
- *`evo_funs`*`.py` – właściwy kod algorytmu podzielony na funkcje

Aby rozwiązać problem algorytmem ewolucyjnym, należy utworzyć obiekt reprezentujący implementację algorytmu, przekazując tym samym odpowiednie parametry. Następnie na tym obiekcie należy wywołać funkcję `.maximise()` lub `.minimise()` odpowiednio do maksymalizacji lub minimalizacji funkcji celu, której to obiekt przekazujemy w argumencie tejże metody.

W `.maximise()` jest pełna implementacja algorytmu ewolucyjnego. Na początku inicjalizujemy populację z rozkładu normalnego o parametrach, które podajemy przy tworzeniu instancji klasy `EvolutionaryMethod`. Ta populacja zostaje oceniana (przez funkcję `evo_funs.evaluate()`) i przechodzimy do głównej pętli. Wykonuje się ona dopóki nie przekroczymy danego nam budżetu wywołań funkcji celu. W środku najpierw wybieramy osobniki do mutacji i ją przeprowadzamy. Dzieje się to za pomocą odpowiednio: `evo_funs.tournament_select()` i `evo_funs.mutate()`. Następnie oceniamy populację mutantów i dzięki funkcji `evo_funs.elite_succession()` wybieramy osobniki, które przeżywają do następnej iteracji. Podczas działania algorytmu zbierana jest informacja o średniej wartości funkcji celu po każdej iteracji w tablicy `average_results`. Wynikiem jest średnia wartość funkcji celu po ostatniej iteracji.

Metoda `.minimise()` jest wywołaniem `.maximise()` z flagą `maximise = False`.

4. Znajdowanie odpowiednich parametrów dla algorytmu

W katalogu `/implementacja` znajduje się skrypt `tests.py`, w którym badam parametry funkcji celu f_1 i f_2, aby dobrać odpowiednie wartości.

Pierwszą badałem funkcję f_1. Na początku ustawiłem:

	parametr	wartość
f_1	p_size	5000
	mutation_sigma	0.7
	elite_succession	3
	tournament_size	2

Dało to następujące wyniki:

	seed	wartość
f_1	342	0.0850
	12312	0.0840
	1	0.0843
	12333	0.0840
	900	0.0841

Widzimy, że te wyniki są niewystarczające. Algorytm utknął w optimum lokalnym ~ 0.085 Przy dalszych badaniach nie udało mi się osiągnąć wymaganego progu, najlepsze wyniki jakie dostałem, to:

	seed	wartość
f_1	342	0.1153
	12312	0.1152
	1	0.1152
	12333	0.1152
	900	0.1149
	112	0.1150
	997	0.1150
	111111	0.1152

Przy czym zmniejszyłem wartość budżetu do $3 * 10^5$ gdyż algorytm już wcześniej się ustalał w tym optimum. Parametry, które dały mi takie wyniki wyglądają następująco:

	parametr	wartość
f_1	p_size	1000
	mutation_sigma	0.3
	elite_succession	5
	tournament_size	2

Potem zbadałem funkcję f_2. Wyniki prezentują się następująco:

	parametr	wartość
f_2	p_size	500
	mutation_sigma	0.08
	elite_succession	2
	tournament_size	2

	seed	wartość
f_2	342	0.8909
	12312	0.9019
	1	0.8535
	12333	0.8980
	900	0.9281
	112	0.8832
	997	0.9128
	111111	0.8639

Jak widać powyższe wartości sprawdziły się w minimalizacji funkcji f_2

5. Badanie parametru k sukcesji elitarnej.

Przy ustalonych wartościach z poprzedniego punktu badałem jak zmiana parametru k wpływa na wyniki.

Dla funkcji f_1:

parametr	wartość
elite_succession	10

	seed	wartość
f_1	342	0.1156
	12312	0.1149
	1	0.1155
	12333	0.1152
	900	0.1140
	112	0.1156
	997	0.1151
	111111	0.1159

parametr	wartość
elite_succession	100

	seed	wartość
f_1	342	0.1218
	12312	0.1217
	1	0.1212
	12333	0.1217
	900	0.1219
	112	0.1220
	997	0.1217
	111111	0.1097

parametr	wartość
elite_succession	300

	seed	wartość
f_1	342	0.1270
	12312	0.1269
	1	0.1271
	12333	0.1270
	900	0.1271
	112	0.1270
	997	0.1270
	111111	0.1272

parametr	wartość
elite_succession	900

	seed	wartość
f_1	342	0.1303
	12312	0.1302
	1	0.1162
	12333	0.1303
	900	0.1161
	112	0.1301
	997	0.1303
	111111	0.1303

parametr	wartość
elite_succession	970

	seed	wartość
f_1	342	0.0770
	12312	0.0719
	1	0.0680
	12333	0.0705
	900	0.0774
	112	0.0776
	997	0.0708
	111111	0.0680

parametr	wartość
elite_succession	1

	seed	wartość
f_1	342	0.1098
	12312	0.1103
	1	0.1149
	12333	0.1136
	900	0.1145
	112	0.1136
	997	0.1122
	111111	0.1112

Przy badaniu parametru k na funkcji f_1 otrzymałem wyniki, które mnie dosyć zaskoczyły. Wraz ze wzrostem parametru sukcesji elitarnej, zwiększała się jakość naszego wyniku. Co prawda to były nieznaczne zmiany, ale doszedłem do bardzo dużych wartości k, gdzie rozsądnym jest ustawianie tego parametru na małą wartość, by nie tracić na eksploracji. Dopiero jak byłem bardzo blisko wartości wielkości populacji, jakość diametralnie spadała. Działo się tak dlatego, że tylko niewielka część zmutowanych osobników przechodziła do następnej iteracji i przestrzeń przeszukiwań malała. Przy tym badaniu zmniejszyłem budżet do 10^5 ze względu na długi czas oczekiwania na wynik. Nie wpłynęło to na szczęście bardzo na wyniki

Dla funkcji f_2 wyniki badań przedstawiają się następująco:

parametr	wartość
elite_succession	1

	seed	wartość
f_2	342	0.9179
	12312	0.9018
	1	0.8958
	12333	0.8904
	900	0.8638
	112	0.9113
	997	0.9132
	111111	0.9301

parametr	wartość
elite_succession	10

	seed	wartość
f_2	342	0.7822
	12312	0.8572
	1	0.7998
	12333	0.8177
	900	0.7523
	112	0.7730
	997	0.8031
	111111	0.8782

parametr	wartość
elite_succession	50

	seed	wartość
f_2	342	0.5308
	12312	2.8652
	1	0.5435
	12333	0.5611
	900	3.8597
	112	0.6218
	997	2.8439
	111111	0.5638

parametr	wartość
elite_succession	100

	seed	wartość
f_2	342	0.3810
	12312	2.7751
	1	0.3872
	12333	2.7617
	900	3.7423
	112	0.4100
	997	2.7594
	111111	0.4004

parametr	wartość
elite_succession	300

	seed	wartość
f_2	342	0.0972
	12312	2.6142
	1	0.0951
	12333	0.1048
	900	0.0996
	112	0.1033
	997	2.6108
	111111	0.1013

Przy funkcji f_2 zauważamy, że wraz ze wzrostem wartości k, algorytm znajduje nam coraz to lepsze rozwiązanie, ale spada tutaj jego niezawodność. Im większe k, tym dla większej ilości ziaren otrzymujemy niezadawalający nas wynik. Optymalną wartością w tym przypadku zdaje się być $\mathrm{k} = 10$. Widzimy, że dla wszystkich naszych ziaren daje nam bardzo dobry wynik.