
	Instytut Informatyki Politechniki Śląskiej			
	Zespół Mikroinformatyki			
	i Teorii Automatów Cyfrowych			
Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot:	Grupa	Sekcja
2009/2010	SSI	BMMSI	BDiSK2	2
Prowadzący przedmiot:	dr inż. Grzegorz Baron		Termin: (dzień tygodnia godzina)	
Imię i Nazwisko:	Paweł Wiejak		Środa 13:30 – 15:00	
Imię i Nazwisko:	Karol Podsiadło			
Email:	karolpodsiadlo@op.pl			
Karta projektu				
Temat projektu:				
Problem Komiwożera.				
Główne założenia projektu:				
<ul style="list-style-type: none"> - Wyznaczenie na najkrótszej trasy pomiędzy poszczególnymi miastami - Wykorzystanie algorytmu genetycznego do rozwiązania problemu. - Krzyżowanie metodą OX. - Wczytywanie miast z pliku. - Język programowania : C# 				

1. Temat

Tematem naszego projektu było napisanie programu rozwiązującego problem komiwojażera za pomocą algorytmu genetycznego. Rozwiązaniem problemu było znalezienie w miarę krótkiej (czasem najkrótszej) drogi, jaką musi przejść komiwojażer, aby odwiedzić wszystkie miasta. Algorytm nie zawsze dawał najlepsze rozwiązanie, ale odpowiednie zwiększenie populacji bądź ilości pokoleń dawało nam coraz lepsze wyniki.

2. Analiza, projektowanie.

2.1 Algorytmy, struktury danych, ograniczenia specyfikacji.

Problem komiwojażera jest NP-trudny, oznacza to, że z każdym dodanym miastem jego złożoność rośnie wykładniczo. Dlatego rozwiązanie tego obliczanie wszystkich kombinacji zmusza nas to sprawdzenia $N!$ rozwiązań, gdzie N to liczba miast. Algorytm genetyczny pozwala znacznie zmniejszyć ilość obliczeń, jednak nie gwarantuje uzyskania najlepszego wyniku.

W naszym algorytmie użyliśmy selekcji metodą turniejową., która polega na wyborze kilku osobników z populacji, a następnie najlepszy z nich przechodzi dalej do krzyżowania. Dzięki takiemu rozwiązaniu już podczas selekcji eliminujemy najsłabsze osobniki. Do krzyżowania użyliśmy krzyżowania z porządkowaniem - OX. W tym krzyżowaniu potomków tworzy się na podstawie podtras pobranych z rodziców (podtrasa pierwszego dziecka pobierana jest z drugiego rodzica, natomiast podtrasa drugiego dziecka z pierwszego). Następnie uzupełnia się trasy miastami pobranymi z drugiego rodzica z zachowaniem porządku z pominięciem miast już wykorzystanych.

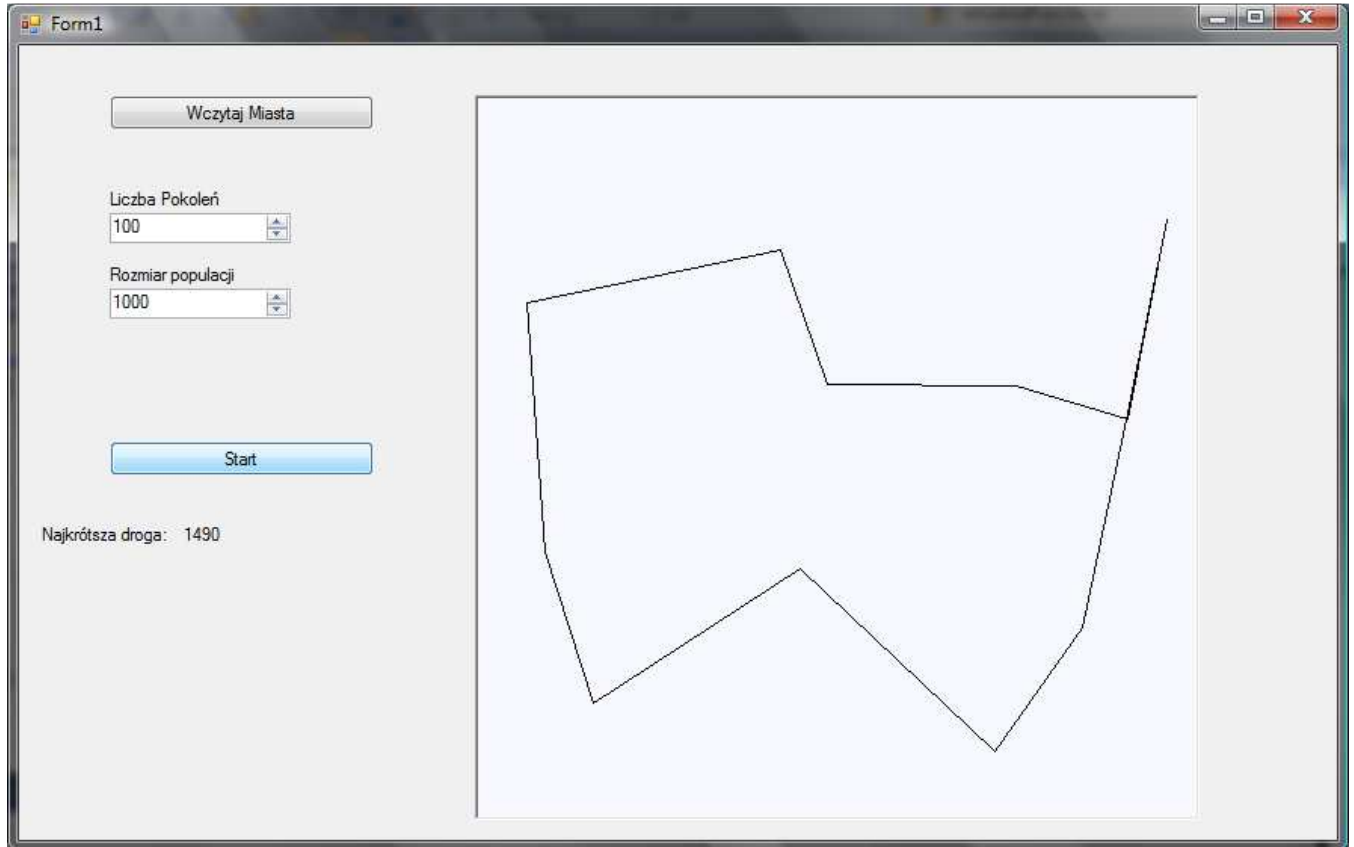
Struktury danych nie były tutaj poważnym problemem do rozwiązania. Każde miasto jest obiektem klasy, która ma indeks porządkowy miasta oraz jego współrzędne. Osobniki populacji to obiekty klasy, która posiada listę miast w kolejności, w jakiej są one odwiedzane oraz długość tej trasy. Podobnie populacja, jej obiekt to lista osobników. Dodatkowa klasa, która była nam potrzebna to klasa z odległościami, która przechowuje macierz dwuwymiarową z wszystkimi odległościami między miastami. Dzięki niej obliczanie drogi dla danego osobnika stało się bardzo łatwe.

3. Specyfikacja zewnętrzna.

3.1 Obsługa programu.

Po załączeniu programu ukazuje nam się główne okno aplikacji. Na początku powinniśmy wczytać wszystkie miasta z pliku (plik Miasta.txt w

katalogu bin\Debug) poprzez naciśnięcie przycisku ‘Wczytaj Miasta’. Na planszy umieszczonej na prawo ukażą nam się wszystkie miasta reprezentowane przez punkty. Następnie możemy określić rozmiar populacji oraz ilość pokoleń. Aby rozpocząć działanie algorytmu wystarczy nacisnąć przycisk ‘Start’. Po skoczeniu algorytmu wyświetla nam się przebieg trasy na planszy po prawej oraz długość trasy poniżej przycisku ‘Start’.



Rys.1 Widok okna programu.

4. Specyfikacja wewnętrzna.

Klasy w programie:

Miasto	Klasa przechowująca współrzędne miasta oraz jego indeks porządkowy.
Miasta	Klasa przechowująca listę obiektów klasy Miasto. W konstruktorze klasy następuje wczytanie wszystkich miast z pliku i dodanie ich do listy.
Odleglosci	Klasa służąca do przechowywania w macierzy dwuwymiarowej odległości pomiędzy wszystkimi miastami. Obiekt klasy przechowuje listę miast, ilość miast oraz macierz typu int. Metoda ObliczOdleglosci() wylicza wszystkie odległości pomiędzy miastami i zapisuje je do macierzy.
Osobnik	Klasa, której obiekty reprezentują pojedyncze osobniki populacji. Każdy osobnik posiada listę miast w kolejności, w jakiej miasta powinny być odwiedzane oraz długość całej

	<p>trasy.</p> <p>Metoda <code>GenerujLosowego()</code> służy do wygenerowania losowej listy odwiedzanych miast. Jest używana podczas generowania początkowej losowej populacji.</p> <p>Metoda <code>LiczTrase()</code> oblicza całkowitą długość trasy na podstawie listy odwiedzanych miast.</p> <p>Metoda <code>LosujLiczbe()</code> jest używana w metodzie <code>GenerujLosowego</code>, gdy chcemy wybrać kolejne losowe miasto i dodać je do listy odwiedzanych miast.</p>
Populacja	<p>Obiekty tej klasy oznaczają kolejne pokolenia i przechowują listę obiektów typu <code>Osobnik</code>.</p> <p>Metoda <code>GenerujKolejnePokolenie()</code> dokonuje selekcji oraz krzyżowania osobników z poprzedniego pokolenia podanego jako argument.</p> <p>Metoda <code>GenerujLosowa()</code> służy do generowania początkowej losowej populacji i jest używana na początku programu.</p> <p>Metoda <code>KrzyzujOsobniki()</code> dokonuje krzyżowania dwóch obiektów typu <code>Osobnik</code>, podanych jako argumenty.</p> <p>Metoda <code>WybierzOsobnikiDoKryzowania()</code> dokonuje selekcji osobników metodą turniejową.</p>

5. Testowanie

Podczas testowania programu zauważyliśmy jedną ważną zależność. Mianowicie, program daje dużo lepsze wyniki, jeśli zwiększymy rozmiar populacji, niż gdy zwiększymy ilość pokoleń. Czas działania programu jest zależny od wszystkich możliwych parametrów, ale zależy głównie od ilości miast oraz pokoleń. Gdy zwiększymy ilość miast do kilkunastu oraz liczbę pokoleń do np. 1000, na wykonanie algorytmu musimy czekać nawet kilkanaście sekund. Lepszy efekt uzyskujemy zwiększając rozmiar populacji, co pozwala nam zyskać kilka sekund.

Przeprowadzone testy:

populacja	10	10	10	10	100	1 000
pokolenia	10	100	1 000	10 000	10 000	10 000
czas	00:00:00.0110000	00:00:00.0300000	00:00:00.2330000	00:00:02.1670000	00:00:21.0110000	00:03:42.2670000
dlugosc trasy	2 038	1 848	1 605	1 480	1 377	1 203
liczba miast	10					

populacja	10	100	1000	10000	100000	10000	10000
pokolenia	10	10	10	10	10	100	1000
czas	00:00:00.0050000	00:00:00.0260000	00:00:00.2410000	00:00:02.5890000	00:00:26.6840000	00:00:24.4560000	00:04:04.0850000
dlugosc trasy	1720	1580	1330	1298	1203	1203	1203
liczba miast	10						

Na podstawie powyższych testów zauważyliśmy, że wydajniejsze jest zwiększenie rozmiaru populacji, osiągamy w ten sposób bardziej optymalne trasy bez znacznej różnicy czasowej przy odwrotnych wartościach tych parametrów.

6. Wnioski

Podczas pisania oraz testowania programu zdaliśmy sobie sprawę, że nie algorytmy genetyczne nie dostarczają nam najlepszego rozwiązania, jednak potrafią dać wynik satysfakcjonujący w czasie o wiele krótszym, niż wykonanie algorytmu zwykłą rekurencją.