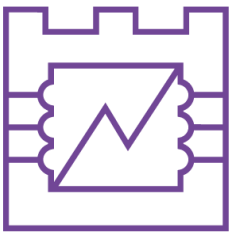


Wydział Inżynierii
Elektrycznej i Komputerowej

POLITECHNIKA KRAKOWSKA

WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ

PROJEKT Z PRZEDMIOTU “Wstęp do programowania w języku C/C++”

“Problem komiwojażera”

Prowadzący przedmiot:

dr inż. S Bąk

Wykonał:

Mateusz Jankowski
Kamil Kujszczyk
Kamil Jagielski

1. Zakres projektu.

Poniższy raport poświęcony będzie omówieniu funkcjonalności i implementacji kodu aplikacji „Problem komiwojażera”.

Lista narzędzi oraz środowisk wykorzystanych do realizacji projektu:

- Visual Studio
- Visual Studio Code
- Code::Blocks
- Github

W ramach zajęć projektowych zrealizowaliśmy następujące kroki:

- Dziedziczenie klas
- Polimorfizm
- Klasy wewnętrzne
- Zapis i odczyt danych przy użyciu plików tekstowych

Celem naszego projektu było napisanie aplikacji, która rozwiązuje problem komiwojażera.

2. Opis problemu.

Historia.

Problem komiwojażera został sformułowany dosyć dawno, bo w 1759 r. przez Eulera (pod nazwą inną, niż używana teraz), który zajmował się rozwiązywaniem zadania ruchu konika po szachownicy. Prawidłowe rozwiązanie problemu opracowywanego przez Eulera wymagało, aby konik odwiedził każde z 64 pól na szachownicy dokładnie raz w czasie całego ruchu.

Termin “komiwojażer” był użyty po raz pierwszy w 1932 r. w książce niemieckiej *“Komiwojażer, jak i co powinien on robić, aby wykonywać zlecenia i mieć powodzenie w interesach”.* Chociaż nie było to głównym tematem książki, zadanie komiwojażera było opisane w ostatnim rozdziale (chodziło o optymalizację kosztów związanych z przebytą trasą).

Problem komiwojażera w obecnym sformułowaniu wprowadziła RAND Corporation w 1948 r. Jej reputacja pomogła w tym, że problem ten, stał się problemem popularnym. Popularność tego problemu wynikała

z powodu opracowania nowej metody programowania liniowego i prób rozwiązywania zadań kombinatorycznych.

Udowodniono, że problem komiwojażera jest NP-trudy, pojawia się on w licznych zastosowaniach i liczba miast może być znacząca.

Dlaczego “problem komiwojażera”?

Komiwojażer (ang. TSP- travelling salesman problem) ma do odwiedzenia pewna liczbę miast. Chciałby dotrzeć do każdego z nich i wrócić do miasta, z którego wyruszył. Dane są również odległości między miastami. Jak powinien zaplanować trasę podróży, aby w sumie przebył możliwie najkrótsza drogę? Przez 'odległość' między miastami możemy rozumieć odległość w kilometrach, czas trwania podróży między tymi miastami albo koszt takiej podróży (na przykład cenę paliwa). W tym ostatnim przypadku, poszukiwanie optymalnej trasy polega na zminimalizowaniu całkowitych kosztów podróży. Tak więc możemy poszukiwać trasy najkrótszej albo najszybszej, albo najtańszej. Zakładamy przy tym, że odległość między dowolnymi dwoma miastami jest nie większa niż długość jakiejkolwiek drogi łączącej te miasta, która wiedzie przez inne miasta.

Założenie to tylko z pozoru wydaje się być zawsze spełnione.

Rozważmy następujący przykład. Załóżmy, że interesuje nas czas trwania podróży koleją. Najszybsze połączenie z Krakowa do Wrocławia wiedzie przez Katowice. Czas trwania tej podróży traktujemy w tym przypadku jako 'odległość' z Krakowa do Wrocławia.

Sformułowanie problemu.

Zbudujmy graf ważony, którego wierzchołki są miastami. Każda parę miast połączmy krawędziami. Każdej krawędzi nadajemy wagę równa 'odległości' między miastami odpowiadającymi wierzchołkom, które są końcami tej krawędzi. Otrzymujemy w ten sposób graf pełny, który ma tyle wierzchołków, ile miast musi odwiedzić komiwojażer (wliczając w to miasto, z którego wyrusza (tak jak na Rysunku 1). Odwiedzenie wszystkich miast odpowiada cyklowi, który przechodzi przez każdy wierzchołek danego grafu dokładnie raz. Cykl taki nazywamy cyklem Hamiltona. Poszukujemy więc w grafie pełnym cyklu Hamiltona o minimalnej sumie wag krawędzi.

Cykl Hamiltona to taki cykl w grafie, w którym każdy wierzchołek grafu odwiedzany jest dokładnie raz (oprócz pierwszego wierzchołka).

Rys 1. Przykład rozwiązanego problemu komiwojażera dla polskich miast.

[https://www.deltami.edu.pl/temat/informatyka/algorytmy/2020/01/27/Problem-komiwojazera-w-praktyce/]

Czy rozwiązanie problemu komiwojażera zawsze istnieje?

Tak, ponieważ dowolny graf pełny posiada co najmniej jeden cykl Hamitlona. Ponieważ każdy graf posiada skończoną liczbę wierzchołków, to w zbiorze cykli Hamiltona istnieje taki, który posiada minimalną sumę wag krawędzi.

3. Teoria.

Algorytm genetyczny

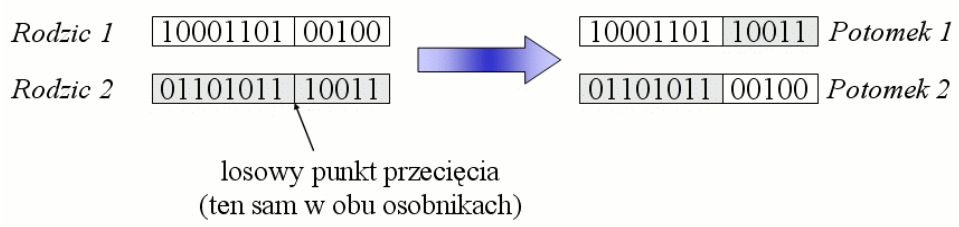
Jest to rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania najlepszych rozwiązań.

W każdym cyklu (każde pokolenie) poddawane są "obróbce" za pomocą operatorów ewolucyjnych. Celem tego etapu jest wygenerowanie nowego pokolenia, na podstawie poprzedniego, które być może będzie lepiej dopasowane do założonego środowiska.

Operator krzyżowania ma za zadanie łączyć w różnych kombinacjach cechy pochodzące z różnych osobników populacji, zaś **operator mutacji** ma za zadanie zwiększać różnorodność tych osobników.

Krzyżowanie

Krzyżowanie polega na połączeniu niektórych (wybierane losowo) genotypów w jeden (jak na rysunku 2). Kojarzenie ma sprawić, że potomek dwóch osobników rodzicielskich ma zespół cech, który jest kombinacją ich cech (może się zdarzyć, że tych najlepszych).



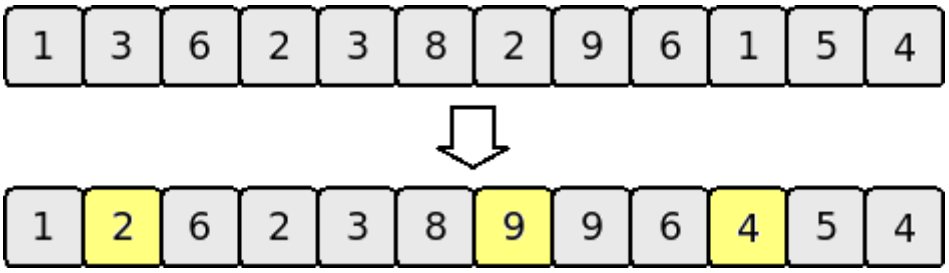
Rys. 2 Przykład krzyżowania.

[http://edu.pjwstk.edu.pl/wyklady/nai/scb/rW10.htm]

Mutacja

Mutacja wprowadza do genotypu losowe zmiany (jak na rysunku 3). Jej zadaniem jest wprowadzanie różnorodności w populacji, czyli zapobieganie (przynajmniej częściowe) przedwczesnej zbieżności algorytmu.

Mutacja zachodzi z pewnym przyjętym prawdopodobieństwem. Jest ono niskie, ponieważ zbyt silna mutacja przynosi efekt odwrotny do zamierzonego: zamiast subtelnie różnicować dobre rozwiązania - niszczy je.



Rys. 3 Przykład mutacji.

[https://kcir.pwr.edu.pl/~witold/aiarr/2008_projekty/plany/]

Elityzm

Elityzm jest to grono najlepszych osobników z danej generacji, które zostanie zastosowane w kolejnej generacji w celu uzyskania optymalnego rozwiązania. Grono to zależy od wartości wprowadzonej przez użytkownika.

4. Budowa wewnętrzna algorytmu.

Nasz program będzie używał algorytmu genetycznego w celu rozwiązania problemu komiwojażera.

Po uruchomieniu programu wyskoczy komunikat o wprowadzeniu następujących danych:

- Liczebność populacji (popSize) - wielkość populacji.
- Liczebność najlepszych osobników (eliteSize) - zbiór najlepszych osobników z każdej populacji.
- Częstotliwość mutacji (mutationRate) - współczynnik mutacji.
- Liczebność generacji (generations) - ilość generacji.
- Ilość miast (numberOfCities) - ilość miast.

(Różne wartości zmieniają zarówno wynik końcowy algorytmu (dystans do przebycia), ale również czas wykonywania algorytmu.)

W przypadku wpisania błędnych danych, np. liter, program pokaże komunikat “Błędne dane! Spróbuj jeszcze raz.”.

W przypadku niewpisania żadnych danych użyte zostaną dane domyślne, które są podane koło komunikatów.

Po uzyskaniu tych danych od użytkownika, program zapyta się, czy współrzędne miast mają być pobierane:

1. Losowo.
2. Z konsoli.
3. Z zewnętrznego pliku tekstowego.

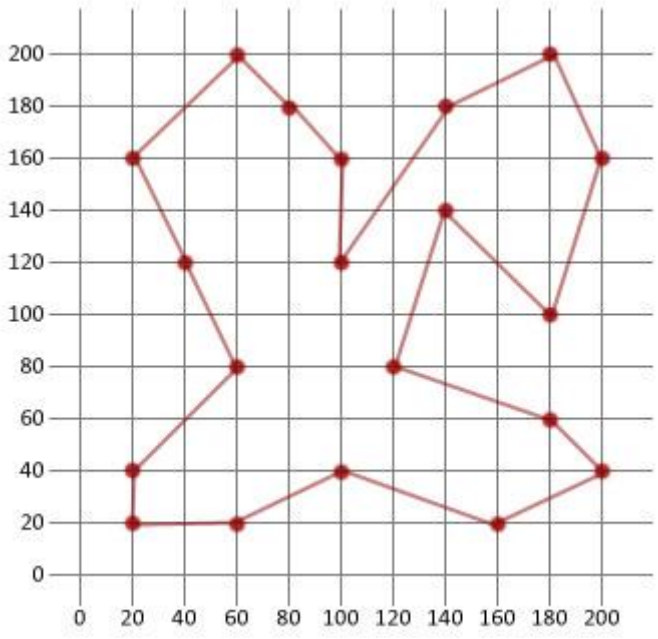
(W 1 i 3 przypadku, współrzędne trzeba zapisywać w formacie X Y)

Kolejnym krokiem będzie rozpoczęcie generowania optymalnej trasy.
W trakcie generowania, wypisywana będzie ostatnia najlepsza droga, aż do wykonania zadanej liczby generacji.

W wyjściowym, zewnętrznym pliku tekstowym będą zapisywane następujące dane:

- Pierwotna długość trasy.
- Długość trasy po optymalizacji.
- Czas realizacji programu.
- Współrzędne miast (w kolejności, która gwarantuje najkrótszą trasę dla danych parametrów).
- Jakość wykonanej optymalizacji (w procentach zostanie pokazany wynik, o jaki udało się zoptymalizować naszą trasę).

Współrzędne miast są przedstawione w postaci (x,y) w celu łatwiejszej realizacji na układzie współrzędnych.



Rys. Przykładowa realizacja współrzędnych miast na układzie współrzędnych.

[https://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5]

Plik Library.cpp

```cpp
vector<Fitness::City> createRoute(const vector<Fitness::City>& cityList) {
        vector<Fitness::City> route = cityList;
        random_device rd;
        mt19937 generator(rd());
        shuffle(route.begin(), route.end(), generator);
        return route;
}
```

W tym kodzie funkcja createRoute tworzy losową trasę dla miast reprezentowanych przez obiekty klasy Fitness::City. Przyjmuje wektor miast cityList, tworzy kopię tego wektora, a następnie losowo miesza ich kolejność, zwracając tak utworzoną trasę. Funkcja korzysta z generatora liczb pseudolosowych.

```cpp
vector<vector<Fitness::City>> initialPopulation(int popSize, const vector<Fitness::City>& cityList) {
        vector<vector<Fitness::City>> population;

        for (int i = 0; i < popSize; i++) {
                population.push_back(createRoute(cityList));
        }

        return population;
}
```

W tej funkcji initialPopulation tworzona jest początkowa populacja tras dla miast reprezentowanych przez obiekty klasy Fitness::City. Przyjmuje rozmiar populacji (popSize) i wektor miast cityList. Następnie w pętli dodawane są losowe trasy do populacji poprzez wywołanie funkcji createRoute dla każdego osobnika. Funkcja zwraca wektor wektorów, reprezentujący początkową populację.

```cpp
vector<pair<int, double>> rankRoutes(const vector<vector<Fitness::City>>& population) {
        map<int, double> fitnessResults;

        for (int i = 0; i < population.size(); i++) {
                fitnessResults[i] = Fitness(population[i]).routeFitness();
        }

        vector<pair<int, double>> sortedResults;
        copy(fitnessResults.begin(), fitnessResults.end(), back_inserter(sortedResults));

        sort(sortedResults.begin(), sortedResults.end(),
                [](const pair<int, double>& a, const pair<int, double>& b) {
                        return a.second > b.second;
                });

        return sortedResults;
}
```

W tej funkcji rankRoutes oceniane są trasy w populacji reprezentowanej przez wektor wektorów miast (population). Dla każdej trasy obliczana jest wartość fitness przy użyciu obiektu Fitness i funkcji routeFitness. Następnie wyniki fitness są przechowywane w mapie, gdzie kluczem jest indeks trasy w populacji, a wartością jest jej fitness.

Ostatecznie, wyniki fitness są sortowane malejąco, a pary (indeks trasy, fitness) są zwracane jako wektor w formie posortowanej listy rankingowej.

```
vector<int> selection(const vector<pair<int, double>>& popRanked, int eliteSize) {
        vector<int> selectionResults;
        vector<double> cumPerc;
        double fitnessSum = 0.0;

        for (const auto& ranked : popRanked) {
                fitnessSum += ranked.second;
        }

        double cumSum = 0.0;
        for (const auto& ranked : popRanked) {
                cumSum += ranked.second;
                cumPerc.push_back(100.0 * cumSum / fitnessSum);
        }

        for (int i = 0; i < eliteSize; i++) {
                selectionResults.push_back(popRanked[i].first);
        }

        random_device rd;
        mt19937 generator(rd());
        // Czemu selekcja przebiega w ten sposób?
        for (int i = 0; i < popRanked.size() - eliteSize; i++) {
                double pick = 100.0 * generate_canonical<double, 10>(generator);        //czym jest pick
                for (int j = 0; j < popRanked.size(); j++) {
                        if (pick <= cumPerc[j]) {
                                selectionResults.push_back(popRanked[j].first);
                                break;
                        }
                }
        }


        return selectionResults;
}
```

W tej funkcji selection dokonywana jest selekcja tras z populacji na podstawie ich ocen fitness. Parametr eliteSize określa liczbę najlepszych tras, które są zachowywane bez zmian.

1.Obliczane są procentowe wartości kumulatywne fitness dla każdej trasy w populacji.

2.Wybierane są najlepsze trasy (elite) na podstawie ich rankingów.

3.Następnie, dla pozostałych tras, losowane są wartości (pick) w przedziale od 0 do 100 procent fitnessSum.

4.Wybierane są trasy na podstawie wartości pick i procentowych wartości kumulatywnych. Im wyższy fitness, tym większa szansa na wybranie danej trasy.

```
vector<vector<Fitness::City>> matingPool(const vector<vector<Fitness::City>>& population, const vector<int>& selectionResults) {
        vector<vector<Fitness::City>> matingpool;

        for (int i = 0; i < selectionResults.size(); i++) {
                int index = selectionResults[i];
                matingpool.push_back(population[index]);
        }

        return matingpool;
}
```

Funkcja tworzy nowy wektor tras (matingpool), dodając do niego trasy z populacji na podstawie indeksów uzyskanych z wyników selekcji. Ostatecznie zwraca tą pule reprodukcyjną, która będzie używana do krzyżowania tras w procesie ewolucji algorytmu genetycznego.

```
vector<Fitness::City> breed(const vector<Fitness::City>& parent1, const vector<Fitness::City>& parent2) {
        vector<Fitness::City> child;
        vector<Fitness::City> childP1;
        vector<Fitness::City> childP2;

        int geneA = static_cast<int>(rand() % parent1.size());
        int geneB = static_cast<int>(rand() % parent1.size());

        int startGene = min(geneA, geneB);
        int endGene = max(geneA, geneB);

        for (int i = startGene; i < endGene; i++) {
                childP1.push_back(parent1[i]);
        }

        for (const auto& item : parent2) {
                if (find(childP1.begin(), childP1.end(), item) == childP1.end()) {
                        childP2.push_back(item);
                }
        }

        child.insert(child.end(), childP1.begin(), childP1.end());
        child.insert(child.end(), childP2.begin(), childP2.end());

        return child;
}
```

W funkcji breed implementowana jest operacja krzyżowania (crossover) między dwoma rodzicami reprezentowanymi przez wektory miast parent1 i parent2. Krzyżowanie polega na losowym wyborze pewnego zakresu genów od jednego rodzica i uzupełnieniu ich genami drugiego rodzica, aby utworzyć potomka (child).

1. Losowane są indeksy geneA i geneB w zakresie od 0 do rozmiaru rodzica.
2. Określany jest zakres genów do skrzyżowania, gdzie startGene i endGene są minimalnym i maksymalnym indeksem spośród geneA i geneB.
3. Geny z wybranego zakresu (childP1) przekazywane są do potomka.
4. Pozostałe geny z drugiego rodzica (parent2), które nie znajdują się jeszcze w potomku, są kopiowane do childP2.
5. Potomkowi przypisywane są geny z childP1 i childP2.
6. Ostatecznie, utworzony potomek (child) jest zwracany.

```cpp
vector<vector<Fitness::City>> breedPopulation(const vector<vector<Fitness::City>>& matingpool, int eliteSize) {
        vector<vector<Fitness::City>> children;
        int length = matingpool.size() - eliteSize;
        vector<vector<Fitness::City>> pool = matingpool;
        random_device rd;
        mt19937 generator(rd());
        shuffle(pool.begin(), pool.end(), generator);

        for (int i = 0; i < eliteSize; i++) {
                children.push_back(matingpool[i]);
        }

        for (int i = 0; i < length; i++) {
                vector<Fitness::City> child = breed(pool[i], pool[matingpool.size() - i - 1]);
                children.push_back(child);
        }

        return children;
}
```

W funkcji breedPopulation tworzona jest nowa populacja potomków na podstawie puli reprodukcyjnej (matingpool). Parametr eliteSize określa liczbę najlepszych tras, które są zachowywane bez zmian.

```cpp
vector<Fitness::City> mutate(const vector<Fitness::City>& individual, double mutationRate) {
        vector<Fitness::City> mutatedIndividual = individual;
        int Method = rand() % 2;


        if (Method == 0) {
                for (int swapped = 0; swapped < individual.size(); swapped++) {

                        if (rand() < mutationRate) {                                    //Czemu nie randomowe double by*o by częściej tak to ma 1/ INT_MAX szans
                                int swapWith = rand() % individual.size();

                                Fitness::City city1 = mutatedIndividual[swapped];
                                Fitness::City city2 = mutatedIndividual[swapWith];

                                mutatedIndividual[swapped] = city2;
                                mutatedIndividual[swapWith] = city1;
                        }
                }
        }
        else {
                if (rand() < mutationRate) {                                    //Czemu nie randomowe double by*o by częściej tak to ma 1/ INT_MAX szans

                        int subVectorSize = 1 + rand() % (individual.size() / 2);
                        int subVectorStart = rand() % (individual.size() - subVectorSize + 1);
                        vector<Fitness::City> subvector(mutatedIndividual.begin() + subVectorStart, mutatedIndividual.begin() + subVectorStart + subVectorSize);
                        int swapWith = rand() % (individual.size() - subVectorSize + 1);
                        mutatedIndividual.erase(mutatedIndividual.begin() + subVectorStart, mutatedIndividual.begin() + subVectorStart + subVectorSize);
                        mutatedIndividual.insert(mutatedIndividual.begin() + swapWith, subvector.begin(), subvector.end());

                }
        }

        return mutatedIndividual;
}
```

W funkcji mutate dokonywane są mutacje w pojedynczym osobniku reprezentowanym przez wektor miast (individual). Parametr mutationRate określa prawdopodobieństwo mutacji dla każdego genu w osobniku.

```cpp
vector<vector<Fitness::City>> mutatePopulation(const vector<vector<Fitness::City>>& population, double mutationRate) {
        vector<vector<Fitness::City>> mutatedPop;

        for (int ind = 0; ind < population.size(); ind++) {
                vector<Fitness::City> mutatedInd = mutate(population[ind], mutationRate);
                mutatedPop.push_back(mutatedInd);
        }

        return mutatedPop;
}
```

W funkcji mutatePopulation dokonywana jest mutacja całej populacji tras. Dla każdego osobnika w populacji (reprezentowanego przez wektor miast), wywoływana jest funkcja mutate, która zwraca zmutowany osobnik. Zmutowane osobniki są następnie dodawane do nowej populacji

(mutatedPop), która jest zwracana na koniec funkcji. W ten sposób, cała populacja zostaje poddana mutacji, co jest jednym z kroków w procesie ewolucji algorytmu genetycznego.

```
vector<vector<Fitness::City>> nextGeneration(const vector<vector<Fitness::City>>& currentGen, int eliteSize, double mutationRate) {
        vector<pair<int, double>> popRanked = rankRoutes(currentGen);
        vector<int> selectionResults = selection(popRanked, eliteSize);
        vector<vector<Fitness::City>> matingpool = matingPool(currentGen, selectionResults);
        vector<vector<Fitness::City>> children = breedPopulation(matingpool, eliteSize);
        vector<vector<Fitness::City>> nextGeneration = mutatePopulation(children, mutationRate);
        return nextGeneration;
}
```

W funkcji nextGeneration implementowany jest kolejny krok algorytmu genetycznego, mającego na celu wygenerowanie następnej generacji tras.

Populacja tras z obecnej generacji jest oceniana, a wyniki są sortowane według fitnessu.

Wybierane są najlepsze trasy (elita) na podstawie wyników rankingowych.

Tworzony jest basen reprodukcyjny (mating pool) na podstawie wyników selekcji.

Trasy w basenie reprodukcyjnym są krzyżowane, tworząc nowe trasy potomne.

Następuje mutacja w populacji potomnej.

Zmutowana populacja potomna jest zwracana jako kolejna generacja.

```
vector<Fitness::City> geneticAlgorithm(const vector<Fitness::City>& population, int popSize, int eliteSize, double mutationRate, int generations) {
        vector<vector<Fitness::City>> pop = initialPopulation(popSize, population);
        double previousDistance = 1 / rankRoutes(pop)[0].second;
        double currentDistance{};
        cout << "Initial route: " << previousDistance << endl;
        cout << "Press any key to initialize the algorithm...\n";
        _getch();

        auto start = chrono::high_resolution_clock::now();
        for (int i = 0; i < generations; i++) {
                pop = nextGeneration(pop, eliteSize, mutationRate);
                currentDistance = 1 / rankRoutes(pop)[0].second;
                if (currentDistance < previousDistance) {
                        cout << "\t" << currentDistance << endl;
                        previousDistance = currentDistance;
                }
        }
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        cout << "Duration: " << duration.count() << "s" << endl;
        cout << "Final shortest route found: " << currentDistance << endl;
        int bestRouteIndex = rankRoutes(pop)[0].first;
        vector<Fitness::City> bestRoute = pop[bestRouteIndex];
        return bestRoute;
}
```

W funkcji geneticAlgorithm implementowany jest pełny algorytm genetyczny dla rozwiązania problemu komiwojażera. Funkcja przyjmuje początkową populację tras (population), rozmiar populacji (popSize), rozmiar elity (eliteSize), współczynnik mutacji (mutationRate), oraz liczbę generacji (generations).

1. Inicjalizowana jest początkowa populacja tras.
2. Obliczana jest odległość dla najlepszej trasy z początkowej populacji.
3. Wyświetlane są informacje dotyczące początkowej trasy.
4. Rozpoczęcie algorytmu po naciśnięciu dowolnego klawisza.

5. Pętla ewolucji algorytmu, w której kolejno następuje generowanie następnej populacji, obliczanie odległości dla najlepszej trasy oraz wyświetlanie postępu.
6. Po zakończeniu pętli wyświetlane są informacje o czasie trwania i znalezionej optymalnej trasie.

```
vector<Fitness::City> readCitiesFromFile(const string& filename) {
	vector<Fitness::City> cityList;
	ifstream inputFile("cities/" + filename + ".txt");
	if (!inputFile.is_open()) {
		throw runtime_error("Couldn't open the file: " + filename);
		_getch();
	}

	string line;
	int lineNumber = 1;
	while (getline(inputFile, line)) {
		istringstream iss(line);
		int x, y;
		if (!(iss >> x >> y)) {
			throw runtime_error("Incorrect data in row number " + to_string(lineNumber) + ": " + line);
			_getch();
		}
		cityList.push_back({ x, y });
		lineNumber++;
	}

	inputFile.close();
	return cityList;
}
```

Funkcja readCitiesFromFile wczytuje dane dotyczące miast z pliku o podanej nazwie. Plik powinien zawierać pary liczb całkowitych, reprezentujących współrzędne x i y miast. Funkcja zwraca wektor obiektów klasy Fitness::City, które reprezentują wczytane miasta.

Plik Ftitness.cpp

```
Fitness::Fitness(const std::vector<City>& route) {
    this->route = route;
    this->distance = 0.0;
    this->fitness = 0.0;
}
```

Ten fragment kodu przedstawia definicję konstruktora klasy Fitness, który przyjmuje wektor miast (route) jako argument.

```
double Fitness::routeDistance() {
    if (distance == 0.0) {
        double pathDistance = 0.0;

        for (size_t i = 0; i < route.size(); i++) {
            //Polimorfizm
            Location* fromCity = new City(route[i].getX(), route[i].getY());
            Location* toCity = (i + 1 < route.size()) ? new City(route[i + 1].getX(), route[i + 1].getY())
                : new City(route[0].getX(), route[0].getY());
            //City& fromCity = route[i];
            //City& toCity = (i + 1 < route.size()) ? route[i + 1] : route[0];
            pathDistance += fromCity->distance(toCity);     //Wywo*anie polimorficzne
        }
        distance = pathDistance;
    }
    return distance;
}
```

Funkcja routeDistance w klasie Fitness służy do obliczania odległości dla danej trasy.

```
double Fitness::routeFitness() {
    if (fitness == 0.0) {
        fitness = 1.0 / routeDistance();
    }
    return fitness;
}
```

Funkcja routeFitness w klasie Fitness służy do obliczania wartości fitness dla danej trasy.

```
Fitness::City::City(int x, int y) : x(x), y(y) {}
```

Ten konstruktor inicjalizuje obiekt klasy City na podstawie przekazanych współrzędnych x i y. Wartości te są przypisywane do odpowiednich pól obiektu.

```
bool Fitness::City::operator==(const City& other) const {
    return (x == other.x) && (y == other.y);
}
```

Ten operator porównuje dwa obiekty klasy City pod kątem równości. Zwraca true, jeśli współrzędne x i y obiektu są równe odpowiadającym współrzędnym obiektu przekazanego jako other. Operator jest oznaczony

jako const, co oznacza, że nie modyfikuje stanu obiektu, na którym jest wywoływany.

```
double Fitness::City::distance(const Location* location) {
    const City* city = dynamic_cast<const City*>(location);
    int xDis = abs(this->x - city->x);
    int yDis = abs(this->y - city->y);
    double distance = sqrt(pow(xDis, 2) + pow(yDis, 2));
    return distance;
}


std::ostream& operator<<(std::ostream& os, const Fitness::City& city) {
    os << "(" << city.x << "," << city.y << ")";
    return os;
}
```

Funkcja distance w klasie City służy do obliczania odległości między dwoma miastami (obiektami typu City).

```
int Fitness::City::getX(){
    return this->x;
}

int Fitness::City::getY(){
    return this->y;
}
```

getX: Zwraca wartość współrzędnej x dla danego miasta.

getY: Zwraca wartość współrzędnej y dla danego miasta.

Plik Main.cpp

```cpp
#include "Library.h"
#include "Fitness.h"

using namespace std;

template<typename T>
void getInput(T& variable, T defaultValue) {
	if (is_same<T, int>::value || is_same<T, double>::value)

		while (true) {

			if (cin.peek() == '\n') {
				variable = defaultValue;
				cin.clear();
				cin.ignore(numeric_limits<streamsize>::max(), '\n');
				break;
			}
			else if (!(cin >> variable)) {
				cout << "Incorrect input! Try again...\n";
				cin.clear();
				cin.ignore(numeric_limits<streamsize>::max(), '\n');
				continue;
			}
			else {
				cin.clear();
				cin.ignore(numeric_limits<streamsize>::max(), '\n');
				break;
			}
		}
}
```

Funkcja ma na celu zapewnić bezpieczne wczytywanie danych z wejścia dla zmiennych liczbowych z obsługą błędów, tak aby uniknąć nieprzewidzianych problemów związanych z wprowadzaniem danych przez użytkownika.

```
while (true) {

        vector<Fitness::City> cityList;
        int popSize, eliteSize, generations, numberOfCities{}, inputMethod, inputX{}, inputY{};
        string fileName;
        bool validInput = false;
        double mutationRate{};

        random_device rd;
        mt19937 generator(rd());
        uniform_int_distribution<mt19937::result_type> dist(100, 800);

        cout << "Traveling Salesman Problem\n\n";

        cout << "Enter the following data:\n";

        cout << "Population count [default = 100]:\t";
        getInput(popSize, 100);
        cout << "Elite population count [default = 20]:\t";
        getInput(eliteSize, 20);
        cout << "Number of generations [default = 500]:\t";
        getInput(generations, 500);
        cout << "Mutation frequency [default = 0.1]:\t";                    //0.1 to jest 10%
        getInput(mutationRate, 0.1);

        cout << "\nChoose your preferred way of generating cities:\n";
        cout << "1.Randomized points\n2.From the terminal\n3.From a file\n";
        cin >> inputMethod;
        if (inputMethod == 1 || inputMethod == 2) {
                cout << "Cities count [default = 100]:\t";
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                getInput<int>(numberOfCities, 100);
        }
```

W tym fragmencie kodu użytkownikowi zadawane są pytania dotyczące parametrów algorytmu genetycznego oraz wyboru sposobu generowania miast

```
while (!validInput)
        switch (inputMethod) {
        case 1:

                for (int i = 0; i < numberOfCities; i++) {
                        int x = static_cast<int>(dist(generator));
                        int y = static_cast<int>(dist(generator));
                        cityList.push_back(Fitness::City(x, y));
                }
                validInput = !validInput;
                break;
```

W przypadku wybrania opcji “1” miasta będą generować się losowo.

```
case 2:
        cout << "Enter the coordinates for " << numberOfCities << " cities in the following format: <X> <Y>.\n";
        for (int i = 0; i < numberOfCities; i++) {
                cout << "Enter the coordinates of citi number " << i + 1 << ":\t";
                cin >> inputX >> inputY;
                if (cin) {
                        int x = static_cast<int>(inputX);
                        int y = static_cast<int>(inputY);
                        cityList.push_back(Fitness::City(x, y));
                }
                else {
                        cout << "Incorrect input! Try again...\n";
                        cin.clear();
                        cin.ignore(numeric_limits<streamsize>::max(), '\n');
                        i--;
                }
        }
        validInput = !validInput;
        break;
```

W przypadku wybrania opcji “2” miasta będą wpisane przez użytkownika.

```
case 3:
        cout << "Enter the name of the file without '.txt' extension: ";
        cin >> fileName;
        try {
                cityList = readCitiesFromFile(fileName);
        }
        catch (const exception& e) {
                cerr << "Error: " << e.what() << endl;
                _getch();
                return 1;
        }
        validInput = !validInput;
        break;
default:
        cout << "Incorrect choice. Try again...\n";
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cin >> inputMethod;
        break;
}
```

W przypadku wybrania opcji “3” miasta będą sczytane z pliku .txt.

```
    vector<Fitness::City> bestRoute = geneticAlgorithm(cityList, popSize, eliteSize, mutationRate, generations);

    cout << "The shortest found route goes through the following cities:\n";
    for (const Fitness::City city : bestRoute) {
        cout << city << ", ";
    }
    cout << endl;

    ofstream outputFile("bestRouteCities.txt");
    if (outputFile.is_open()) {
        outputFile << "The shortest found route goes through the following cities:\n";
        for (const auto& city : bestRoute) {
            outputFile << city << "\n";
        }
        outputFile.close();
        cout << "Results written to bestRouteCities.txt" << endl;
    }
    else {
        cout << "Couldn't open the file" << endl;
    }

    cout << "Press ESC key to exit the program ...";
    if (_getch() == 27)
    {
        exit(0);
    }

    system("cls");

}
return 0;
```

Ten fragment kodu kończy program po zakończeniu algorytmu genetycznego dla problemu komiwojażera. Poniżej znajduje się krótka analiza:

- Wywołanie funkcji geneticAlgorithm: Algorytm genetyczny jest uruchamiany dla listy miast (cityList) i zdefiniowanych parametrów, takich jak rozmiar populacji (popSize), rozmiar elity (eliteSize), częstość mutacji (mutationRate) i liczba generacji (generations). Wyniki są zapisywane w wektorze bestRoute.
- Wypisanie trasy do konsoli: Trasa o najkrótszej długości jest wypisywana na standardowe wyjście za pomocą pętli for.
- Zapisanie trasy do pliku: Trasa o najkrótszej długości jest również zapisywana do pliku o nazwie "bestRouteCities.txt". Jeśli plik zostanie otwarty poprawnie, trasa jest zapisywana w nowych liniach.
- Komunikat o zapisaniu wyników: Jeśli zapis do pliku powiedzie się, program wypisuje komunikat "Results written to bestRouteCities.txt".

5. Pseudokod.

1. Uruchom program, wprowadź parametry trasy

   - Liczebność populacji
   - Liczebność najlepszych osobników
   - Częstotliwość mutacji
   - Liczebność generacji
   - Ilość miast

2. Zapytaj użytkownika o sposób wprowadzenia danych:

   A) Losowo
   B) W konsoli
   C) Z pliku

3. Inicjalizuj trasę początkową:

   - Dla podanej ilości miast wygeneruj początkową trasę (nie jest optymalna)

4.Powtarzaj do momentu wykonania wszystkich generacji:

   - Szukaj lepszej trasy w każdej generacji i dla każdej jednostki w populacji (weź pod uwagę współczynnik krzyżowania oraz mutacji),
   - Po dokonaniu generacji, zostaw wprowadzoną wcześniej liczbę najlepszy rozwiązań.

5. Zwróć najlepsze znalezione rozwiązanie (najlepszą trasę).

6. Oblicz jakość optymalizacji.

7. Wypisz do pliku:

   - Trasę początkową.
   - Najlepszą znalezioną trasę.
   - Współrzędne miast (w kolejności gwarantującej najlepszą trasę)
   - Jakość optymalizacji.
   - Czas realizacji programu.

8. Zakończ program

6. Wyniki testów.

1. (Rozwiązanie dla danych generowanych losowo – 2 różne przykłady).

1.1

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]: 300

Liczebnosc najlepszych osobnikow [default = 20]: 10

Ilosc generacji [default = 500]: 1500

Czestotliwosc mutacji [default = 0.1]:

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

1

Ilosc miast [default = 100]: 40

Wstepna droga: 10194.8

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 9810.88

[…]

ND: 4465.75

Czas trwania generacji: 21.4986s

Ostateczna najkrotsza znaleziona droga: 4465.75

Jakosc optymalizacji: 62.9285%

(249,455), (165,339), (168,229), (183,191), (335,234), (334,174), (362,188), (352,106), (496,119), (484,212), (452,232), (394,204), (305,367), (314,393), (341,399), (464,392), (432,420), (447,506), (437,479), (610,453), (580,542), (505,747), (611,684), (664,750), (759,789), (683,732), (717,646), (770,641), (735,652), (710,513), (717,456), (731,366), (772,159), (770,423), (797,477), (760,477), (754,499), (122,727), (131,694), (164,473)

1.2

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]:

Liczebnosc najlepszych osobnikow [default = 20]:

Ilosc generacji [default = 500]:

Czestotliwosc mutacji [default = 0.1]:

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

1

Ilosc miast [default = 100]:

Wstepna droga: 32661.4

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 30501.4

[…]

ND: 11742.5

Czas trwania generacji: 4.29797s

Ostateczna najkrotsza znaleziona droga: 11742.5

Jakosc optymalizacji: 64.0477%

(455,280), (418,317), (423,356), (378,431), (453,457), (528,618), (610,727), (650,469), (689,118), (513,132), (509,250), (488,232), (445,211), (322,140), (394,187), (349,282), (437,253), (295,300), (238,234), (261,160), (256,121), (151,209), (153,146), (161,152), (215,350), (160,283), (154,395), (121,312), (141,372), (175,317), (324,335), (321,347), (206,247), (257,315), (205,344), (240,429), (147,565), (272,570), (234,626), (293,574), (404,584), (527,535), (739,640), (728,670), (689,682), (649,739), (516,693), (546,668), (743,608), (741,744), (659,556), (627,742), (561,730), (286,780), (398,755), (256,726), (317,557), (317,630), (334,755), (181,697), (214,465), (336,519), (308,560), (145,557), (252,735), (478,544), (400,444), (548,610), (492,695), (585,760), (727,779), (707,796), (714,512), (722,564), (572,545), (748,542), (687,366), (788,414), (664,443), (685,536), (721,403), (781,282), (768,273), (684,221), (724,123), (682,354), (636,378), (570,418), (630,398), (644,350), (645,278), (585,249), (599,275), (522,146), (465,157), (482,131), (552,297), (557,351), (560,315), (389,259)

2. (Rozwiązanie dla danych wpisywanych z konsoli – 2 różne przykłady).

2.1

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]: 300

Liczebnosc najlepszych osobnikow [default = 20]: 10

Ilosc generacji [default = 500]: 1500

Czestotliwosc mutacji [default = 0.1]: 0.2

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

2

Ilosc miast [default = 100]: 20

Podaj wspolrzedne dla 20 miast w formacie: <X> <Y>.

Podaj wspolrzedne miasta nr1: 123 456

[…]

Podaj wspolrzedne miasta nr20: 3 4

Wstepna droga: 7781.2

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 7225.22

[…]

ND: 3602.38

Czas trwania generacji: 14.5364s

Ostateczna najkrotsza znaleziona droga: 3602.38

Jakosc optymalizacji: 53.704%

(136,699), (213,777), (365,752), (582,792), (997,998), (777,569), (690,420), (848,241), (639,179), (400,300), (98,26), (3,4), (1,2), (2,5), (100,200), (111,222), (158,458), (123,456), (123,464), (112,555),

2.2

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]:

Liczebnosc najlepszych osobnikow [default = 20]:

Ilosc generacji [default = 500]:

Czestotliwosc mutacji [default = 0.1]:

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

2

Ilosc miast [default = 100]: 20

Podaj wspolrzedne dla 20 miast w formacie: <X> <Y>.

Podaj wspolrzedne miasta nr1: 123 456

[..]

Podaj wspolrzedne miasta nr20: 3 4

Wstepna droga: 8347.42

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 7970.92

[…]

ND: 3602.38

Czas trwania generacji: 1.50481s

Ostateczna najkrotsza znaleziona droga: 3602.38

Jakosc optymalizacji: 56.8444%

(400,300), (98,26), (3,4), (1,2), (2,5), (100,200), (111,222), (158,458), (123,456), (123,464), (112,555), (136,699), (213,777), (365,752), (582,792), (997,998), (777,569), (690,420), (848,241), (639,179)

3. (Rozwiązanie dla danych pobieranych z pliku tekstowego – 2 różne przykłady).

3.1

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]: 300

Liczebnosc najlepszych osobnikow [default = 20]: 10

Ilosc generacji [default = 500]: 1500

Czestotliwosc mutacji [default = 0.1]: 0.2

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

3

Podaj nazwe pliku:test.txt

Wstepna droga: 8106.87

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 7791.24

[…]

ND: 3602.38

Czas trwania generacji: 16.2446s

Ostateczna najkrotsza znaleziona droga: 3602.38

Jakosc optymalizacji: 55.5639%

(400,300), (98,26), (3,4), (1,2), (2,5), (100,200), (111,222), (158,458), (123,456), (123,464), (112,555), (136,699), (213,777), (365,752), (582,792), (997,998), (777,569), (690,420), (848,241), (639,179)

3.2

Program rozwiazujacy problem komiwojazera.

Podaj nastepujace dane:

Liczebnosc populacji [default = 100]:

Liczebnosc najlepszych osobnikow [default = 20]:

Ilosc generacji [default = 500]:

Czestotliwosc mutacji [default = 0.1]:

Wybierz metode wprowadzania danych:

1.Losowe punkty

2.Z konsoli

3.Z pliku

3

Podaj nazwe pliku:test.txt

Wstepna droga: 8421.06

Nacisnij dowolny przycisk, aby przyjsc do generacji

ND: 8370.99

[…]

ND: 3602.38

Czas trwania generacji: 1.70366s

Ostateczna najkrotsza znaleziona droga: 3602.38

Jakosc optymalizacji: 57.2218%

(3,4), (1,2), (2,5), (100,200), (111,222), (158,458), (123,456), (123,464), (112,555), (136,699), (213,777), (365,752), (582,792), (997,998), (777,569), (690,420), (848,241), (639,179), (400,300), (98,26)

7. Wnioski.

Po sprawdzeniu wyników, z wynikami z programu Visual Genetic TSP (program rozwiązujący problem komiwojażera), możemy stwierdzić, że nasz algorytm działa poprawnie. Dodatkowo, możemy zauważyć, że przy dużych wartościach zmiennych takich jak “generations” czy “numberOfCities” czas trwania programu znacząco się wydłuża. Wynika to, z konieczności dokonania większej ilość operacji optymalizujących naszą drogę. Program był napisany w języku C++, sprawdzony na komputerach z systemem Windows 10 i w środowisku Visual Studio.

8. Literatura.

1. Z. Michalewicz: “Algorytmy genetyczne + struktury danych = programy ewolucyjne.” Warszawa: WNT, 1996

[https://drive.google.com/drive/folders/1ofasPKnvzi5XcrK9bk8G0bycPap7qhFfbclid=IwAR0HcvVWmksZ0486eszLHV58mSaO7kg9N4vqz43JhoChfZskJXCxGme_XkI]

2. Wiadomości z wykładów i zajęć.

3. D. E. Goldberg: Algorytmy genetyczne i ich zastosowania. Warszawa: WNT, 1998.

4. John R. Koza: Genetic Programming: On The Programming of Computers by Means of Natural Selection. MIT Press, 1992.

Dodatkowe materiały:

- Program Visual Genetic TSP (użyty do sprawdzenia poprawności wyników)
- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
- https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/