SPRAWOZDANIE

Projektowanie efektywnych algorytmów

Zadanie projektowe nr 3

Implementacja i analiza efektywności algorytmu genetycznego dla problemu komiwojażera (TSP)

Autor:
Patryk Dulęba 259213
Czwartek 11:15

Prowadzący:
Mgr inż. Antoni Sterna

Spis treści

1 Opis rozpatrywanego problemu

Celem tego zadania projektowego było zaimplementowanie oraz późniejsza analiza efektywności algorytmu znajdującego rozwiązanie dla problemu komiwojażera. Problem komiwojażera polega na poszukiwaniu w grafie takiego cyklu, który zawiera wszystkie wierzchołki (każdy tylko raz), kończy się i zaczyna w tym samym punkcie i ma jak najmniejszy koszt całej drogi. Inaczej mówiąc problem komiwojażera polega na znalezieniu cyklu Hammiltona o najmniejszej wadze.

Aktualnie nie jest znany efektywny, czyli działający w czasie co najwyżej wielomianowym algorytm dający gwarancję znalezienia optymalnego rozwiązania problemu komiwojażera. Z tego powodu problem ten jest zaliczany do klasy problemów NP-trudnych. Dzieje się tak z powodu ogromnej liczby kombinacji do sprawdzenia przy grafach o większej liczbie wierzchołków. W grafie pełnym mającym n wierzchołków liczba możliwych cykli Hammiltona wynosi aż $\frac{(n-1!)}{2}$.

2 Algorytm genetyczny (GA)

Algorytm genetyczny (GA) jest heurystycznym algorytmem optymalizacji stosowanym do rozwiązywania problemów z zakresu inteligencji obliczeniowej. Jest to algorytm ewolucyjny, który wykorzystuje techniki selekcji, mutacji, krzyżowania i selekcji przystosowania, aby wybrać najlepszy zestaw parametrów dla danego problemu. GA może być stosowany do zarówno prostych, jak i złożonych problemów, takich jak optymalizacja czasu, optymalizacja kosztów, optymalizacja wykorzystania zasobów itp.

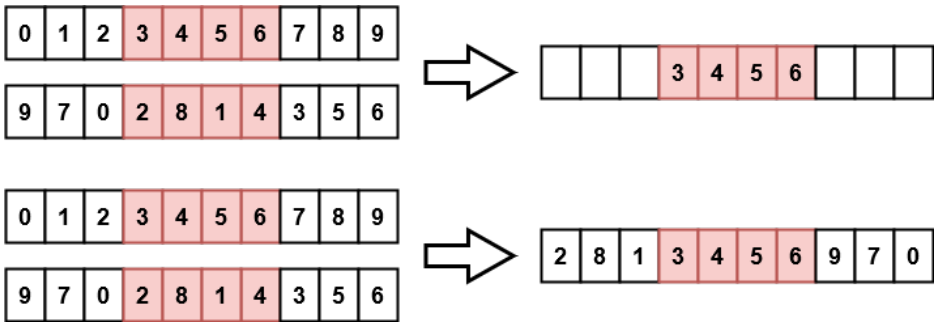
2.0.1 Opis działania algorytmu

Działanie algorytmu genetycznego opisane w krokach:

1. Inicjalizacja: tworzy się początkową populację losowych rozwiązań.
2. Ocena: każde rozwiązanie jest oceniane na podstawie jego wyniku, w zależności od problemu.
3. Selekcja: wybiera się najlepsze rozwiązania z populacji zgodnie z zasadami selekcji.
4. Krzyżowanie: wybrane osobniki są łączone ze sobą losowo, aby stworzyć nowe, wykorzystując technikę krzyżowania.
5. Mutacja: wybrane osobniki są następnie poddawane mutacji, aby wprowadzić nowe cechy do populacji.
6. Selekcja przystosowania: wybiera się najlepsze rozwiązania z populacji zgodnie z zasadami przystosowania.
7. Powtarzalność: powtarza się kroki 2-6, aż do osiągnięcia zdefiniowanego poziomu optymalizacji.

2.0.2 Najważniejsze elementy zaimplementowanego Algorytmu genetycznego

- **Selekcja turniejowa** - Selekcja turniejowa jest jedną z technik selekcji stosowanej w algorytmie genetycznym. Polega ona na wybraniu grupy losowych rozwiązań z populacji, a następnie porównaniu ich pod kątem jakości. Przegrana jednostka jest odrzucana, a zwycięzca jest zatrzymywany w populacji i staje się dostępny do reprodukcji. Proces jest powtarzany, aż wybierze się wymaganą liczbę jednostek. Selekcja turniejowa jest często używana w algorytmach genetycznych, ponieważ jest szybka i skuteczna.
- **Krzyżowanie OX** - Krzyżowanie typu OX jest jedną z technik krzyżowania stosowanych w algorytmie genetycznym. Polega na wylosowaniu dwóch punktów w sekwencjach genów u pierwszego rodzica i skopiowaniu fragmentu pomiędzy nimi od pierwszego rodzica do pierwszego dziecka, następnie zaczynając od drugiego punktu u dziecka skopiować po kolei od początku od drugiego rodzica pozostałe nieużyte i niepowtarzające się geny do pierwszego dziecka. Analogicznie postąpić dla drugiego dziecka odwracając role rodziców.



Rysunek 1: Przykład krzyżowania OX

- **Mutacja** - Mutacja jest jedną z technik stosowanych w algorytmie genetycznym. Polega ona na losowym zmienianiu elementów chromosomu, aby wprowadzić nowe cechy do populacji. Mutacja jest często stosowana w algorytmach genetycznych, ponieważ umożliwia populacji odświeżenie i pomaga w uniknięciu zablokowania w określonej przestrzeni, czyli w tzw. minimum lokalnym. W programie zastosowane zostały 2 typy mutacji :
  - **SWAP** – losowa zamiana miejscami dwóch genów.
  - **INVERT** – odwrócenie kolejności genów w wylosowanym podciągu.

3 Opis najważniejszych klas w projekcie

3.1 Klasa main

Klasa zawierająca menu główne napisanego programu.

```
//Glowne menu programu
int main(int, char**) {

    AdjacencyMatrix mat = *(new AdjacencyMatrix());
    srand(time(NULL));
    bool koniec = false;
    char wybor;

    system("cls");

    while (!koniec) {

        system("cls");
        cout << " ============MENU============ " << endl;
        cout << "-----------------------------" << endl;
        cout << "-----------------------------" << endl;
        cout << "1.  Wczytaj dane z pliku" << endl;
        cout << "2.  Automatyczne generowanie grafu" << endl;
        cout << "x.  Zamkniecie programu " << endl;
        cout << "-----------------------------" << endl;
        cout << "-----------------------------" << endl;
        cout << "Wybor: ";
        cin >> wybor;
```

Rysunek 2: Klasa main

3.2 Klasa AdjacencyMatrix

Klasa odpowiedzialna za generowanie, wczytywanie oraz przechowywanie danych o grafie.

```cpp
#include <iostream>

using namespace std;

class AdjacencyMatrix{
    private:
        int **Matrix;
        int size;

    public:
        AdjacencyMatrix();
        ~AdjacencyMatrix();
        int** getMatrix();
        int getSize();
        bool readMatrix(string name);
        void show();
        void generateMatrix(int mainSize);
};
```

Rysunek 3: Klasa AdjacencyMatrix

3.3 Klasa Genetic

Klasa odpowiedzialna za wykonywanie algorytmu genetycznego.

```cpp
#include <vector>
#include <iostream>
#include <chrono>
#include <random>
#include <algorithm>

using namespace std;

class Genetic
{
private:

    struct element {
        int pathSize;
        vector<int> path;
    };

    int stop;
    int **matrix;
    int size;
    int populationSize;
    float crossRate, mutationRate;
    double t;

public:

    int path_distance(const vector<int>& path, int** Matrix);
    void CrossoverOX(vector<int> &parent1, vector<int> &parent2, vector<element> &population, int** Matrix);
    bool isInPath(int element, int begin, int end, vector<int> &path);
    int TSP(bool mutationType, int popSize, float crossRat, float mutationRat, double max_iterations, int** Matrix, int mainSize);
    void mutationSWAP(vector<element> &population, int** Matrix);
    void mutationINVERT(vector<element> &population, int** Matrix);
};
```

Rysunek 4: Klasa Genetic

3.4 Klasa Time

Klasa odpowiedzialna za pomiar czasu wykonywania algorytmów.

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

class Timer {
public:
    high_resolution_clock::time_point startTime;
    high_resolution_clock::time_point endTime;

    void start();
    void stop();
    double timeCount();
};
```

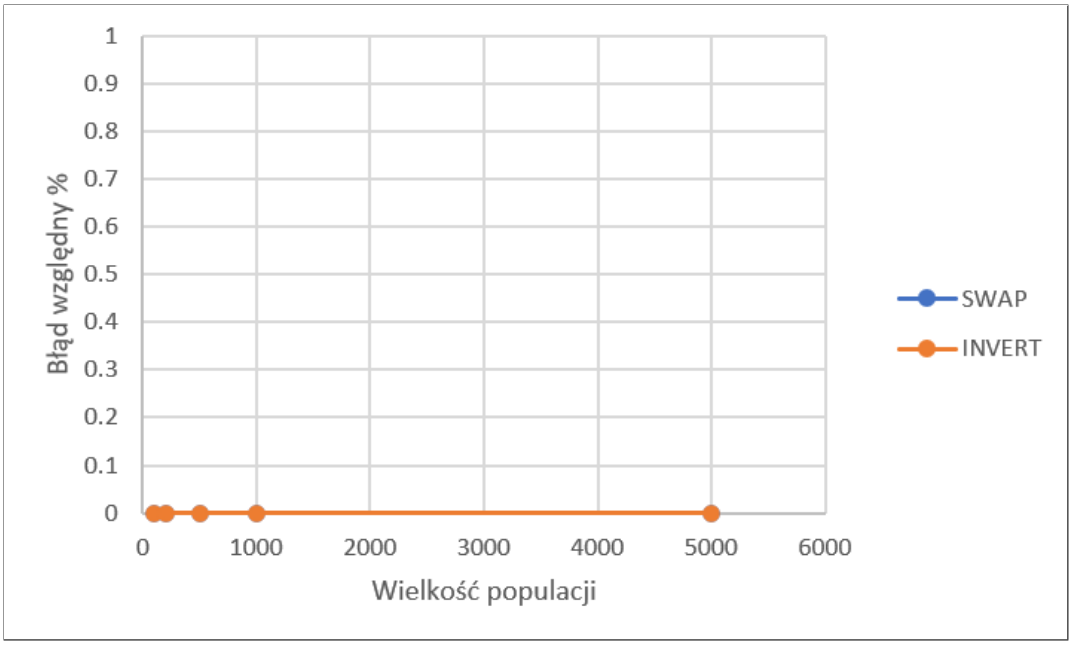
Rysunek 5: Klasa Time

4 Wyniki eksperymentu

4.1 Pomiary dla 17 miast

Wielkość populacji	Czas[s]	Mutacja	
		SWAP	INVERT
100	60	0%	0%
200		0%	0%
500		0%	0%
1000		0%	0%
5000		0%	0%

Tabela 1: Wyniki pomiarów błędu względnego algorytmu genetycznego w zależności od wielkości populacji i typu mutacji dla 17 miast.

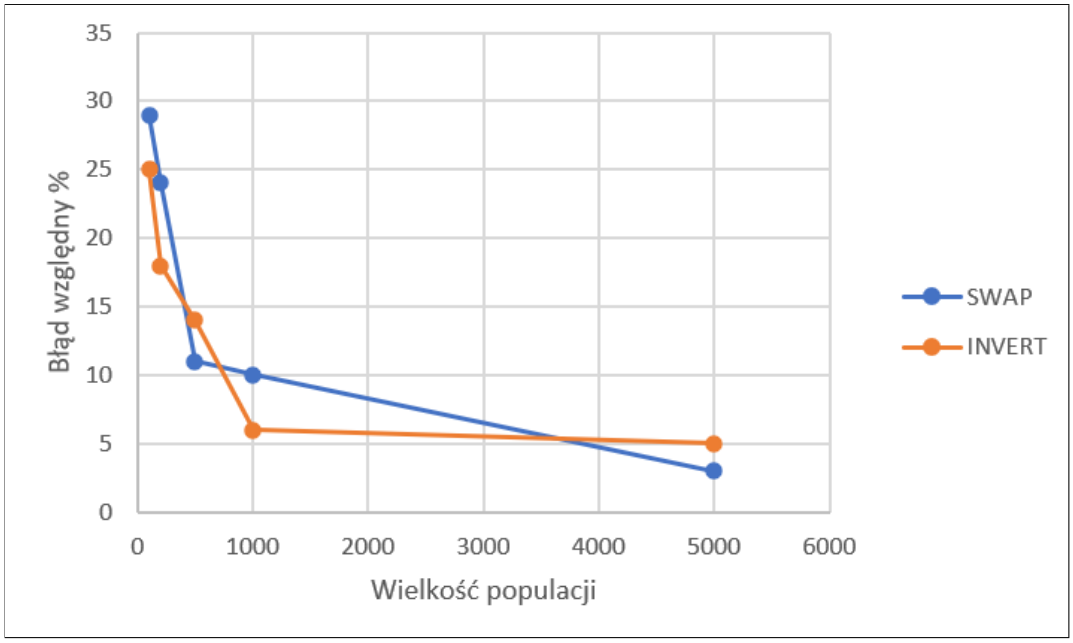


Rysunek 6: Zależność błędu względnego od wielkości populacji i typu mutacji dla 17 miast

4.2 Pomiary dla 55 miast

Wielkość populacji	Czas[s]	Mutacja	
		SWAP	INVERT
100	60	29%	25%
200		24%	18%
500		11%	14%
1000		10%	6%
5000		3%	5%

Tabela 2: Wyniki pomiarów błędu względnego algorytmu genetycznego w zależności od wielkości populacji i typu mutacji dla 55 miast.

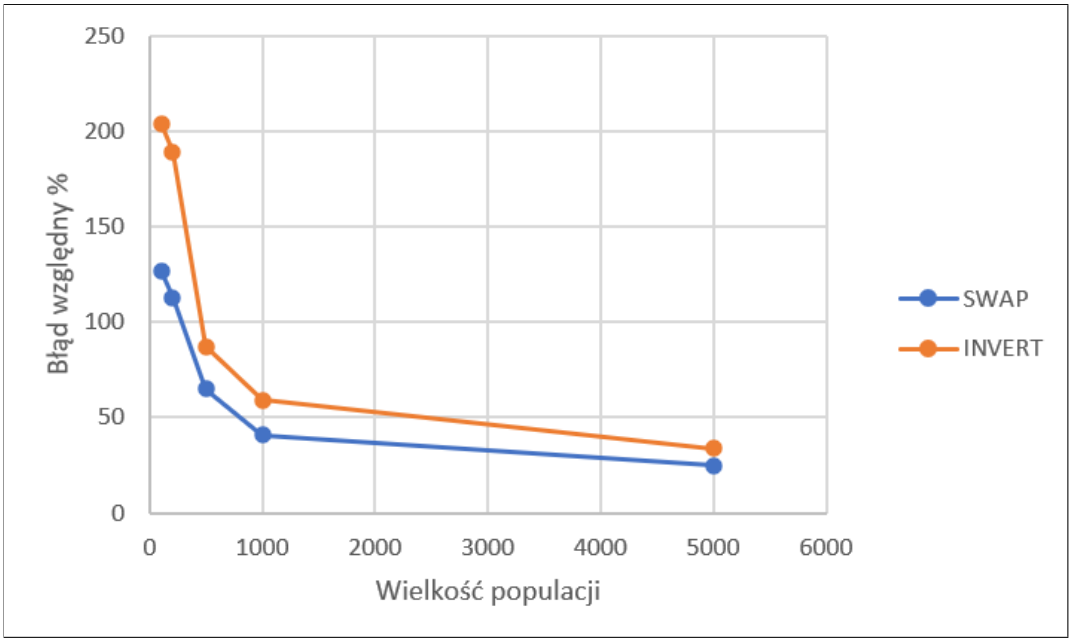


Rysunek 7: Zależność błędu względnego od wielkości populacji i typu mutacji dla 55 miast

4.3 Pomiary dla 170 miast

Wielkość populacji	Czas[s]	Mutacja	
		SWAP	INVERT
100	60	127%	204%
200		113%	189%
500		65%	87%
1000		41%	59%
5000		25%	34%

Tabela 3: Wyniki pomiarów błędu względnego algorytmu genetycznego w zależności od wielkości populacji i typu mutacji dla 170 miast.



Rysunek 8: Zależność błędu względnego od wielkości populacji i typu mutacji dla 170 miast

4.4 Porównanie Tabu Search do Algorytmu Genetycznego

Liczba miast	Tabu Search	Algorytm Genetyczny
17	0%	0%
55	30%	3%
170	133%	25%

Tabela 4: Porównanie najlepszych wyników Tabu Search do Algorytmu Genetycznego

5 Wnioski

Na podstawie otrzymanych wyników można stwierdzić, że efektywność działania algorytmu jest wprost proporcjonalna do wielkości populacji. Z danych można również wywnioskować, że przy większej liczbie miast mutacja typu SWAP daje lepsze wyniki.

Porównując algorytm genetyczny do Tabu Search dla 17 miast TS również dawał wyniki z błędem równym 0%, natomiast przy większej liczbie miast widać przewagę GA, gdzie przy 55 miastach błąd to 3% natomiast TS 30%. Przy 170 miastach przewaga GA również jest widoczna 170% do 25%. Podsumowując algorytm genetyczny cechuje się znacznie lepszą efektywnością niż Tabu Search.