Jan Ignatowicz
Daniel Pogrebniak
Daniel Wojdak

Sprawozdanie do projektu z Podstaw sztucznej inteligencji

1. Treść zadania

Celem projektu jest napisanie aplikacji, która umożliwi znalezienie minimum funkcji Rosenbrocka

$$f(x,y)=(1-x)^2+100(y-x^2)^2$$

na przedziale $x\in[-2,2]$, $y\in[-2,2]$

za pomocą algorytmu ewolucyjnego.

2. Podjęte decyzje projektowe

Pierwszą decyzją, którą podjęliśmy był wybór języka programowania, w którym napiszemy program. Wybraliśmy język Java, ponieważ zależało nam, aby program był w pełni konfigurowalny. Pozwoliło nam to na przetestowanie algorytmu z wieloma różnymi parametrami. Kolejną ważną decyzją był wybór zastosowanego algorytmu. Wybraliśmy algorytm $\mu + \lambda$, który jest jednym z najpopularniejszych algorytmów ewolucyjnych. W przy tworzeniu potomków korzystaliśmy ze strategii elitarnej, ponieważ wiemy, że funkcja Rosenbrocka ma tylko 1 minimum globalne i żadnych minimów lokalnych. Pozwoliło nam to na zwiększenie efektywności algorytmu. Dalsze decyzje nie były już tak oczywiste, bo zależały w głównej mierze od tego co w danym eksperymencie chcieliśmy sprawdzić. Nasz program pozwala użytkownikowi samodzielnie podawać parametry, więc testowaliśmy go na różnych konfiguracjach. Zachęcamy użytkownika do samodzielnego testowania programu z własnymi parametrami. W dalszej części sprawozdania opiszemy jedynie wyniki otrzymane dla jednego przykładowego zestawu danych.

Wybrane przez nas wartości liczbowe:

- rozmiar populacji: $\mu = 100$
- ilość potomków w każdej iteracji: $\lambda = 200$
- początkowa σ dla każdego osobnika: $\sigma = 0{,}5$
- współczynnik interpolacji = 0,7
- procent populacji, który zostanie zmutowany po reprodukcji = 5%
- algorytm zostanie zatrzymany jeśli przez 100 iteracji nie będzie poprawy wartości funkcji celu najlepszego osobnika o $\varepsilon = 0$ lub zostanie przekroczony limit 10000 iteracji.

3. Pseudokod algorytmu

Krok 1: Stworzenie populacji μ-osobnikowej, w której każdy osobnik ma wartości z zadanego zakresu oraz ma zadane odchylenie standardowe.
Krok 2: Wylosowanie λ par osobników oraz stworzenie ich potomków.
Krok 3: Mutacja starego pokolenia.
Krok 4: Wybranie μ najlepszych osobników do kolejnej populacji.
Krok 5: Sprawdzenie warunku stopu. Jeśli warunek jest spełniony to zakończenie działania algorytmu. Gdy nie jest spełniony to powrót do kroku 2.

4. Instrukcja do programu

Po uruchomieniu dobieramy parametry potrzebne do poprawnego wykonywania się algorytmu:

- μ – rozmiar populacji,
- λ – ilość generowanych potomków w każdej iteracji,
- rodzaj wyboru – sposób wybierania osobników po reprodukcji,
- min x – minimalna wartość współrzędnej x dopuszczalna w osobnikach,
- min y – minimalna wartość współrzędnej y dopuszczalna w osobnikach,
- rodzaj optymalizacji – cel przeszukiwania: minimum w zakresie, maksimum w zakresie, poszukiwanie konkretnej wartości,
- max x – maksymalna wartość współrzędnej x dopuszczalna w osobnikach,
- max y – maksymalna wartość współrzędnej y dopuszczalna w osobnikach,
- cel optymalizacji – wartość do jakiej ma dążyć algorytm po wybraniu opcji „do wartości" w polu rodzaj optymalizacji,
- σ x – współczynnik mutacji dla współrzędnej x,
- σ y – współczynnik mutacji dla współrzędnej y,
- procent mutacji – procent osobników mutowanych w każdej iteracji algorytmu,
- ε – minimalna zmiana wartości osobnika uważana za zmianę znaczącą,
- max iteracji – maksymalna ilość iteracji algorytmu, jeżeli zostanie przekroczona algorytm zatrzyma działanie,
- max bez poprawy – maksymalna dopuszczalna ilość iteracji bez poprawy najlepszego osobnika o co najmniej ε, jeżeli zostanie przekroczona algorytm zatrzyma działanie,
- wsp. interpolacji – współczynnik interpolacji lepszego osobnika względem gorszego,
- rodzaj algorytmu – wybór algorytmu pomiędzy „μ + λ" a „μ, λ",
- ilość wyświetlanych – ilość najlepszych wyświetlanych osobników w każdej iteracji.

Przyciski:

- Inicjuj – wprowadzanie ustawionych parametrów do klasy implementującej algorytm,
- 1 krok algorytmu – przejście jednej iteracji algorytmu,
- Znajdź rozwiązanie – rozpoczęcie wyszukiwania rozwiązania do czasu osiągnięcia warunków końca,
- Zatrzymaj rozwiązywanie – zatrzymanie wyszukiwania rozwiązania,
- Rozpocznij od nowa – wyczyszczenie aktualnej instancji rozwiązania, powrót do wprowadzania parametrów,
- Wyczyść konsole – czyszczenie zawartości konsoli.

5. Opis struktury programu

Program składa się z klas głównych: Widok, Kontroler i Algorytm, z interfejsu Osobnik i FunkcjaPrzystosowania oraz z klas pomocniczych: FunkcjaRosenbrocka, OsobnikMaxComparator, OsobnikMinComparator, OsobnikDoCeluComparator, Main, Punkt, ParametryPanel, Zakres.

- Algorytm – Klasa implementująca algorytm ewolucyjny. Klasy pomocnicze:
    - FunkcjaRosenbrocka – Jest to klasa zawierająca funkcję dla której poszukujemy rozwiązania. Klasa ta implementuje interfejs FunkcjaPrzystosowania.
    - OsobnikMaxComparator, OsobnikMinComparator, OsobnikDoCeluComparator – Klasy służą do definiowania relacji między osobnikami populacji,
    - Punkt – Klasa implementująca interfejs Osobnik, reprezentuje osobnika populacji.
    - Zakres – Klasa służąca do sprawdzania czy osobnik mieści się w zakresie poszukiwania.

- Widok – Klasa tworząca i zarządzająca graficznym interfejsem użytkownika. Klasa pomocnicza:
    - ParametryPanel – Klasa pomagająca wczytywać parametry pobierane od użytkownika.
- Kontroler – Klasa zarządzająca algorytmem. Pośredniczy między klasą Algorytm a klasą Widok.

6. Wnioski dotyczące osiągniętych rezultatów

Funkcja Rosenbrocka jest dobrze znaną dla nas funkcją z zajęć w dziedzinie optymalizacji. Wiedzieliśmy, iż ma ona jedno minimum globalne. Było dla nas wskazówką, że algorytm ewolucyjny „$\mu + \lambda$” powinien być odpowiedni. Już pierwsze testy ukazały szybkie zaognianie się osobników populacji wokół punku [1;1], który jest jej minimum równym 0. Pozostało więc dostrojenie algorytmu. Dobraliśmy parametry podane w punkcie pierwszym. Uznaliśmy, iż dają one zadowalające efekty czasowe. Generują one około 1100 iteracji (najlepsza zaobserwowana 942, najgorsza zaobserwowana 1420). Ze względu na niewielką ilość populacji wykonuje się to szybko. Ważną informacją jest to, że to zwiększenie ilości potomków, a nie samej populacji, zmniejsza ilość iteracji, w których algorytm znajduje rozwiązanie. Procent mutacji, jak się można było spodziewać, spowalniał rozwiązywanie, gdyż w momencie gdy on rósł, zwiększało to szanse na utracenie najlepszego rozwiązania. Związane jest to z charakterystyką funkcji jak i zakresem, na którym pracowaliśmy. Sigmy, ze względu na bananowy kształt funkcji oraz szybkie zmniejszanie przez mutacje, nie miały większego wpływu na szybkość znajdowania rozwiązania. Współczynnik interpolacji na poziomie 0.7 według nas był najlepszym rozwiązaniem. Dalsze zwiększanie go do jedynki spowalniało schodzenie do doliny, na której dwóch brzegach dość często trafiały się kolejne osobniki. Natomiast przy zmniejszaniu do 0.5 mieliśmy zmniejszanie zysku wyrobionego w ramach jednej krawędzi. Co do epsilonu to dopiero ustawienie go na 0 dawało niemal pewne zejście rozwiązania do maksymalnej dokładności (nawet na wartościach typu double) oraz pokazanie rozwiązania w punkcie [1;1]. Ograniczenia iteracji natomiast były zdroworozsądkowe. Przy 10000 iteracji algorytm musiałby gdzieś zabłądzić. Natomiast 100 iteracji bez poprawy dawało pewność, że faktycznie znaleźliśmy już minimum, a nie że tylko nie udało się nam poprawić ostatniego rezultatu. Dalsze zwiększanie tego ograniczenia nie prowadziło by do znacznego zwiększania pewności optymalnego rozwiązania, a tylko by sztucznie zwiększało ilość wszystkich iteracji algorytmu. Oczywiście trafiają się rozwiązania które nie są w pełni optymalne, aczkolwiek różnica od rozwiązania optymalnego jest rzędu 10^{-20} .