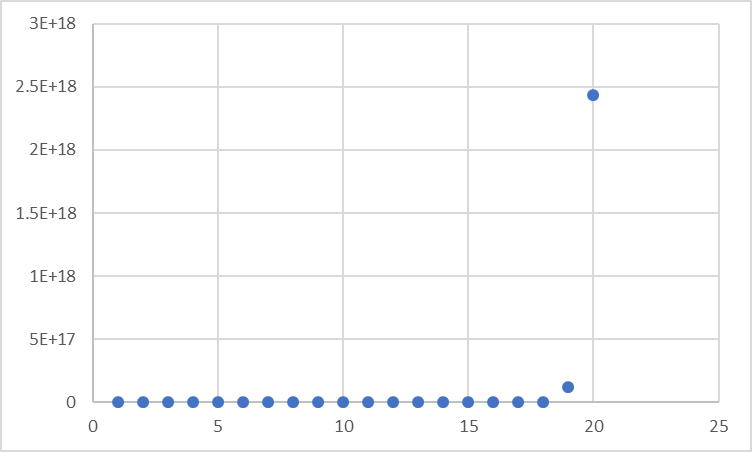
## Wstęp

Problem kwadratowego przydziału zasobów QAP jest jednym z najważniejszych w przemyśle i obecnym w literaturze naukowej. Problem sprowadza się do wskazania lokalizacji dla N fabryk, tak, aby zminimalizować koszt transportu pomiędzy nimi, co stanowi rozszerzenie klasycznego problemu komiwojażera i jest problemem NP-trudnym.

Jednym z rozwiązań problemów tego problemu może być przegląd zupełny, czyli sprawdzenie wszystkich możliwych rozwiązań. Da to rozwiązanie optymalne, jednak w bardzo nieefektywny sposób. W przypadku problemu QAP zależność między parametrem N a liczbą rozwiązań to N!.



Wykres 1 - Wartość silni dla zadanego x

Jak łatwo zauważyć, na wykresie 1 wartość silni rośnie bardzo szybko, i dla większych wartości N przegląd zupełny jest bardzo nieefektywny i w praktyce niewykonalny.

W związku z tym, kolejnym możliwym rozwiązaniem problemu może być np. algorytm genetyczny. Algorytm ten przypomina zjawisko ewolucji biologicznej, a sam twórca algorytmu właśnie z biologii czerpał inspirację do stworzenia algorytmu.

## Opis problemu QAP

W przypadku problemu QAP musimy wskazać N lokalizacji (1, 2, … n) dla fabryk (A, B, …) tak, aby koszty były najmniejsze. Liczba fabryk i lokalizacji jest taka sama. Każda fabryka ma zdefiniowany wymagany przepływ do innej fabryki a także koszt transportu między danymi lokalizacjami. Chcemy uzyskać informację, jak przypisać fabrykę do lokalizacji w taki sposób, aby sumaryczny koszt był największy.

**Wejście**: liczba fabryk (N), macierz odległości (D), macierz przepływu (F)

**Wyjście**: Numery przypisanej lokalizacji odpowiednio dla fabryk A, B …

## Opis algorytmu genetycznego

W algorytmie genetycznym podstawowym pojęciem jest populacja. Jest to zbiór osobników z pewnym przypisanym genotypem (zbiorem informacji), który stanowi podstawę do utworzenia fenotypu (zbioru cech). Następnie algorytm dokonuje losowania pewnej populacji początkowej, która poddawana jest selekcji (ocenie). Osobniki z największą oceną biorą udział w procesie reprodukcji, poprzez ich skrzyżowanie i przeprowadzenie mutacji. Skrzyżowane osobniki są kolejnym pokoleniem. Aby utrzymać stałość populacji najsłabsze osobniki są z niej usuwane.

Zatem, algorytm genetyczny możemy zapisać w następujący sposób:

* Dokonaj losowania populacji początkowej (1)
* Poddaj wybraną populację selekcji (2)
* Wybierz najlepsze osobniki które wezmą udział w reprodukcji (3)
* Genotypy najlepszych osobników skrzyżuj oraz poddaj mutacji (4)
* Poddaj nowe osobniki ocenie. Jeżeli osiągnięto warunek końcowy zakończ, jeżeli nie, wróć do punktu 2 (5).

## Realizacja algorytmu dla problemu QAP

Niech:

* *N –* liczba miast/lokalizacji
* *pop\_size* – wielkość populacji
* *gen* – ilość generacji
* *Px* – prawdopodobieństwo krzyżowania
* *Py* – prawdopodobieństwo mutacji
* *Tour* – wielkość turnieju (istotna dla selekcji)

1. Na początku dokonujemy wybrania losowej populacji początkowej. W tym celu wybieramy losowe *pop\_size* ze zbioru wszystkich możliwych rozwiązań problemu.
2. Następnie dokonujemy selekcji każdego osobnika, wg funkcji oceny:

Niech S oznacza wektor cech osobnika (tj. kolejno przyporządkowane lokacje). Wtedy, funkcja kosztu wyraża się jako:

1. Następnie wybieramy osobników, których poddamy krzyżowaniu. Będziemy to robić w następujący sposób: wybieramy z całej populacji losowo *Tour* osobników. Następnie, z tego turnieju wybieramy tego osobnika, który miał największą adopcję (tj. najmniejszy koszt). Powtarzamy ten krok do momentu, aż nasza nowa populacja osiągnie *pop\_size* osobników.
2. Następnie poddajemy osobniki krzyżowaniu i mutacji, wg schematu 1.

Pula osobników

Osobnik 1

Osobnik 2

Krzyżuj?

Dziecko 2

Pula wyjściowa

Mutuj?

Dziecko 1

Tak

Osobnik 2

Osobnik 1

Nie

Zmutowany

Niezmutowany

Tak

Nie

Schemat 1 - sposób krzyżowania i mutacji

Dla każdego krzyżowania wybieramy 2 osobników bez zwracania. Decydujemy, czy ich krzyżować (zgodnie z prawdopodobieństwem *Px*). Jeżeli tak, dokonujemy krzyżowania. Losujemy liczbę x z zakresu 1..N. Z pierwszego osobnika „bierzemy” x pierwszych wartości, pozostałe wartości uzupełniamy biorąc kolejne wartości z drugiego osobnika, które nie występują w pierwszym. Z drugim osobnikiem postępujemy tak samo. Dzięki temu uzyskujemy 2 nowych osobników. Jeżeli natomiast ich nie krzyżujemy pozostawiamy je bez zmian.

Następnie, po wykonaniu krzyżowania dokonujemy mutacji. Dla każdego genu osobnika decydujemy zgodnie z prawdopodobieństwem *Py* czy dokonujemy mutacji. Jeżeli tak – zamieniamy gen miejscem z losowym innym genem.

1. Poddajemy osobniki ocenie. Jeżeli osiągnęliśmy zadane *gen* (czyli nr\_pokolenia = *gen*), przerywamy algorytm. Jeżeli nie, wracamy do punktu 2.

# Implementacja algorytmu

Algorytm został zaimplementowany w języku Kotlin. Aby go uruchomić, należy w klasie Main.kt ustawić odpowiednie parametry. Opis parametrów w tabeli 1.

Aplikacja wykonuje zadaną liczbę przebiegów dla każdego z plików, wyświetlając uśrednione wyniki na ekranie. Na wykresie widoczny jest również uśredniony błąd średniokwadratowy dla danego pokolenia

|  |  |  |
| --- | --- | --- |
| Nazwa parametru | Opis | Domyślna wartość |
| outputPrefix | Folder, do którego będą zapisywać się wyniki prób | output/ |
| inputPrefix | Folder, z którego będą brane dane do uruchomienia algorytmu | input/ |
| inputFilenames | Pliki z folderu, na których zostanie uruchomiony algorytm | had12.dat … |
| populationSelector | Sposób wyboru populacji. Instancja selektora musi implementować interfejs *IPopulationSelector*. Aplikacja ma zaimplementowane: *RouletteSelector* oraz *TournamentSelector* | TournamentSelector |
| geneticOperations | Instancja interfejsu *IGeneticOperations*, implementująca metody *crossover* oraz *mutate*. | StandardGeneticOperations |
| numberOfTries | Liczba wywołań algorytmu dla pojedynczego pliku. | 10 |
| startPopulationCount | Liczba osobników w startowej populacji | 100 |
| numberOfGenerations | Liczba generacji do zakończenia algorytmu | 100 |
| tournamentSampleCount | Liczba osobników biorących udział w turnieju (dotyczy tylko sytuacji, gdy wybrany jest TournamentSelector) | 10 |
| crossoverPropability | Prawdopodobieństwo krzyżowania 2 osobników | 0.8f |
| mutationPropability | Prawdopodobieństwo mutacji jednego genu osobnika | 0.03f |

Tabela 1- parametry aplikacji

W programie użyto zewnętrzne biblioteki, tj:

*"com.github.dpaukov:combinatoricslib3:3.2.0" – obliczenia do kombinatoryki*

*"org.knowm.xchart:xchart:3.5.0" - wykresy*

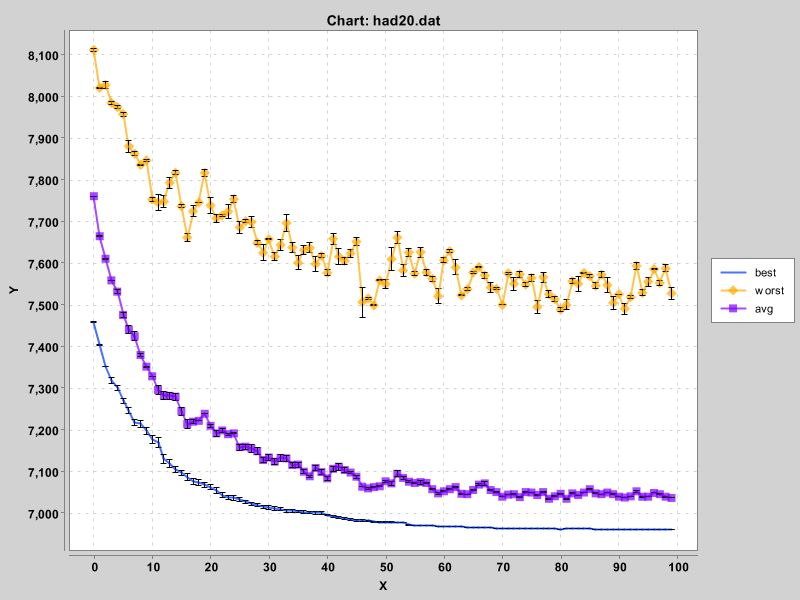
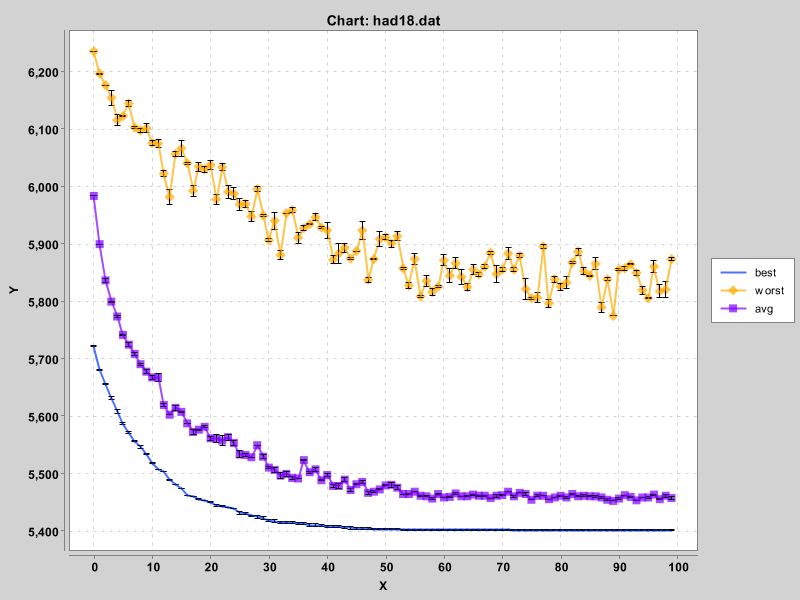
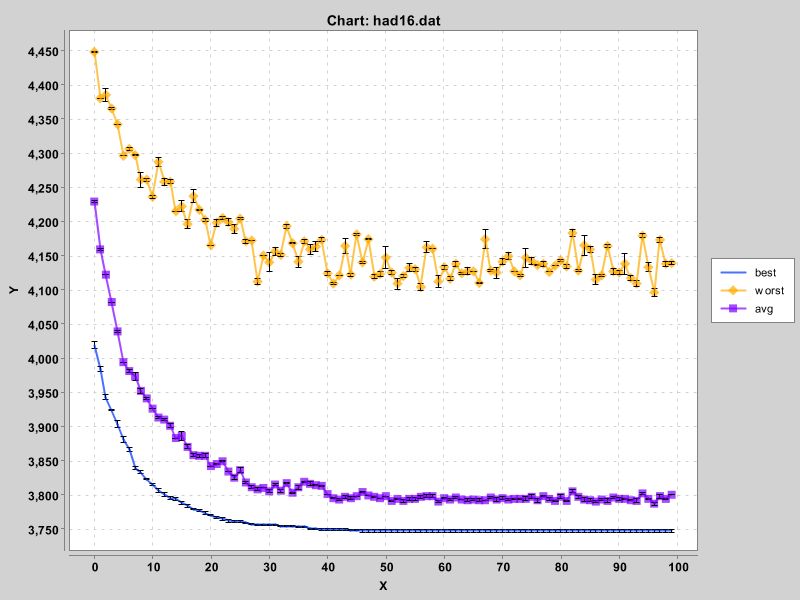
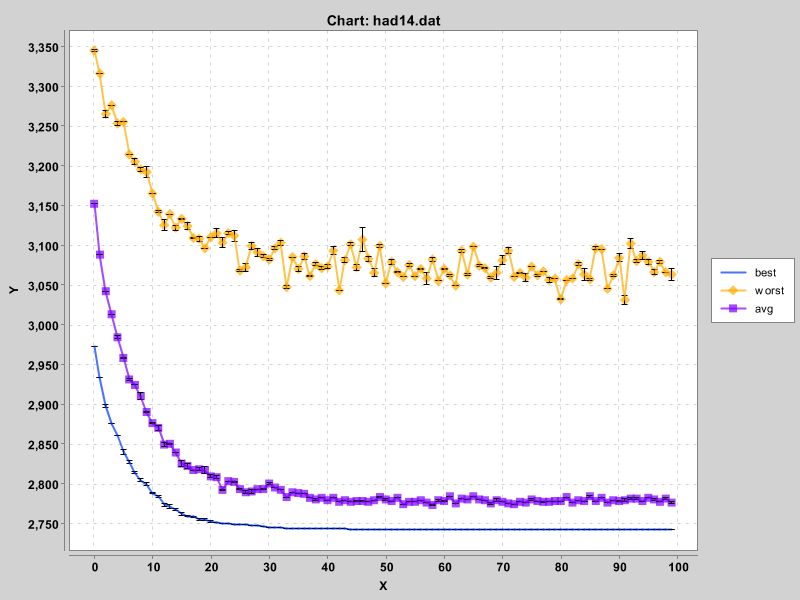
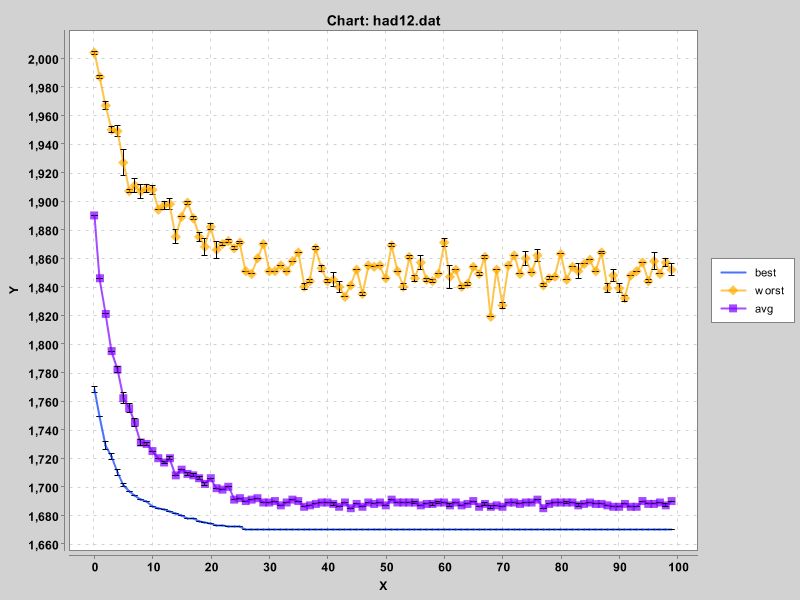
# Zbadanie działania algorytmu dla przykładowych plików.

Zostało zbadane działanie algorytmu dla 5 instancji dla domyślnych parametrów. Wyniki z 10 prób uśredniono uzyskując następujące dane. Wartość „optymalny” pochodzi ze strony <http://anjos.mgi.polymtl.ca/qaplib/inst.html#HRW> i jest optymalnym rozwiązaniem danego problemu.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Najlepszy | | Średni | | Najgorszy | |
| Plik | Optymalny | Wynik | Błąd | Wynik | Błąd | Wynik | Błąd |
| had12 | 1652 | 1670 | 0 | 1690 | 0 | 1852 | 4 |
| had14 | 2724 | 2742 | 0 | 2776 | 1 | 3063 | 8 |
| had16 | 3720 | 3747 | -2 | 3800 | 0 | 4139 | 2 |
| had18 | 5358 | 5401 | -1 | 5457 | -2 | 5873 | 3 |
| had20 | 6922 | 6960 | 0 | 7036 | 0 | 7527 | 15 |

**W pozostałej części wszystkie pominięte (niewymienione) parametry algorytmu są domyślne i zgodne z Tabela 1.**

Wykresy dla kolejnych instancji wyglądają następująco:



Tym samym zapoznano się z działaniem algorytmu genetycznego.

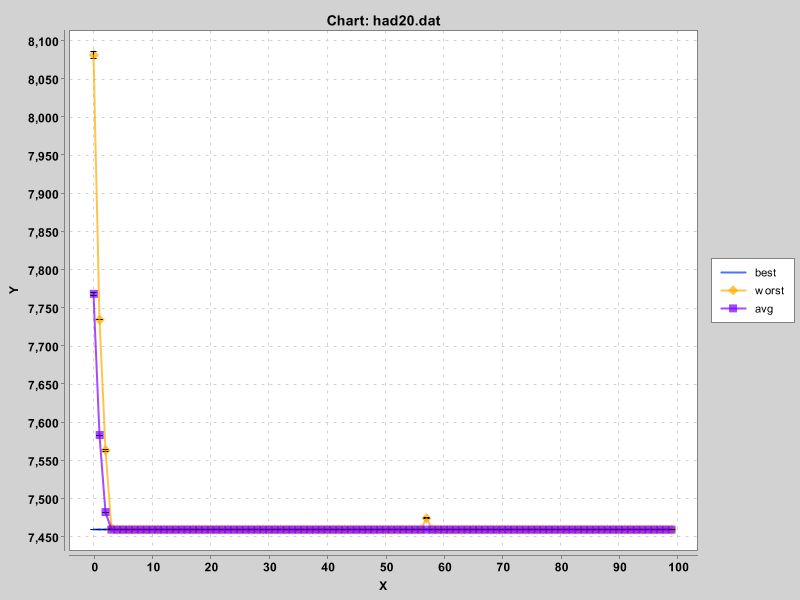
Następnie, dla wybranej instancji (tj. had20) zbadano wpływ poszczególnych parametrów na wynik.

### Prawdopodobieństwo krzyżowania Px, prawdopodobieństwo mutacji Pm

Działanie algorytmu zbadano dla następujących kombinacji [wyniki z tabeli dotyczą tylko ostatniego pomiaru]:

1. Px = 0 && Pm = 0 (mutacja i krzyżowanie nie działają)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0 | 0 | 7459 | 7459 | 7459 |



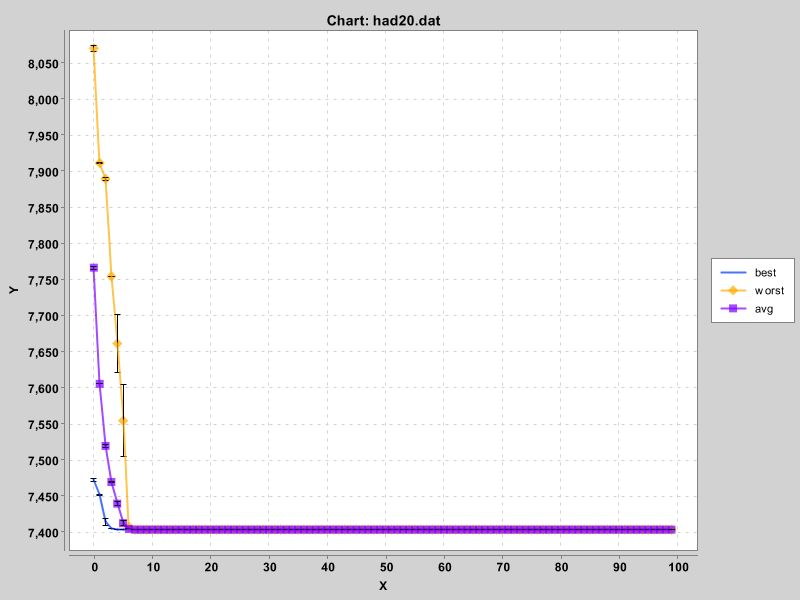
Jak widać algorytm bardzo szybko znalazł swoje optymalne minimum, jednak z powodu wyłączonej mutacji i krzyżowania wynik okazał się być bardzo niski i daleki od optymalnego.

Program w zasadzie ciągle operował na tych samych danych, przez co nie mógł znaleźć lepszego rozwiązania**. Brak różnorodności populacji z powodu braku mutacji jest widoczny.**

1. Px = 0.2 && Pm = 0 (Mutacja wyłączona, krzyżowanie na bardzo niskim poziomie)

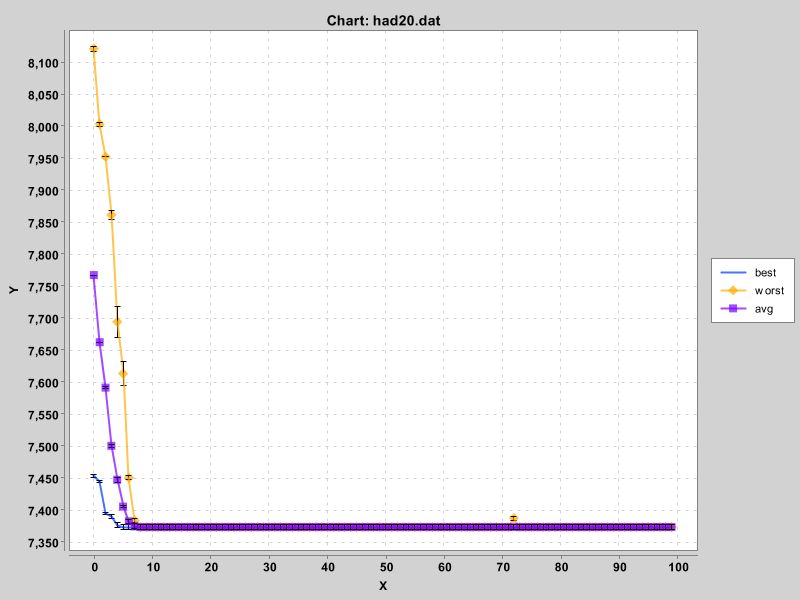
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0.2 | 0 | 7403 | 7403 | 7403 |

Algorytm poradził sobie nieco lepiej, jednak dalej miał problem z wychodzeniem z minimów lokalnych. Ponieważ do krzyżowania dochodziło bardzo rzadko, program operował w zasadzie ciągle na bardzo podobnej populacji. Wynik był daleki od optymalnego.



1. Px = 0.8 && Pm=0

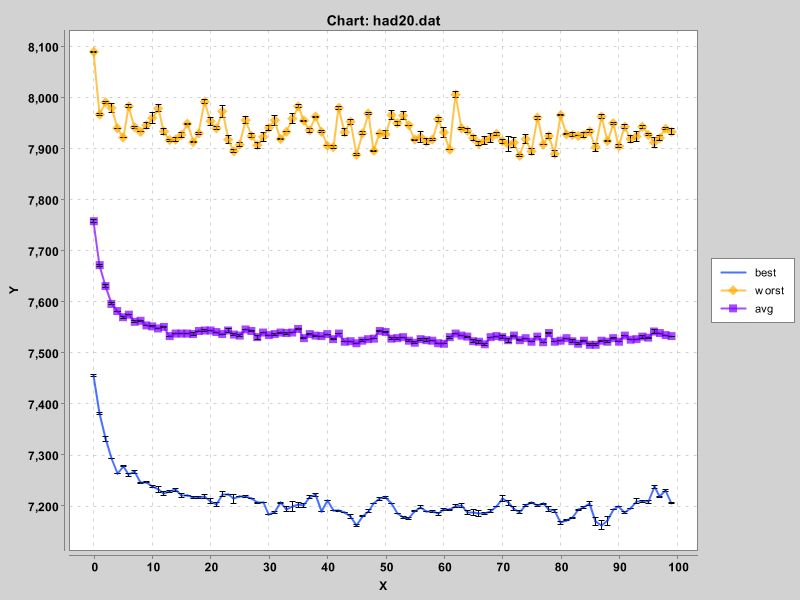
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0.8 | 0 | 7373 | 7373 | 7373 |



W przypadku dużego prawdopodobieństwa krzyżowania oraz zerowego prawdopodobieństwa mutacji efekt był taki, że była większa różnorodność, dzięki czemu algorytm był w stanie zbliżyć się do dosyć dobrego wyniku, jednak z powodu braku mutacji często utykał minimach lokalnych.

1. Px = 0 && Pm = 0.2

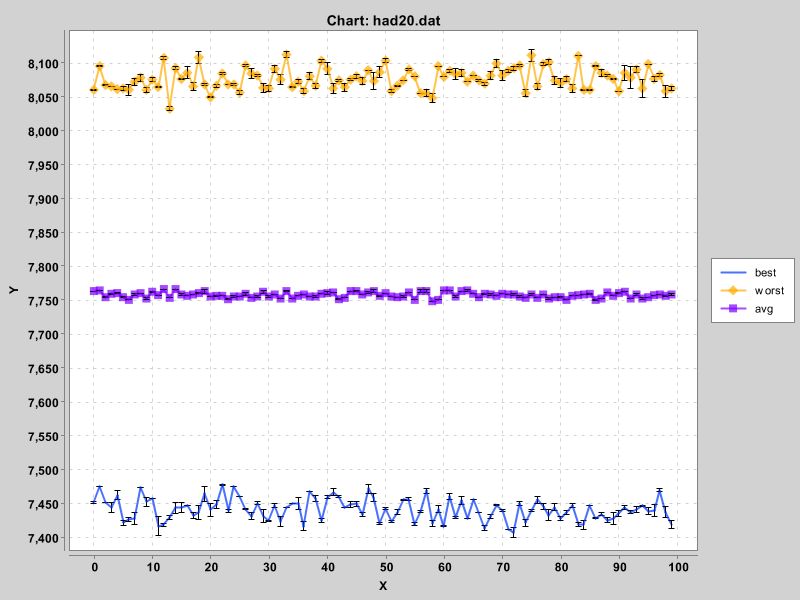
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0 | 0.2 | 7205 | 7532 | 7932 |



Bez krzyżowania, z włączoną mutacją na poziomie 0.2 poradził sobie nadzwyczaj dobrze. Fakt mutowania oraz wybierania z populacji najlepszych osobników wystarczył, aby osiągnąć wynik lepszy, niż przy samym krzyżowaniu. Różnorodność z powodu mutacji jest widoczna.

1. Px = 0 && Pm = 0.8

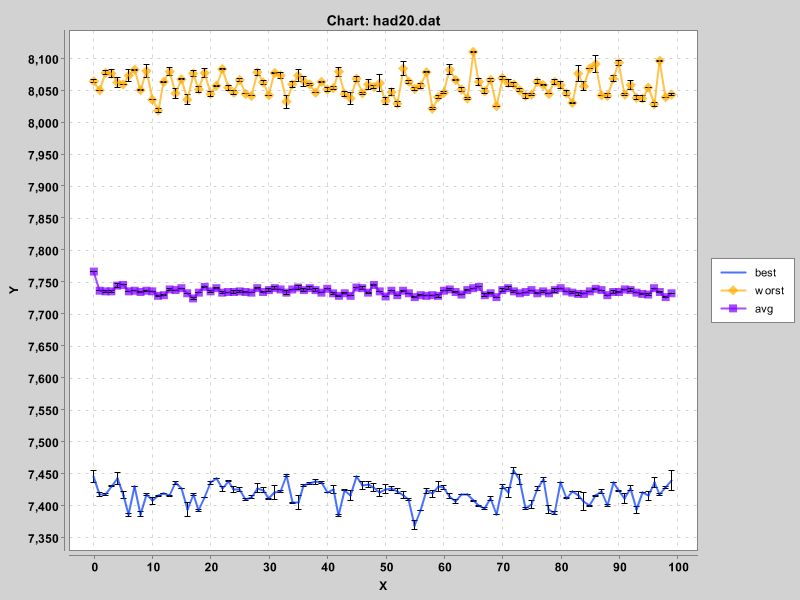
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0 | 0.8 | 7419 | 7758 | 8062 |



Tym razem zbyt wysoka mutacja spowodowała, że w zasadzie brak jest widocznej poprawy wyników. Wyniki utrzymują się w zasadzie na jednym poziomie (czyli są losowe).

1. Px = 0.5 && Pm = 0.5

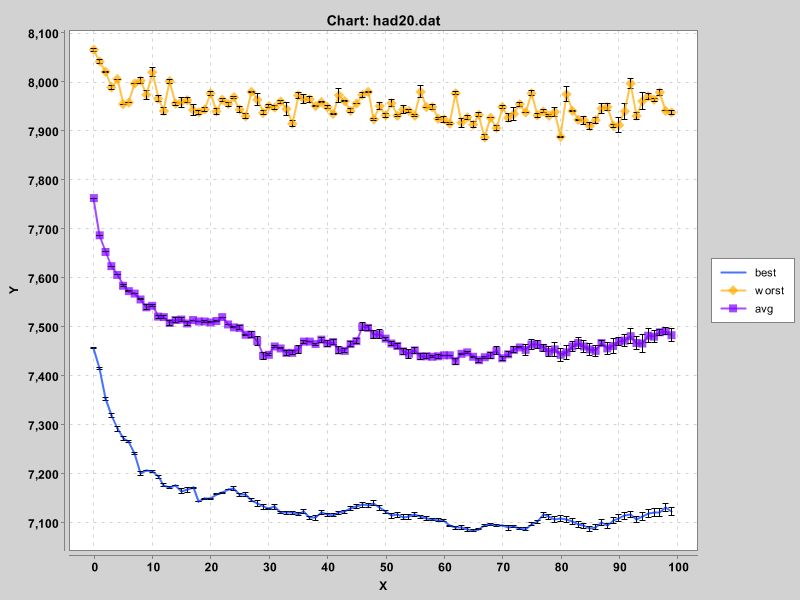
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0.5 | 0.5 | 7439 | 7732 | 8043 |



Podobnie jak w powyższym przypadku wyniki są losowe.

1. Px = 0.8 && Pm = 0.1

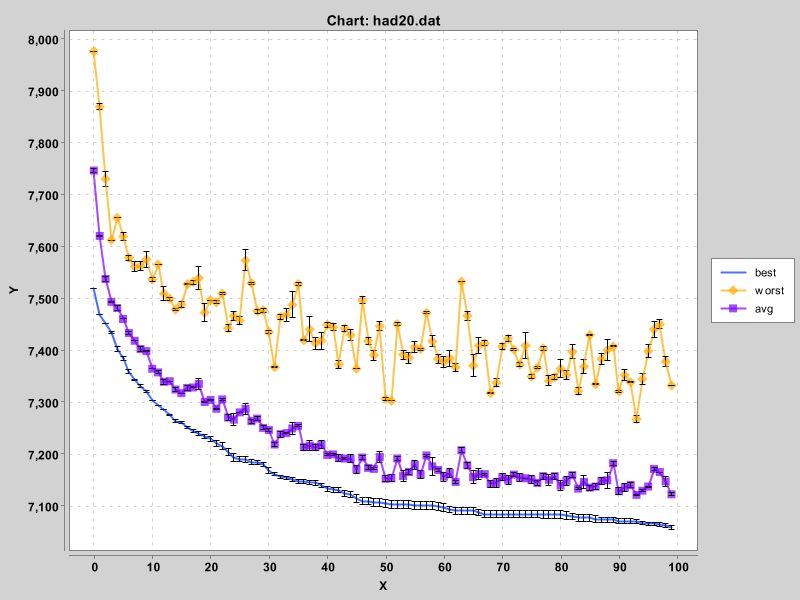
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Najlepszy | Średni | Najgorszy |
| Px | Pm | Wynik | Wynik | Wynik |
| 0.8 | 0.1 | 7122 | 7482 | 7937 |



Jak widać, algorytm znajduje kolejne minima, jednak z powodu zbyt dużej mutacji często gubi minimum lokalne.

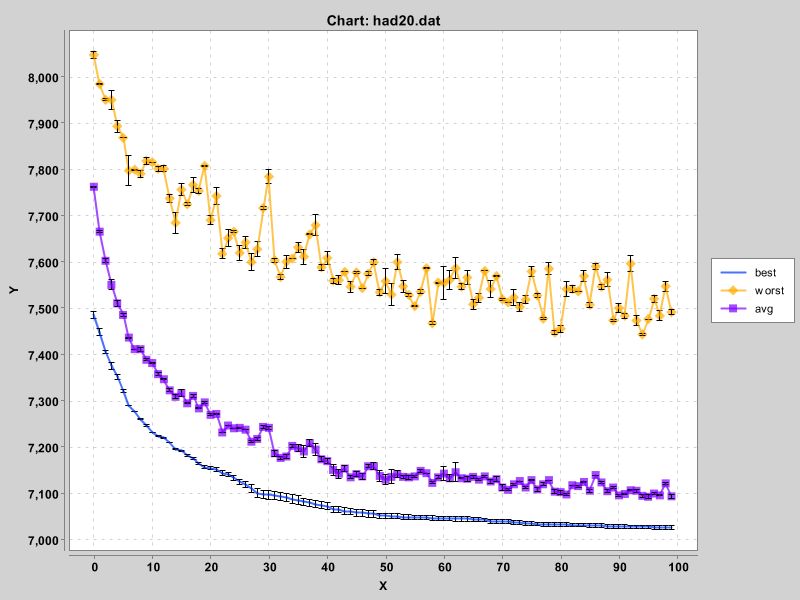
## Zbadanie pop\_size oraz gen

1. pop\_size = 20



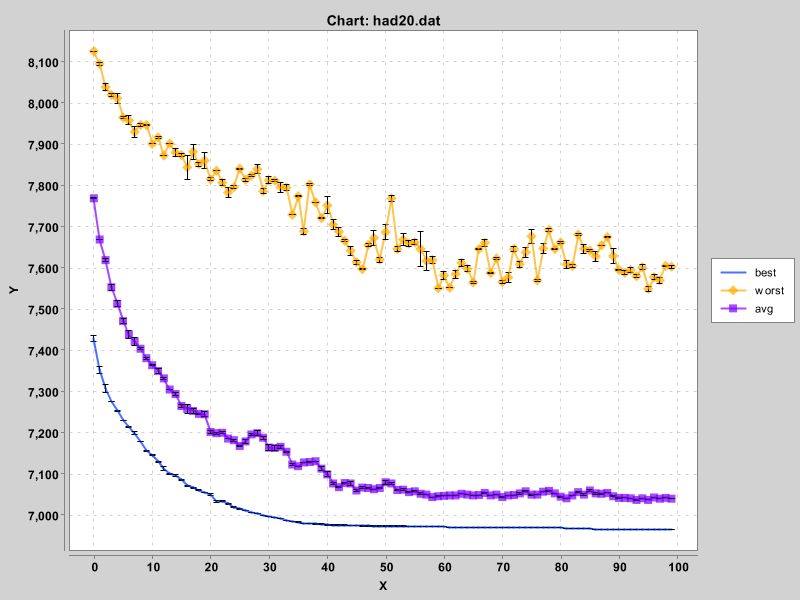
Mała liczba populacji początkowej powoduje, że jest mała różnorodność, widać, że z każdym krokiem jest coraz lepsze rozwiązanie, przez co algorytm potrzebuje większą liczbę przebiegów, aby znaleźć optymalny wynik.

1. Pop\_size = 50



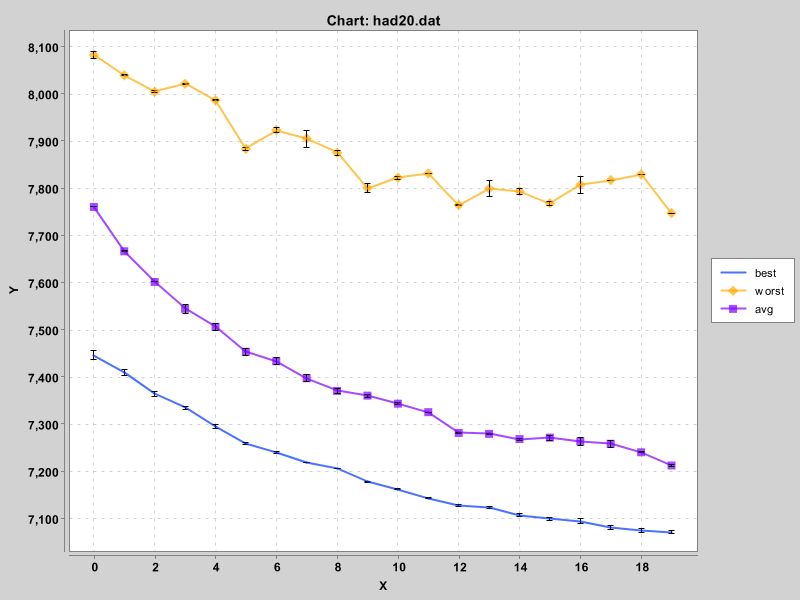
Większa liczba populacji powoduje, że algorytm szybciej znajduje lepsze populacje, jednak w późniejszych pokoleniach nie idzie to tak wydajnie.

1. Pop\_size = 250



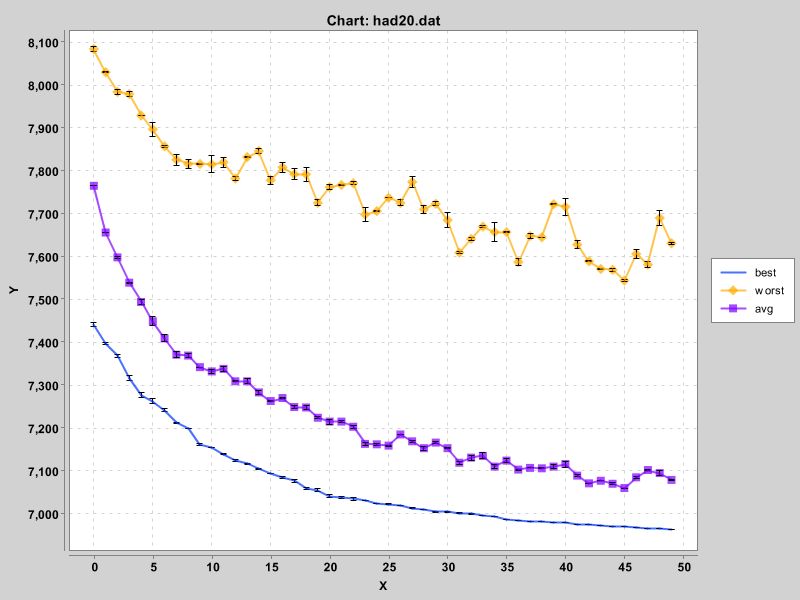
Algorytm ma tendencję malejącą, znajduje lepsze rozwiązania. Duża liczba osobników w populacji początkowej powoduje, że rozwiązanie znajdywane jest bardzo szybko, a w późniejszych generacjach algorytm ma z tym problem.

1. Gen = 20



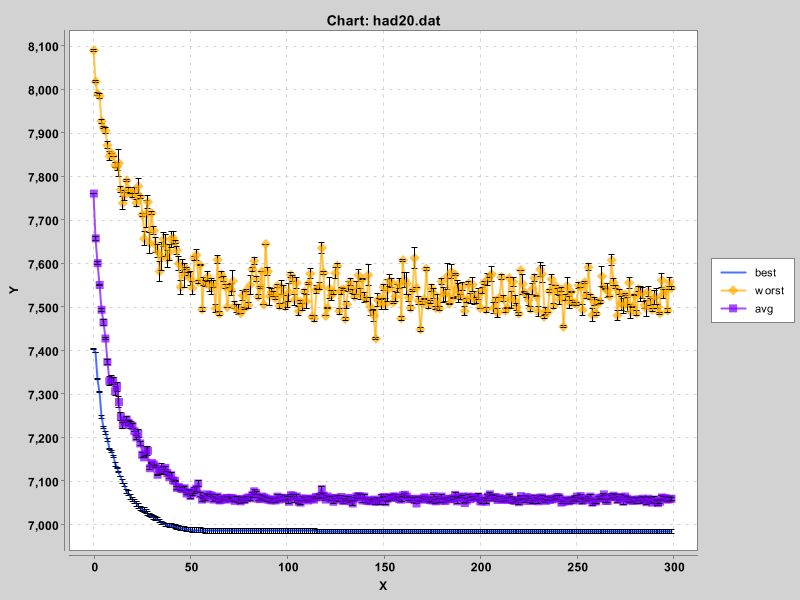
Dla małej liczby pokoleń algorytm ma ciągle tendencję malejącą. Mała liczba pokoleń powoduje, że algorytm nie zdąży znaleźć optymalnego rozwiązania.

1. Gen = 50



Algorytm w późniejszych pokoleniach ma coraz spadek, ponieważ algorytm zbliża się do swojego minimum.

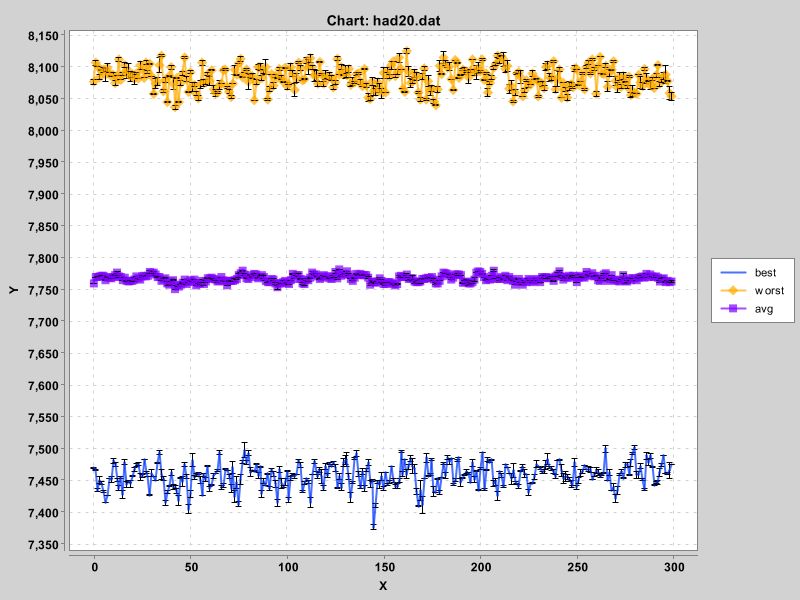
1. Gen = 300



W około 80 pokoleniu algorytm znalazł rozwiązanie bliskie swojemu minimum. W późniejszych pokoleniach algorytm miał problem ze znalezieniem lepszego rozwiązania.

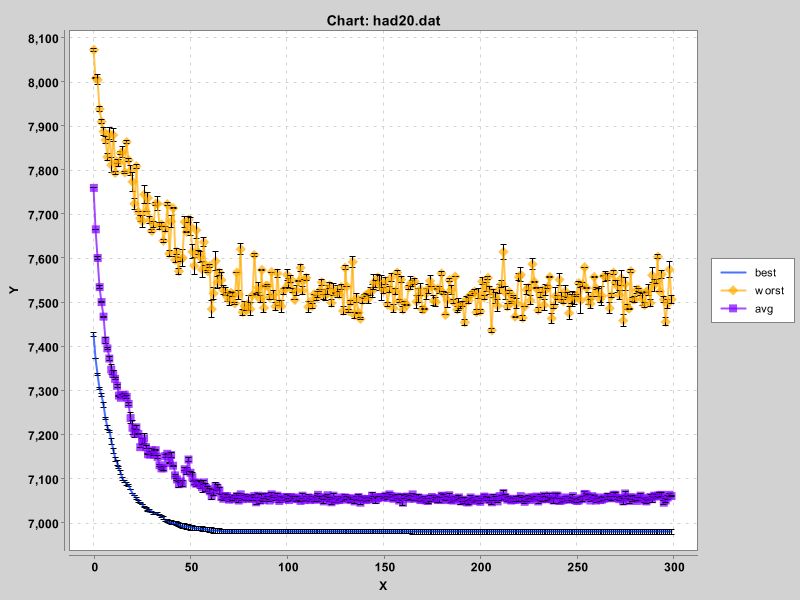
## Wpływ sposobu selekcji na wynik działania algorytmu

1. Selekcja za pomocą ruletki:



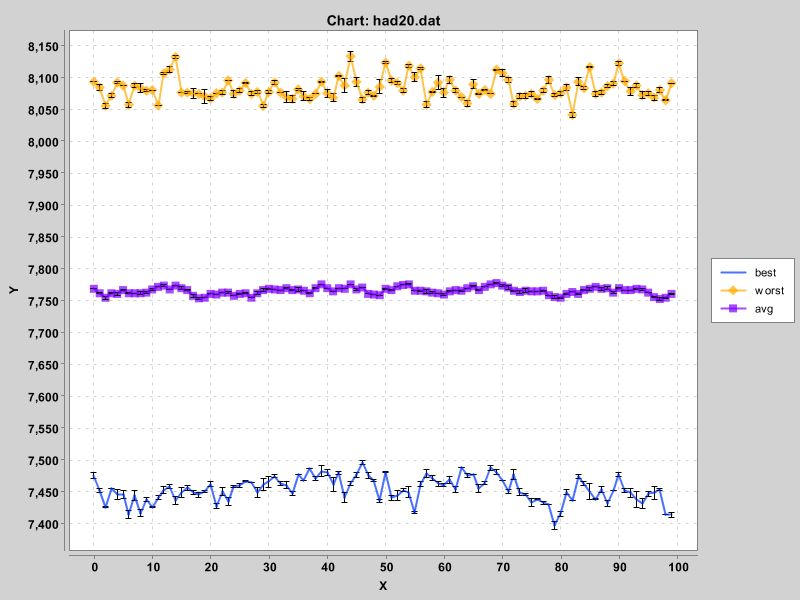
Wybór za pomocą algorytmu ruletki nie daje rady, ponieważ do kolejnych pokoleń dostają się wszystkie osobniki, w tym również słabe. Algorytm w zasadzie zwraca losowe rozwiązania.

1. Selekcja za pomocą turnieju:



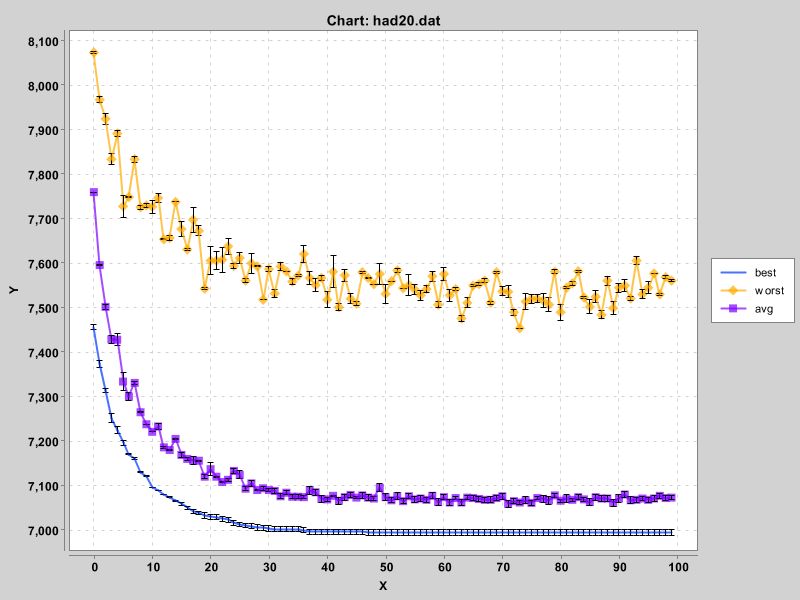
Ponieważ selekcja była dokonywana za pomocą turnieju, do puli osobników przechodziły tylko najlepsze osobniki, dlatego też w kolejnych były znajdywane lepsze rozwiązania.

1. Selekcja za pomocą turnieju, przy rozmiarze turnieju = 1



Występuje bardzo podobna sytuacja jak przy selekcji za pomocą ruletki. Algorytm nie jest w stanie wytypować najlepszych osobników (ponieważ jest jeden losowy).

1. Selekcja za pomocą turnieju, przy rozmiarze turnieju = 50



Do turnieju jest wybierane zbyt wiele osobników, przez co słabsze osobniki nie mają szansy ewoluować. Algorytm szybko znajduje lokalne minimum.

## Podsumowanie

Algorytm genetyczny stanowi bardzo ciekawą alternatywę dla algorytmów nieheurystycznych. Nie znajduje on optymalnego rozwiązania, jednak często są to rozwiązania bardzo dobre i wystarczające. Dlatego też może być używany do rozwiązywania problemów NP-trudnych.

Rozwiązanie, które otrzymamy dzięki zastosowaniu tego algorytmu bardzo zależy od wybranych parametrów, z których za główne możemy wymienić:

* pop\_size – liczba startowej populacji. Jeżeli jest zbyt wielka to algorytm szybko znajdzie rozwiązanie, jednak dla późniejszych pokoleń przestaje on znajdować lepsze rozwiązania (szybko osiąga minimum lokalne).
* gen – liczba generacji. Jeżeli jest zbyt wielka algorytm dla późniejszych pokoleń przestanie znajdować lepsze wyniki. Zbyt mała powoduje natomiast, że algorytm nie zdąży znaleźć swojego minimum.
* tour – liczba osobników w populacji. Zbyt mała powoduj, że do populacji wybierane są słabe osobniki, przez co algorytm działa w zasadzie tak samo jak dla ruletki. Dla zbyt dużej słabsze osobniki nie mają szans się krzyżować.
* Px – krzyżowanie odpowiada za wybór z najlepszych cech z osobników. Zbyt małe powoduje, że osobniki nie są krzyżowane, przez co algorytm utyka w swoich minimach lokalnych.
* Pm – brak mutacji powoduje, że algorytm wpada bardzo szybko w optimum lokalne. Zbyt wielka powoduje jednak, że algorytm nie może się utrzymać w minimum, przez co znajduje często gorsze rozwiązania niż w poprzedniej populacji.

Pula osobników

Osobnik 1

Osobnik 2

Krzyżuj?

Dziecko 2

Pula wyjściowa

Mutuj?

Dziecko 1

Tak

Osobnik 2

Osobnik 1

Nie

Zmutowany

Niezmutowany

Tak

Nie