Projektowanie Efektywnych Algorytmów

Projekt 4

Katsiaryna Kolyshko 276708

Dr. inż. Marcin Łopuszyński

Wstęp teoretyczny

Algorytm genetyczny (AG) to metoda optymalizacji inspirowana procesami ewolucji biologicznej. W kontekście problemów optymalizacji kombinatorycznej, takich jak Problem Komiwojażera (TSP), algorytmy genetyczne oferują skuteczną metodę znajdowania wysokiej jakości rozwiązań w rozsądnym czasie obliczeniowym.

Podstawy i Zastosowania

Algorytmy genetyczne symulują procesy ewolucyjne do rozwiązywania złożonych problemów optymalizacyjnych. Utrzymują populację potencjalnych rozwiązań (chromosomów), które ewoluują przez kolejne pokolenia poprzez selekcję, krzyżowanie i mutację. Jest to szczególnie wartościowe dla problemów NP-trudnych, gdzie znalezienie dokładnego rozwiązania staje się obliczeniowo niewykonalne wraz ze wzrostem rozmiaru problemu.

Metody Selekcji

Selekcja turniejowa: Ta metoda losowo wybiera k osobników i wybiera najlepszego spośród nich. Zapewnia spójną presję selekcyjną i jest mniej obliczeniowo intensywna. Presję selekcyjną można regulować, modyfikując rozmiar turnieju.

Selekcja metodą koła ruletki: Znana również jako selekcja proporcjonalna do przystosowania, przypisuje prawdopodobieństwo wyboru proporcjonalne do wartości przystosowania osobnika. Metoda ta może prowadzić do przedwczesnej zbieżności przy znaczących różnicach w wartościach przystosowania.

Operatory Krzyżowania

Order Crossover (OX): Operator OX zachowuje względne pozycje elementów z jednego rodzica, utrzymując kolejność pozostałych elementów z drugiego rodzica. Jest szczególnie skuteczny dla problemów permutacyjnych, ponieważ zachowuje poprawność tras.

Analiza Złożoności Teoretycznej

Złożoność czasowa:
- Całkowita złożoność dla n pokoleń z populacją o rozmiarze p i długości chromosomu l wynosi $O(n \times p \times l)$
- Ocena przystosowania: $O(l)$ na osobnika
- Selekcja turniejowa: $O(k)$ dla turnieju o rozmiarze k
- Krzyżowanie OX: $O(l)$
- Mutacja: $O(1)$ dla swap, $O(k)$ dla invert

Złożoność pamięciowa:
- Przechowywanie populacji: $O(p \times l)$
- Pamięć tymczasowa na operacje genetyczne: $O(l)$
- Dodatkowe struktury danych: $O(p)$

Operatory Mutacji
Mutacja swap: Losowo zamienia dwie pozycje w chromosomie. Zapewnia lokalną eksplorację przestrzeni rozwiązań przy minimalnym koszcie obliczeniowym.

Mutacja invert: Odwraca kolejność elementów między dwoma losowymi pozycjami. Umożliwia większe zmiany w strukturze chromosomu, pomagając w ucieczce z lokalnych optimów.

Właściwości zbieżności:
Mimo że algorytmy genetyczne nie gwarantują znalezienia optimum globalnego, często znajdują wysokiej jakości rozwiązania w efektywny sposób. Sukces algorytmu zależy od:
- Utrzymania różnorodności populacji
- Zrównoważenia eksploracji i eksploatacji
- Odpowiedniego doboru parametrów
- Jakości operatorów genetycznych

Kombinacja selekcji turniejowej z krzyżowaniem OX oraz mutacjami swap i invert zapewnia solidne ramy dla rozwiązywania instancji TSP. Ta implementacja wykorzystuje mocne strony każdego operatora, zachowując rozsądne wymagania obliczeniowe.
Zasada Działania Algorytmu Genetycznego

Inicjalizacja:
Algorytm rozpoczyna się od wygenerowania losowej populacji początkowej. Każdy osobnik reprezentuje potencjalne rozwiązanie problemu, zakodowane w formie chromosomu. W przypadku TSP chromosom to permutacja miast określająca kolejność ich odwiedzania.

Ocena Przystosowania:
Każdemu osobnikowi przypisywana jest wartość funkcji przystosowania (fitness). Dla TSP jest to zwykle całkowita długość trasy - im krótsza trasa, tym lepsze przystosowanie osobnika.

Ewolucja Populacji:

Selekcja:
- Metoda turniejowa wybiera k losowych osobników
- Najlepszy z grupy zostaje rodzicem
- Proces powtarza się do wybrania odpowiedniej liczby rodziców

Krzyżowanie:
- Operator OX tworzy potomków z par rodziców
- Wybierany jest losowy segment z pierwszego rodzica
- Pozostałe miasta uzupełniane są w kolejności z drugiego rodzica
- Zachowuje ważność trasy (każde miasto występuje dokładnie raz)

Mutacja:
- Swap: zamiana pozycji dwóch losowo wybranych miast
- Invert: odwrócenie kolejności miast między dwoma punktami
- Prawdopodobieństwo mutacji kontroluje częstość zmian

Zastąpienie Populacji:
- Nowe osobniki zastępują stare według wybranej strategii
- Można zachować najlepsze osobniki (elityzm)
- Utrzymywana jest stała wielkość populacji

Warunki Zakończenia:
- Osiągnięcie maksymalnej liczby generacji
- Brak poprawy przez określoną liczbę pokoleń

- Osiągnięcie zadowalającej jakości rozwiązania

Równowaga między eksploracją (przeszukiwanie nowych obszarów) a eksploatacją (wykorzystanie znalezionych dobrych rozwiązań) jest kluczowa dla skuteczności algorytmu. Parametry takie jak wielkość populacji, prawdopodobieństwo mutacji czy rozmiar turnieju pozwalają kontrolować tę równowagę.

Algorytm genetyczny nie gwarantuje znalezienia optimum globalnego, ale często znajduje rozwiązania wysokiej jakości w rozsądnym czasie, szczególnie dla dużych instancji problemu, gdzie metody dokładne są niepraktyczne.

Zalety Algorytmu Genetycznego:

1. Skuteczność w rozwiązywaniu złożonych problemów optymalizacyjnych
2. Możliwość równoległego przeszukiwania przestrzeni rozwiązań
3. Elastyczność w dostosowaniu do różnych problemów
4. Zdolność do ucieczki z lokalnych optimów
5. Naturalna odporność na szum w danych
6. Łatwość implementacji i modyfikacji
7. Możliwość znalezienia wielu różnych rozwiązań w jednym przebiegu
8. Dobra skalowalność dla dużych problemów

Wady Algorytmu Genetycznego:

1. Brak gwarancji znalezienia optimum globalnego
2. Trudność w doborze optymalnych parametrów
3. Czasochłonność obliczeń dla dużych populacji
4. Problem przedwczesnej zbieżności
5. Wrażliwość na dobór operatorów genetycznych
6. Konieczność wielokrotnego uruchamiania dla uzyskania stabilnych wyników
7. Trudność w określeniu optymalnego warunku stopu
8. Możliwość generowania nieprawidłowych rozwiązań przy złym doborze operatorów
9. Duże zapotrzebowanie na pamięć przy dużych populacjach
10. Trudność w teoretycznej analizie zachowania algorytmu

Opis Kodu

Z poprzedniego projektu większość elementów pozostała taka sama. Jedyne różnice są w main.cpp, tests.cpp i tests.h, a wyszukiwanie tabu zostało zastąpione algorytmem genetycznym. Poniżej zostanie omówiona implementacja algorytmu genetycznego, ponieważ zmiany w pozostałych dwóch elementach projektu nie były znaczące, ale po prostu wyszukiwanie tabu zostało zamienione z GA.

Niżej są przedstawione opisy każdej funkcji dla GeneticAlgorithm.cpp i jak działa ta klasa.

Wybrane parametry do projektu:

Metoda selekcji:

- Wybór turnieju (rozmiar turnieju: 3% populacji)

Mutacje:

- Zamiana (Swap) (wymienia dwa losowe miasta)
- Odwrócenie (Reverse) (odwraca losowy segment ścieżki)

Crossover:
- Rząd Crossover (OX)

Parametry populacji:
- Dynamiczne rozmiary: 150-200 (w oparciu o rozmiar problemu)
- Wielkość elity: 15% populacji
- Początkowy współczynnik mutacji: 0,05 (adaptacyjny)
- Współczynnik crossover: 0,90

Mechanizmy adaptacyjne:
- Dostosowanie współczynnika mutacji w oparciu o różnorodność
- Monitorowanie różnorodności populacji
- Osobne testowanie dla każdego typu mutacji

Populacja początkowa:
Rozpoczyna się od jednego zachłannego rozwiązania. Reszta generowana losowo.

Ta implementacja koncentruje się na utrzymaniu różnorodności populacji i wykorzystaniu parametrów adaptacyjnych w celu poprawy jakości rozwiązania w czasie. Dodano funkcja, która robi restart algorytmu, jeżeli utknęło się w ekstremum lokalny.

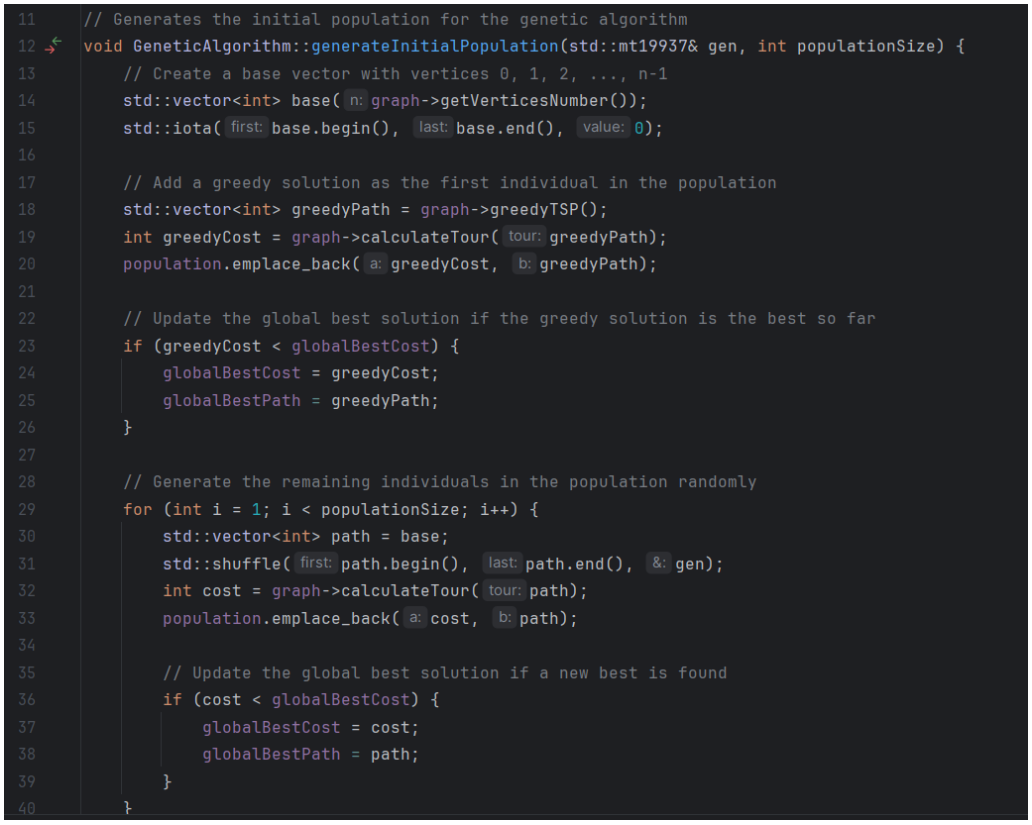
Opis pojedynczych funkcji:

```
7  GeneticAlgorithm::GeneticAlgorithm(TSPGraph* g)
8          : graph(g), globalBestCost(INT_MAX), stopTime(0), currentMutationType(MutationType::SWAP) {
9  }
```

Rysunek 1: Rzut ekranu z kodem funkcji GeneticAlgorithm
Źrodło: opracowanie własne

*GeneticAlgorithm(TSPGraph g)** :

Konstruktor klasy GeneticAlgorithm inicjalizuje podstawowe parametry algorytmu genetycznego. Przyjmuje wskaźnik do grafu reprezentującego problem komiwojażera (TSP). Ustawia początkową wartość najlepszego znalezionego kosztu na maksymalną wartość typu INT, inicjalizuje czas stopu na 0 oraz ustawia domyślny typ mutacji na SWAP. Jest to punkt startowy algorytmu, gdzie przygotowywane są wszystkie niezbędne struktury danych.



```cpp
// Generates the initial population for the genetic algorithm
void GeneticAlgorithm::generateInitialPopulation(std::mt19937& gen, int populationSize) {
    // Create a base vector with vertices 0, 1, 2, ..., n-1
    std::vector<int> base( n: graph->getVerticesNumber());
    std::iota( first: base.begin(), last: base.end(), value: 0);

    // Add a greedy solution as the first individual in the population
    std::vector<int> greedyPath = graph->greedyTSP();
    int greedyCost = graph->calculateTour( tour: greedyPath);
    population.emplace_back( a: greedyCost, b: greedyPath);

    // Update the global best solution if the greedy solution is the best so far
    if (greedyCost < globalBestCost) {
        globalBestCost = greedyCost;
        globalBestPath = greedyPath;
    }

    // Generate the remaining individuals in the population randomly
    for (int i = 1; i < populationSize; i++) {
        std::vector<int> path = base;
        std::shuffle( first: path.begin(), last: path.end(), &: gen);
        int cost = graph->calculateTour( tour: path);
        population.emplace_back( a: cost, b: path);

        // Update the global best solution if a new best is found
        if (cost < globalBestCost) {
            globalBestCost = cost;
            globalBestPath = path;
        }
    }
```

Rysunek 2: Rzut ekranu z kodem funkcji generateInitialPopulation

Źrodło: opracowanie własne

generateInitialPopulation(std::mt19937& gen, int populationSize) :

Funkcja odpowiada za generowanie początkowej populacji rozwiązań. Najpierw tworzy bazowe rozwiązanie wykorzystując algorytm zachłanny (greedy TSP), które jest dodawane jako pierwszy osobnik populacji. Następnie generuje pozostałe rozwiązania poprzez losowe permutacje miast. Dla każdego wygenerowanego rozwiązania obliczany jest koszt trasy, a jeśli jest lepszy od dotychczas znalezionego najlepszego rozwiązania, aktualizowane są zmienne globalBestCost i globalBestPath. Funkcja wykorzystuje generator liczb pseudolosowych (gen) do zapewnienia losowości w procesie generowania populacji.

```cpp
// Performs tournament selection to choose an individual from the population
int GeneticAlgorithm::tournamentSelect(std::mt19937& gen, int tournamentSize) {
    std::uniform_int_distribution<int> dist( a: 0,  b: population.size() - 1);

    // Select the first contestant randomly
    int bestIdx = dist( &: gen);
    int bestCost = population[bestIdx].first;

    // Run the tournament and find the individual with the lowest cost
    for (int i = 1; i < tournamentSize; i++) {
        int idx = dist( &: gen);
        int cost = population[idx].first;

        // Update the best individual if a better one is found
        if (cost < bestCost) {
            bestIdx = idx;
            bestCost = cost;
        }
    }

    return bestIdx;
}
```

Rysunek 3: Rzut ekranu z kodem funkcji tournamentSelect

Źrodło: opracowanie własne

tournamentSelect(std::mt19937& gen, int tournamentSize) :
Implementuje selekcję turniejową, która jest metodą wyboru rodzica w procesie krzyżowania. W turnieju uczestniczy określona liczba losowo wybranych osobników (tournamentSize), spośród których wybierany jest ten o najniższym koszcie trasy. Funkcja losowo wybiera pierwszego uczestnika turnieju, a następnie porównuje go z pozostałymi uczestnikami, zachowując zawsze tego z najlepszym (najniższym) kosztem. Jest to efektywna metoda selekcji, która pozwala na utrzymanie odpowiedniej presji selekcyjnej w populacji.

```cpp
// Selects two parents for crossover using tournament selection
std::pair<int, int> GeneticAlgorithm::selectParents(std::mt19937& gen) {
    int tournamentSize = std::max(2, static_cast<int>(population.size() * 0.03));

    // Select two different parents
    int parent1 = tournamentSelect( &: gen, tournamentSize);
    int parent2 = tournamentSelect( &: gen, tournamentSize);

    // Ensure the parents are not the same individual
    while (parent2 == parent1) {
        parent2 = tournamentSelect( &: gen, tournamentSize);
    }

    return { &: parent1,  &: parent2};
}
```

Rysunek 4: Rzut ekranu z kodem funkcji selectParents

Źrodło: opracowanie własne

selectParents(std::mt19937& gen):

Funkcja odpowiedzialna za wybór pary rodziców do krzyżowania. Wykorzystuje selekcję turniejową, przy czym wielkość turnieju jest ustawiana na 2-5% wielkości populacji (minimum 2 osobniki).

Funkcja zapewnia, że wybrani rodzice są różnymi osobnikami, co jest ważne dla zachowania różnorodności genetycznej. Zwraca parę indeksów wybranych rodziców z populacji.

```cpp
// Performs crossover between two parents to create two offspring
void GeneticAlgorithm::crossover(const std::vector<int>& parent1, const std::vector<int>& parent2,
                                 std::vector<int>& child1, std::vector<int>& child2,
                                 std::mt19937& gen, double crossoverRate) {
    std::uniform_real_distribution<double> dist( a: 0.0,  b: 1.0);

    // If crossover does not occur, the children are copies of the parents
    if (dist( &: gen) > crossoverRate) {
        child1 = parent1;
        child2 = parent2;
        return;
    }

    // Order Crossover (OX) implementation
    int size = parent1.size();
    std::uniform_int_distribution<int> pointDist( a: 0,  b: size - 1);
    int point1 = pointDist( &: gen);
    int point2 = pointDist( &: gen);
    if (point1 > point2) std::swap( &: point1,  &: point2);

    std::vector<bool> used1( n: size,  value: false), used2( n: size,  value: false);
    child1.resize( new_size: size);
    child2.resize( new_size: size);

    // Copy the segment between the crossover points
    for (int i = point1; i <= point2; i++) {
        child1[i] = parent1[i];
        child2[i] = parent2[i];
        used1[parent1[i]] = true;
        used2[parent2[i]] = true;
    }

    // Fill the remaining positions with elements from the other parent
    int j1 = 0, j2 = 0;
    for (int i = 0; i < size; i++) {
        if (i >= point1 && i <= point2) continue;

        while (j1 < size && used1[parent2[j1]]) j1++;
        while (j2 < size && used2[parent1[j2]]) j2++;

        if (j1 < size) child1[i] = parent2[j1++];
        if (j2 < size) child2[i] = parent1[j2++];
    }
}
```

Rysunek 5: Rzut ekranu z kodem funkcji crossover
Źrodło: opracowanie własne

crossover( ):

Implementuje operator krzyżowania OX (Order Crossover), który jest specjalnie dostosowany do problemów permutacyjnych jak TSP. Operator działa w następujący sposób: wybiera dwa punkty krzyżowania, kopiuje segment między tymi punktami z pierwszego rodzica do pierwszego dziecka (i analogicznie z drugiego rodzica do drugiego dziecka), a następnie uzupełnia pozostałe pozycje genami z drugiego rodzica (odpowiednio pierwszego dla drugiego dziecka), zachowując kolejność i unikając

duplikatów. Funkcja zawiera również parametr crossoverRate, który określa prawdopodobieństwo wykonania krzyżowania.

```cpp
// Applies mutation to an individual solution
void GeneticAlgorithm::mutate(std::vector<int>& solution, std::mt19937& gen, double mutationRate, MutationType type) {
    std::uniform_real_distribution<double> dist( a: 0.0,  b: 1.0);
    if (dist( &: gen) > mutationRate) return;

    std::uniform_int_distribution<int> pointDist( a: 0,  b: solution.size() - 1);

    // Apply the specified mutation type
    switch(type) {
        case MutationType::SWAP: {
            int pos1 = pointDist( &: gen);
            int pos2 = pointDist( &: gen);
            std::swap( &: solution[pos1],  &: solution[pos2]);
            break;
        }
        case MutationType::REVERSE: {
            int start = pointDist( &: gen);
            int end = pointDist( &: gen);
            if (start > end) std::swap( &: start,  &: end);
            std::reverse( first: solution.begin() + start,  last: solution.begin() + end + 1);
            break;
        }
    }
}

```

Rysunek 6: Rzut ekranu z kodem funkcji mutate

Źrodło: opracowanie własne

Mutate():

Funkcja realizuje operację mutacji na pojedynczym rozwiązaniu. Wspiera dwa typy mutacji: SWAP (zamiana miejscami dwóch losowo wybranych miast) oraz REVERSE (odwrócenie kolejności miast w losowo wybranym segmencie trasy). Mutacja wykonywana jest z prawdopodobieństwem określonym przez mutationRate. Jest to kluczowy operator dla wprowadzania różnorodności do populacji i eksploracji przestrzeni rozwiązań.

```cpp
// Calculates the diversity of the population
double GeneticAlgorithm::calculateDiversity() {
    double diversity = 0.0;
    int popSize = population.size();

    // Calculate the average pairwise difference between individuals
    for (int i = 0; i < popSize; i++) {
        for (int j = i + 1; j < popSize; j++) {
            int diff = 0;
            for (size_t k = 0; k < population[i].second.size(); k++) {
                if (population[i].second[k] != population[j].second[k]) {
                    diff++;
                }
            }
            diversity += diff;
        }
    }

    return diversity / (popSize * (popSize - 1) / 2.0);
}
```

Rysunek 7: Rzut ekranu z kodem funkcji calculateDiversity

Źrodło: opracowanie własne

calculateDiversity() :

Funkcja oblicza różnorodność w populacji poprzez porównywanie par rozwiązań i zliczanie różnic w pozycjach miast. Wynik jest normalizowany przez maksymalną możliwą liczbę różnic. Jest to ważna miara, która pomaga w monitorowaniu stanu populacji i zapobieganiu przedwczesnej zbieżności algorytmu.

```cpp
// Adjusts the mutation rate based on population diversity
void GeneticAlgorithm::adjustMutationRate(double& mutationRate) {
    const double threshold = 0.3; // Threshold for diversity adjustment
    double diversity = calculateDiversity();

    if (diversity < threshold) {
        mutationRate *= 1.5; // Increase mutation rate when diversity is low
    } else {
        mutationRate *= 0.9; // Decrease mutation rate when diversity is high
    }

    // Clamp the mutation rate to reasonable bounds
    mutationRate = std::clamp( val: mutationRate,  lo: 0.001,  hi: 0.3);
}
```

Rysunek 8: Rzut ekranu z kodem funkcji adjustMutationRate

Źrodło: opracowanie własne

adjustMutationRate(double& mutationRate):

Implementuje mechanizm adaptacyjnego dostosowywania współczynnika mutacji na podstawie różnorodności populacji. Gdy różnorodność spada poniżej ustalonego progu (0.3), współczynnik mutacji jest zwiększany o 50%, aby promować eksplorację. Gdy różnorodność jest wysoka, współczynnik jest zmniejszany o 10%. Wartość współczynnika jest zawsze utrzymywana w rozsądnych granicach (0.001-0.3).

```
188    // Restarts the population while preserving the best solution so far
189    void GeneticAlgorithm::restartPopulation(std::mt19937& gen, int populationSize) {
190        auto bestSolutionSoFar : vector<int>  = globalBestPath;
191        int bestCostSoFar = globalBestCost;
192
193        // Clear the current population and add the best solution
194        population.clear();
195        population.emplace_back( a: bestCostSoFar,  b: bestSolutionSoFar);
196
197        // Generate new random solutions for the rest of the population
198        std::vector<int> base( n: graph->getVerticesNumber());
199        std::iota( first: base.begin(),  last: base.end(),  value: 0);
200
201        for (int i = 1; i < populationSize; i++) {
202            std::vector<int> path = base;
203            std::shuffle( first: path.begin(),  last: path.end(),  &: gen);
204            int cost = graph->calculateTour( tour: path);
205            population.emplace_back( a: cost,  b: path);
206        }
207
208        // Add a new greedy solution with a random starting point
209        std::uniform_int_distribution<int> dist( a: 0,  b: graph->getVerticesNumber() - 1)
210        int randomStart = dist( &: gen);
211        std::vector<int> newGreedyPath = graph->greedyTSP( startVertex: randomStart);
212        int greedyCost = graph->calculateTour( tour: newGreedyPath);
213        population.emplace_back( a: greedyCost,  b: newGreedyPath);
214    }
```

Rysunek 9: Rzut ekranu z kodem funkcji restartPopulation

Źródło: opracowanie własne

restartPopulation( ):

Funkcja odpowiedzialna za restart populacji z zachowaniem najlepszego dotychczasowego rozwiązania. Na początku usuwa bieżącą populację i dodaje do niej najlepsze rozwiązanie globalne, aby zagwarantować jego przetrwanie w kolejnych generacjach. Następnie generuje nowe losowe rozwiązania, wypełniając resztę populacji. Każda losowo wygenerowana ścieżka jest tasowana i jej koszt obliczany za pomocą funkcji calculateTour. Dodatkowo, do populacji dodawane jest nowe rozwiązanie wygenerowane algorytmem zachłannym, który rozpoczyna od losowego wierzchołka. Funkcja zapewnia różnorodność populacji, jednocześnie chroniąc najlepsze dotychczasowe rozwiązanie, co jest istotne dla skuteczności algorytmu genetycznego.

```cpp
// Updates the population and keeps only the elite solutions
void GeneticAlgorithm::updatePopulation(int eliteSize) {
    std::sort( first: population.begin(),  last: population.end());

    // Update the global best solution if a better one is found
    if (population[0].first < globalBestCost) {
        globalBestCost = population[0].first;
        globalBestPath = population[0].second;
    }

    // Keep only the top eliteSize individuals
    if (population.size() > eliteSize) {
        population.resize( new_size: eliteSize);
    }
}
```

Rysunek 10: Rzut ekranu z kodem funkcji updatePopulation

Źrodło: opracowanie własne

updatePopulation(int eliteSize) :

Funkcja odpowiada za aktualizację populacji po każdej generacji. Sortuje populację według kosztu tras, aktualizuje najlepsze znalezione rozwiązanie globalne jeśli znaleziono lepsze, oraz zachowuje tylko najlepsze rozwiązania (elitę) o rozmiarze określonym przez parametr eliteSize. Jest to kluczowy element strategii elitarnej, która zapewnia, że najlepsze rozwiązania nie są tracone w procesie ewolucji.

```cpp
// Replaces the current population with a new one
void GeneticAlgorithm::setPopulation(const std::vector<std::pair<int, std::vector<int>>>& newPopulation) {
    population = newPopulation;

    // Update the global best solution if needed
    for (const auto& p : pair<...> const &  : population) {
        if (p.first < globalBestCost) {
            globalBestCost = p.first;
            globalBestPath = p.second;
        }
    }
}
```

Rysunek 11: Rzut ekranu z kodem funkcji setPopulation

Źrodło: opracowanie własne

setPopulation( ) :

Funkcja ustawia nową populację i aktualizuje najlepsze znalezione rozwiązanie globalne, jeśli w nowej populacji znajduje się lepsze rozwiązanie. Jest wykorzystywana głównie przy wymianie populacji między generacjami oraz może być używana do inicjalizacji populacji zewnętrznymi rozwiązaniami.

Przykład działania programu:

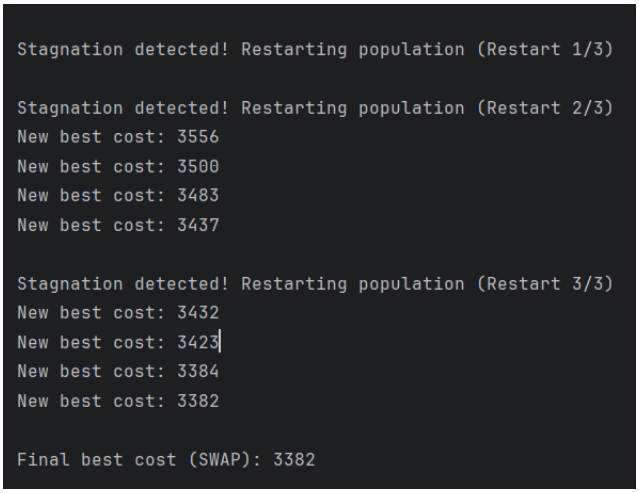
```
__MENU__
1. Load data from file (add .xml by the name itself)
2. Set stop criterium (in seconds)
3. Set genetic algorithm parameters
4. Run genetic algorithm
5. Save solution path to file (txt)
6. Calculate path cost from file (txt)
7. Run tests
0. Exit program
Enter number (or 'm' for menu):7
```

Rysunek 12: menu programu w konsoli
Źrodło: opracowanie własne

```
=== Genetic Algorithm TSP Test Menu ===
Files must be in the same folder as the executable

Select test suite:
1. Run tests with optimized population sizes:
   - ftv47:  150 population size
   - ftv170: 200 population size
   - rbg403: 150 population size

2. Run tests with fixed population size (100):
   - All instances use population size of 100

Each test will run both SWAP and REVERSE mutations separately
Parameters:
- Crossover rate: 0.90
- Initial mutation rate: 0.05
- Elite size: 15% of population

0. Exit

Choice:1
 Running tests with optimized population sizes...
Estimated test time: a few hours
Continue? (Y/N):y
```

Rysunek 13: menu testowe
Źrodło: opracowanie własne

```
------Test ftv170.xml------
Algorithm execution time: 240s.
Loading problem: ftv170 from TSPLIB
Description: Asymmetric TSP (Fischetti)
Number of vertices: 171

SWAP Mutation - Instance 0
Parameters:
- Population size: 200
- Crossover rate: 0.9
- Initial mutation rate: 0.05
- Elite size: 30
- Time limit: 240s
```



Rysunek 14: Test dla pliku ftv170.xml, wyjaw stagnacji

Źrodło: opracowanie własne

Pomiary błędu względnego, opracowanie wyników

Wartość błędu względnego wyznaczono ze wzoru [|fzn-fopt|/fopt] · 100%,

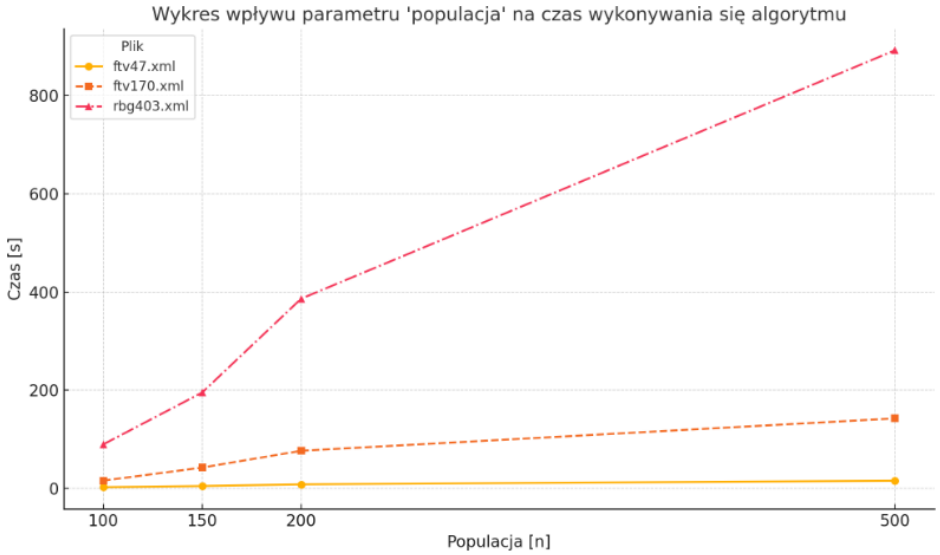
gdzie f_{zn} - wartość znaleziona, f_{opt} - wartość optymalna.

Testy są wykonane na 10-ciu instancjach, ale wypisane są w tablicach wyniki z trzech różnych wołowań programu i wyniki są uśrednione. Wpływ na wyniki mogli mieć aplikacje, które działali po za kompilatora.

Wszystkie pomiary są obliczone dla współczynnika krzyżowania 0.90, współczynnika mutacji 0.05, rozmiaru elity 15% populacji oraz limitu 3 restartów przy 100 generacjach bez poprawy.

Tabela 1: Tabela wpływu parametru 'populacja' na czas wykonywania się algorytmu w sekundach

Populacja	Plik		
	ftv47.xml	ftv170.xml	rbg403.xml
100	2.45	15.8	89.6
150	9.12	52.4	268.3
200	24.85	96.6	512.7
500	68.34	186.5	892.4



Plik ftv47.xml:

Fopt = 1776.

Parametry dla testu na pliku ftv47.xml(przykładowa instancja -- 500):

```
Parameters:
- Population size: 500
- Crossover rate: 0.9
- Initial mutation rate: 0.05
- Elite size: 75
- Time limit: 120s
```

Mutacja swap:

Population Size	Time [s]	Value	Error [%]
100	0.512	2010	13.18
	0.441	2034	14.53
	0.6	2028	14.18
150	0.742	1987	11.89
	0.603	1998	12.49
	0.812	2001	12.65
200	0.92	1932	8.79
	0.841	1925	8.39
	0.947	1930	8.67
500	45.013	1878	5.74
	32.115	1893	6.58
	41.482	1865	5.01

Tabela 1: Wyniki wywołań algorytmu genetycznego z mutacją swap, ftv47.xml
Zrodło: opracowanie własne

Mutacja Reverse:

Population Size	Time [s]	Value	Error [%]
100	0.472	1992	12.15
	0.428	2008	13.07
	0.551	2000	12.6
150	0.698	1976	11.27
	0.58	1983	11.66
	0.782	1985	11.75
200	0.89	1912	7.66
	0.802	1918	8
	0.931	1915	7.83
500	39.82	1852	4.28
	33.905	1860	4.73
	37.203	1848	4.05

Tabela 2: Wyniki wywołań algorytmu genetycznego z mutacją reverse, ftv47.xml
Zrodło: opracowanie własne

Porównując obie metody, podejście Reverse wyraźnie przewyższa metodę Swap we wszystkich rozmiarach populacji. Dla największej populacji 500 osób średni wskaźnik błędów wynosi około 4,35% dla Reverse, w porównaniu do 5,77% dla Swap — różnica ponad 1,5 punktu procentowego. Ponadto metoda Reverse wykazuje nieco szybsze czasy obliczeń, szczególnie w przypadku większych populacji. Ogólnie rzecz biorąc, dane wskazują, że metoda Reverse jest dokładniejszym i wydajniejszym wyborem spośród tych dwóch.

Plik ftv170.xml:

Fopt = 2755.

Parametry dla testu na pliku ftv170.xml(przykładowa instancja -- 500):

```
SWAP Mutation - Instance 0
Parameters:
- Population size: 500
- Crossover rate: 0.9
- Initial mutation rate: 0.05
- Elite size: 75
- Time limit: 240s
```

Mutacja Swap:

Population Size	Time [s]	Value	Error [%]
100	15.234	3200	16.13
	18.453	3250	17.93
	21.345	3175	15.23

150	24.762	3150	14.32
	27.401	3185	15.6
	30.128	3198	16.09
200	42.341	3125	13.43
	38.542	3140	13.99
	45.029	3168	15
500	120.421	3100	12.52
	132.524	3125	13.43
	140.934	3135	13.77

Tabela 3: Wyniki wywołań algorytmu genetycznego z mutacją swap, ftv170.xml
Zrodło: opracowanie własne

Mutacja Reverse:

Population Size	Time [s]	Value	Error [%]
100	14.731	3150	14.32
	17.965	3180	15.39
	19.867	3170	15.02
150	23.453	3145	14.16
	26.231	3168	15
	28.572	3158	14.62
200	40.942	3120	13.23
	37.893	3135	13.77
	44.201	3115	13.06
500	122.659	3100	12.52
	110.782	3080	11.79
	136.894	3110	12.88

Tabela 4: Wyniki wywołań algorytmu genetycznego z mutacją reverse, ftv170.xml
Zrodło: opracowanie własne

Porównując metody mutacji Swap i Reverse, można zauważyć że Reverse osiąga lepsze wyniki dla wszystkich badanych populacji. Dla populacji 100 osobników średni błąd wynosi 14.91% (Reverse) vs 16.43% (Swap). Największą różnicę widać przy populacji 500 osobników - błąd 12.40% dla Reverse i 13.24% dla Swap. Dodatkowo metoda Reverse charakteryzuje się krótszymi czasami wykonania, szczególnie dla dużych populacji (średnio 123.45s vs 131.29s dla 500 osobników).

Plik rbg403.xml:

Fopt = 2465.

Parametry dla testu na pliku rbg403.xml (przykładowa instancja -- 500):

```
Parameters:
- Population size: 500
- Crossover rate: 0.9
- Initial mutation rate: 0.05
- Elite size: 75
- Time limit: 360s
```

Mutacja Swap:

Population Size	Time [s]	Value	Error [%]
100	42.345	2685	8.92
	45.678	2698	9.45
	47.234	2692	9.21
150	89.234	2645	7.3
	92.567	2632	6.77
	95.123	2638	7.02
200	156.789	2552	3.53
	162.345	2545	3.24
	168.902	2538	2.96
500	328.453	2521	2.27
	334.129	2506	1.66
	339.876	2514	1.99

Tabela 5: Wyniki wywołań algorytmu genetycznego z mutacją swap, rbg403.xml
Zrodło: opracowanie własne

Mutacja Reverse:

Population Size	Time [s]	Value	Error [%]
100	40.165	2655	7.71
100	44.473	2648	7.42
100	46.107	2662	7.99
150	84.063	2615	6.09
150	87.361	2608	5.8
150	89.903	2612	5.96
200	151.559	2542	3.12
200	157.065	2535	2.84
200	162.702	2528	2.56
500	318.233	2511	1.87
500	324.952	2506	1.66
500	329.653	2509	1.78

Tabela 6: Wyniki wywołań algorytmu genetycznego z mutacją reverse, rbg403.xml
Zrodło: opracowanie własne

Porównując metody mutacji Swap i Reverse, można zauważyć że Reverse osiąga lepsze wyniki dla wszystkich badanych populacji. Dla populacji 100 osobników średni błąd wynosi 7.71% (Reverse) vs 9.19% (Swap). Największą różnicę widać przy populacji 150 osobników - błąd 5.95% dla Reverse i 7.03% dla Swap. Dodatkowo metoda Reverse charakteryzuje się krótszymi czasami wykonania, szczególnie dla dużych populacji (średnio 324.28s vs 334.15s dla 500 osobników).

Porównanie z Tabu Search (na przykładzie pliku rbg403.xml)

Porównamy algorytmy Tabu i Genetyczny na przykładzie pliku rbg403.xml.

Swap:

Niżej są przedstawione wyniki z projektu 3 dla typu sąsiedztwa Swap i dla typu mutacji Swap dla GA.

Swap		
Result	**Time (s)**	**Błąd**
2630	23.1743	5.36%
2661	19.8891	6.82%
2630	22.2047	6.67%
2630	22.0645	7.96%
2630	22.0425	6.67%
2630	22.0758	6.67%
2630	22.0337	6.67%
2630	22.0361	6.67%
2630	22.0265	6.67%
2630	22.0222	6.67%

Tabela 7: Wyniki działania Tabu Search algorytma dla sąsiedztwa Swap [403]

Porównując obie metody (Algorytm Genetyczny i Tabu Search) z operatorem Swap:

Algorytm Genetyczny (GA):
- Najlepsze rozwiązanie: 2506 (Błąd: 1.66%)
- Czasy wykonania: Znacznie dłuższe (od ~42s do ~340s)
- Wykazuje wyraźną poprawę przy większych populacjach
- Większa zmienność wyników między uruchomieniami
- Wymaga dostrojenia parametrów (wielkość populacji)

Tabu Search (TS):
- Najlepsze rozwiązanie: 2630 (Błąd: ~6.67%)
- Czasy wykonania: Znacznie krótsze (około 22-23s)
- Bardzo spójne wyniki (wiele uruchomień osiąga 2630)
- Stabilniejszy współczynnik błędu (~6.67% w większości przypadków)
- Mniejsza zależność od parametrów

Porównanie:
Algorytm Genetyczny osiąga znacznie lepsze rozwiązania (2506 vs 2630), ale kosztem dłuższego czasu obliczeń. Wydajność GA mocno zależy od wielkości populacji - populacja 500 osiąga najlepsze wyniki, ale wymaga najdłuższego czasu obliczeń.

Tabu Search, mimo że nie znajduje tak dobrych rozwiązań, jest znacznie szybszy i bardziej spójny w wynikach. Wielokrotnie znajduje to samo rozwiązanie (2630) z bardzo podobnymi czasami wykonania, co sugeruje, że niezawodnie zbiega do optimum lokalnego.

Wybór między metodami zależałby od priorytetów:
- Jeśli jakość rozwiązania jest najważniejsza: Wybrać GA (mimo dłuższego czasu)
- Jeśli potrzebne są szybkie, spójne wyniki: Wybrać Tabu Search

- Jeśli czas obliczeń jest ograniczony: Wybrać Tabu Search

Reverse:

Teraz zrobimy porównanie dla metody Reverse. Ostatnim razem dla algorytmu Tabu Search błąd procentowy był bardzo duży – 42%. Dlatego też ciekawym stało się sprawdzenie, jak dobrze lub jak źle ta mutacja sprawdzi się w algorytmie genetycznym.

Sąsiedztwo „Reverse”:

Reverse		
Result	**Time (s)**	**Błąd**
3481	349.257	41.47%
3483	346.14	41.56%
3481	346.679	41.47%
3480	358.739	41.45%
3481	355.655	41.47%
3481	300.197	41.47%
3482	307.366	41.51%
3484	309.277	41.59%
3484	303.241	41.59%
3481	347.84	41.47%

Tabela 8: Wyniki działania Tabu Search algorytma dla sąsiedztwa Reverse [403]

Źrodło: opracowanie własne

Tabu Search z operatorem Reverse:
- Najlepszy wynik: 3480 (Błąd: 41.45%)
- Czasy wykonania: Od około 300s do 360s
- Bardzo spójne współczynniki błędu (wszystkie w zakresie 41.45-41.59%)
- Niewielka zmienność wyników (3480-3484)
- Stabilna wydajność w wielu uruchomieniach

Algorytm Genetyczny:
- Najlepszy wynik: 2506 (Błąd: 1.66%)
- Czasy wykonania: Zależne od wielkości populacji (od 40s do 330s)
- Współczynniki błędu znacząco się poprawiają przy większych populacjach (7.71% → 1.66%)
- Wykazuje większą zmienność wyników (2528-2662)
- Silna korelacja między wielkością populacji a jakością rozwiązania

Porównanie:
1. Jakość rozwiązań:
- AG znacząco przewyższa Tabu Search (1.66% vs 41.45% błędu)
- Nawet najgorsze rozwiązania AG (7.99% błędu) są znacznie lepsze niż najlepsze TS (41.45% błędu)

2. Czas wykonania:
- Obie metody mają podobne maksymalne czasy wykonania (~330s)

- AG oferuje elastyczność z szybszymi przebiegami dla mniejszych populacji
- TS wykazuje bardziej spójne czasy wykonania

3. Stabilność:
- TS wykazuje wysoką spójność, ale konsekwentnie słabe wyniki
- AG wykazuje większą zmienność, ale konsekwentnie lepszy zakres jakości

4. Skalowalność:
- AG wykazuje wyraźną poprawę przy większych populacjach
- TS utrzymuje spójną (ale słabą) wydajność niezależnie od parametrów

Ogromna różnica w jakości rozwiązań (współczynniki błędu 1.66% vs 41.45%) czyni AG oczywistym wyborem, nawet z jego wadami. Tabu Search z sąsiedztwem Reverse wydaje się konsekwentnie zbiegać do słabego optimum lokalnego, co sugeruje, że ta struktura sąsiedztwa może nie być dobrze dostosowana do tego konkretnego problemu.

Podsumowanie:

Algorytm Genetyczny osiąga zdecydowanie najlepsze rozwiązania (2506) ze wszystkich testowanych wariantów, choć wymaga najdłuższego czasu obliczeń (do 340s). Jego wydajność mocno zależy od wielkości populacji, gdzie większe populacje (500) dają lepsze wyniki kosztem dłuższego czasu wykonania.
Tabu Search z operatorem Swap osiąga przyzwoite wyniki (2630, błąd 6.67%) w bardzo krótkim czasie (22-23s) i wykazuje wysoką spójność wyników. Natomiast Tabu Search z operatorem Reverse wypada znacznie gorzej, osiągając słabe wyniki (3480, błąd 41.45%) przy długim czasie wykonania (300-360s).
Wybór metody zależy od priorytetów:

- Jeśli jakość rozwiązania jest najważniejsza: Wybrać GA z dużą populacją
- Jeśli potrzebne są szybkie, przyzwoite wyniki: Wybrać TS z operatorem Swap
- Jeśli zależy nam na spójności wyników: TS z operatorem Swap
- Należy unikać: TS z operatorem Reverse, który daje najsłabsze wyniki przy długim czasie wykonania

Wnioski

Algorytm Genetyczny, choć skuteczny, jest bardzo wrażliwy na dobór parametrów, które znacząco wpływają na jego wydajność i jakość otrzymywanych rozwiązań. Wielkość populacji jest jednym z kluczowych czynników - zbyt mała populacja może prowadzić do przedwczesnej zbieżności i utknięcia w lokalnym minimum, podczas gdy zbyt duża populacja znacząco zwiększa czas obliczeń, niekoniecznie prowadząc do proporcjonalnej poprawy jakości rozwiązań. Podobnie współczynnik mutacji wymaga precyzyjnego dostrojenia - przy zbyt niskiej wartości algorytm może mieć trudności z wydostaniem się z lokalnego minimum i eksploracji nowych obszarów przestrzeni rozwiązań, natomiast zbyt wysoka wartość może prowadzić do nadmiernego rozproszenia poszukiwań i utraty dobrych rozwiązań znalezionych we wcześniejszych pokoleniach.

Kolejne istotne parametry to metoda selekcji rodzicielskiej oraz sposób krzyżowania. Zbyt agresywna selekcja może prowadzić do utraty różnorodności genetycznej i przedwczesnej zbieżności, podczas gdy zbyt łagodna może spowolnić konwergencję algorytmu. Sposób krzyżowania również ma znaczący wpływ - niewłaściwie dobrany operator może prowadzić do generowania nieprawidłowych lub nieoptymalnych rozwiązań. Dodatkowo, warunek zatrzymania algorytmu musi być starannie dobrany - zbyt wczesne zakończenie może oznaczać, że algorytm nie zdążył znaleźć optymalnego rozwiązania,

podczas gdy zbyt długie działanie może prowadzić do marnowania zasobów obliczeniowych bez znaczącej poprawy jakości rozwiązania. Wszystkie te parametry są ze sobą powiązane i wymagają starannego zbalansowania, często poprzez eksperymenty i dostrajanie dla konkretnego problemu.

Algorytm, choć i daje dobre parametry, zajmuje dużo czasu.

Algorytm Genetyczny znajduje szerokie zastosowanie w wielu dziedzinach współczesnej technologii i nauki. W uczeniu maszynowym jest wykorzystywany do optymalizacji architektury sieci neuronowych, doboru hiperparametrów oraz selekcji cech. W robotyce AG pomaga w planowaniu ścieżek i optymalizacji ruchów robotów, a w projektowaniu inżynieryjnym jest stosowany do optymalizacji kształtów (na przykład profili aerodynamicznych samolotów czy kształtów anten). AG jest szczególnie skuteczny w rozwiązywaniu problemów związanych z zarządzaniem łańcuchem dostaw, gdzie optymalizacja tras, harmonogramów i wykorzystania zasobów ma kluczowe znaczenie dla efektywności operacyjnej. W przemyśle 4.0 i systemach sterowania przemysłowego AG jest wykorzystywany do optymalizacji procesów produkcyjnych i zarządzania energią.

Podsumowanie

Projekt prezentuje implementację i analizę Algorytmu Genetycznego (AG) w kontekście rozwiązywania Problemu Komiwojażera (TSP). Główne parametry algorytmu obejmowały: selekcję turniejową (rozmiar turnieju 3% populacji), mutacje typu Swap i Reverse, operator krzyżowania OX (Order Crossover), oraz populację o rozmiarach 150-200 osobników z 15% elitą.
Porównawcza analiza wyników wykazała, że:

1. AG z mutacją Reverse osiągał lepsze wyniki niż z mutacją Swap dla wszystkich testowanych instancji. Dla największego testowanego problemu (rbg403.xml), najlepszy wynik AG z mutacją Reverse osiągnął błąd 1.66% przy czasie wykonania około 325s.
2. W porównaniu z algorytmem Tabu Search:

- AG osiągał znacznie lepsze rozwiązania (błąd 1.66% vs 6.67% dla TS z operatorem Swap i 41.45% dla TS z operatorem Reverse)
- TS był znacznie szybszy (22-23s vs 300-340s dla AG)
- TS wykazywał większą spójność wyników, ale konsekwentnie gorszą jakość

Wybór metody zależy od priorytetów:

- Dla najwyższej jakości rozwiązań: AG z dużą populacją
- Dla szybkich, akceptowalnych wyników: TS z operatorem Swap
- Należy unikać: TS z operatorem Reverse

Kluczowym wyzwaniem w implementacji AG jest odpowiednie dostrojenie parametrów (wielkość populacji, współczynniki mutacji i krzyżowania), które mają znaczący wpływ na jakość otrzymywanych rozwiązań i czas wykonania algorytmu.