Algorytmy ewolucyjne

Sprawozdanie z projektu nr 2

Kamil Matysek, Mateusz Roszkowski

1. **Wstęp**

Zadaniem w projekcie 2 było znalezienie rozwiązania problemu plecakowego stosując algorytm ewolucyjny.

Do wykonania projektu został stworzony skrypt w matlabie *proj2.m,* gdzie został zaimplementowany algorytm genetyczny. Funkcja celu wykorzystywana przez algorytm genetyczny znajduje się w pliku *fun.m*.

Czynniki, które będą testowane w czasie projektu:

* Rozmiar populacji
* Liczba iteracji
* Rozmiar elity
* Prawdopodobieństwo mutacji
* Stosunek mutacji krzyżowych do mutacji binarnych

Wstępne kryterium stopu: osiągnięcie założonej liczby iteracji

Dane bazowe:

* Rozmiar populacji - 100
* Liczba iteracji – 100
* Rozmiar elity - 33
* Prawdopodobieństwo mutacji – 1/16
* Stosunek mutacji krzyżowych do mutacji binarnych – 1:1

Graphical user interface, application

Description automatically generated

*Wyk 1. Wykres testowy wartości maksymalnej w funkcji numeru iteracji dla wartości bazowych*

Wynik: 1 1 0 0 0 0 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0

Wartość f. celu: 944

Poniższe testy mają na celu sprawdzenie jakie powinny być wartości zmiennych aby algorytm działał jak najlepiej, czyli znajdował najlepsze rozwiązanie funkcji celu i nie tkwił w minimum lokalnym.

1. **Testowanie liczności elity**

*Poniższy test odbył się przy użyciu wartości bazowych i jedynie przy zmianie wartości rozmiaru elity*

Pierwszy test został przeprowadzony w celu dobrania wielkości elity. Testowane wartości (na populację równą 100) to:

* 2 (prawie minimalna dla tej populacji)
* 5
* 10
* 15
* 33 (bardzo duża)

Graphical user interface

Description automatically generated with medium confidence

*Wyk 2. Wartość maksymalna uzyskana w funkcji numeru iteracji dla testów liczności elity*

A picture containing graphical user interface

Description automatically generated

*Wyk 3. Wartość minimalna uzyskana w funkcji numeru iteracji dla testów liczności elity*

Graphical user interface, application

Description automatically generated

*Wyk 4. Wartość średnia uzyskana w funkcji numeru iteracji dla testów liczności elity*

Chart

Description automatically generated

*Wyk 5. Wartość wariancji uzyskana w funkcji numeru iteracji dla testów liczności elity*

Z powyższego testu widać, że zbyt duży rozmiar elity powoduje dłuższe szukanie optimum. Widać to szczególnie na *Wyk5*, w którym test o rozmiarze elity równym 33 nie osiągnął po 100 iteracjach optimum. Mniejsza wartość elity pozwala na znalezienie lepszego optimum ze względu na dużą liczbę mutacji.

Przy kolejnych testach wartość rozmiaru elity będzie wynosić **5**, ponieważ wartość ta najszybciej uzyskała optimum.

1. **Testowanie prawdopodobieństwa mutacji**

Poniższe testy prawdopodobieństwa mutacji obyły się dla wartości:

* 1/2 -średnio połowa wartości wektora powinna się zmienić w czasie mutacji
* 1/4
* 1/8
* 1/16
* 1/32 -średnio jedna wartość wektora powinna się zmienić w czasie mutacji

Graphical user interface

Description automatically generated with low confidence

*Wyk 6. Wartość maksymalna uzyskana w funkcji numeru iteracji dla testów prawdopodobieństwa mutacji*

Graphical user interface

Description automatically generated

*Wyk 7. Wartość minimalna uzyskana w funkcji numeru iteracji dla testów prawdopodobieństwa mutacji*

Graphical user interface

Description automatically generated

*Wyk 8. Wartość średnia uzyskana w funkcji numeru iteracji dla testów prawdopodobieństwa mutacji*

Chart

Description automatically generated

*Wyk 9. Wartość wariancji uzyskana w funkcji numeru iteracji dla testów prawdopodobieństwa mutacji*

Małe prawdopodobieństwo mutacji sprawia, że algorytm może utknąć w optimum lokalnym – widać to na *Wyk 6* na którym jako pierwszy w optimum został algorytm z prawdopodobieństwem mutacji równym 1/32. Duże prawdopodobieństwo sprawia, że algorytm trochę dłużej szuka optimum lokalne.

Kolejne testy zostaną przeprowadzone z prawdopodobieństwem mutacji równym **1/16**, ponieważ ma mniejszą szansę na zatrzymanie się na minimum lokalne aniżeli mniejsze od niego wartości, ale również w kolejnych testach lepsze wyniki zauważy się przy dłuższym szukaniu optimum.

1. **Testowanie proporcji mutacji jednopunktowych do liczby mutacji bitowych**

Poniższe testy proporcji różnych mutacji odbyły się dla wartości (mutacja krzyżowa : mutacja bitowa):

* 4:1
* 3:2
* 1:1
* 2:3
* 1:4

Graphical user interface, application

Description automatically generated

*Wyk 10. Wartość maksymalna uzyskana w funkcji numeru iteracji dla testów proporcji mutacji*

Chart

Description automatically generated with medium confidence

*Wyk 11. Wartość minimalna uzyskana w funkcji numeru iteracji dla testów proporcji mutacji*

Graphical user interface, chart

Description automatically generated with medium confidence *Wyk 12. Wartość średnia uzyskana w funkcji numeru iteracji dla testów proporcji mutacji*

Graphical user interface, chart

Description automatically generated

*Wyk 13. Wartość wariancji uzyskana w funkcji numeru iteracji dla testów proporcji mutacji*

Z testów widać, że mutacje krzyżowe są wydajniejsze w znajdowaniu optimum funkcji aniżeli mutacje binarne.

Zastosowanie mutacji krzyżowych powoduje, że algorytm dłużej szuka poprawnego optima (Wyk 13. -najdłużej dużą wariancję otrzymuje algorytm z proporcją 4:1, a najkrócej z 1:4), ale też najszybciej znajduje najlepsze optimum

1. **Najlepszy wynik:**

W znajdowaniu optimum globalnego posłużono się dużą populacją, aby algorytm mógł jak najszybciej znaleźć rozwiązanie. Wartości czynników zostały podobne do tych z testów, lecz w ramach testów na dużej populacji, niektóre z nich uległy zmianie (np. Rozmiar elity zmienił się z 2 osobników do 5).

Wartości czynników:

* Populacja – 2000
* Rozmiar elity – 5
* Prawdopodobieństwo – 1/16
* Stosunek mutacji krzyżowej do binarnej – 4:1

Zwiększenie liczności populacji znacznie poprawiło wyniki uzyskiwane przez algorytm, niestety spowodowało to z wydłużenie czasu obliczeń.

**4.1 Kryterium stopu: osiągnięcie 100 iteracji**

Graphical user interface, application

Description automatically generated

*Wyk 14. Wartość maksymalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku– test 1.*

Chart

Description automatically generated *Wyk 15. Wartość minimalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku– test 1.*

Graphical user interface, chart

Description automatically generated with medium confidence *Wyk 16. Wartość średnia uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku– test 1.*

Chart

Description automatically generated *Wyk 17. Wartość maksymalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 1.*

Dla powyżej przedstawionych czynników algorytm najszybciej znajdował optimum, które zdawało się być globalne. W teście na 100 iteracjach wartość maksymalna funkcji celu nie zmieniła się od 26 iteracji.

Wynik: 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0

Wartość funkcji celu: 1034

* 1. **Kryterium stopu: wariancja równa mniejsza niż 40**

Aby sprawdzić czy dla danych czynników algorytmu optimum na pewno się nie zmieni ustawiono kryterium stopu zależne od wariancji. Mała wariancja oznacza, że populacja coraz mniej się zmienia i coraz mocniej zbliża się do optimum.

Graphical user interface, application

Description automatically generated

*Wyk 18. Wartość maksymalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 2.*

Chart, line chart

Description automatically generated *Wyk 19. Wartość minimalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku– test 2.*

Graphical user interface

Description automatically generated *Wyk 20. Wartość średnia uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 2.*

Graphical user interface, chart

Description automatically generated *Wyk 21. Wartość wariancji uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 2.*

W 116 iteracji wariancja osiąga wartość 36.97, co spełnia kryterium stopu. Wartość funkcji celu nie zmieniła się i nadal wynosi 1034.

Wynik: 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0

* 1. **Zastosowanie stosunki mutacji 3:2**

Aby sprawdzić czy mała zmiana mutacji krzyżowych nie polepszy algorytmu sprwadzono algorytm dla stosunku mutacji krzyżowych do bitowych równym 3:2.

**Kryterium stopu: Wariancja mniejsza niż 50**

Graphical user interface, application

Description automatically generated

*Wyk 22. Wartość maksymalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 3.*

Graphical user interface, application, table, Excel

Description automatically generated *Wyk 23. Wartość minimalna uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 3.*

Graphical user interface, application

Description automatically generated *Wyk 24. Wartość średnia uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 3.*

Graphical user interface, application, table

Description automatically generated *Wyk 25. Wartość wariancji uzyskana w funkcji numeru iteracji dla najlepszego otrzymanego wyniku – test 3.*

Wariancja (*Wyk. 25)* w tym teście nie osiągnęła wariancji niższej niż 50 (minimalna wartość była równa 80.69), więc przerwano eksperyment przy 1000 iteracji.

Wyniki osiągnięte przez algorytm były takie same jak w poprzednich dwóch testach.

1. **Wykorzystanie funkcji ga() do znalezienia optymalnego rozwiązania.**

W celu sprawdzenia czy nasz algorytm działa dobrze, do rozwiązania problemu plecakowego wykorzystaliśmy również funkcję ga() (skrypt zawarty w pliku *gaa.m*). Najlepszy uzyskany wynik przy użyciu funkcji ga() z domyślnymi parametrami, był tożsamy z tym uzyskany naszym autorskim algorytmem.

Postanowiliśmy również porównać skuteczność obu algorytmów na podstawie 25 niezależnych uruchomień, w tym przypadku aby skrócić czas obliczeń parametry naszego algorytmu ustawiliśmy następująco:

rozmiar\_populacji=1000;

rozmiar\_elity=1;

liczba\_iteracji=50;

praw\_mutacji=1/16;

liczba\_pot\_krzyz=400;

liczba\_pot\_mut=700;

Funkcja ga() dała średni wynik 1019.3, podczas gdy nasz algorytm 1032.7, uzyskał on wynik optymalny (1034) w 22 z 25 prób, za każdym razem robił to w mniej niż 50 iteracji. Jednak dużą zaletą jest szybkość z jaką uzyskujemy wynik przy użyciu funkcji ga(), mimo że potrzebuje ona często więcej iteracji (średnio 66) to przez to, że populacja jest znacznie mniejsza niż w naszym algorytmie, obliczenia wykonują się znacznie szybciej.

1. **Rozwiązanie problemu:**

Przedmioty, które należy spakować: [1,2,3,6,7,10,11,14,15,16,24,25,26,27,29]

[1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0]

Waga optymalnego plecaka = 5.1

Dopuszczalna waga plecaka = 5.13

Wartość optymalnego plecaka = 1034