Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 17:05*
Temat *Algorytm genetyczny*	**Problem** *TSP*
Autor *229218 Michał Honc*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *17 stycznia 2019*

1 Opis problemu

Problem Komiwojażera (Travelling Salesman Problem), W problemie mamy n miast oraz macierz nxn przejść pomiędzy miastami. Za zadanie mamy znaleźć jak najkrótszą ścieżkę przechodzącą przez wszystkie miasta wliczając w to powrót do miasta początkowego.

2 Metoda rozwiązania

2.1 Algorytm genetyczny

Metaheurystyczny algorytm genetyczny czerpie swoją nazwę ze zjawiska ewolucji biologicznej, bowiem był na nim wzorowany. Zaliczany jest do grupy algorytmów ewolucyjnych stworzonych przez John'a Henry'ego Holland'a. W praktyce algorytm genetyczny opiera się na utworzeniu populacji, najczęściej poprzez losowanie rozwiązań z naszego środowiska, następnie zaś na procesie 'rozmnażania' jej. Nie jest to jednak rozmnażanie w sensie powiększania puli rozwiązań, lecz sukcesywne zastępowanie bieżącej populacji z pomocą nowych genotypów. Proces rozmnażania opiera się na 3 etapach: selekcji, krzyżowaniu i mutacji. Selekcja polega na wybraniu z populacji rodziców do następnej krzyżówki. Dobry algorytm selekcji opiera się na wartościach funkcji przystosowania osobników i faworyzuje, mniej lub bardziej w zależności od zadanych parametrów, osobniki z o lepszych cechach – czyli lepsze rozwiązania. Krzyżówka jest to proces utworzenia nowego rozwiązania (dziecka) z pary innych rozwiązań (rodziców). Pozwala nam na 'przechodzenie' zbioru rozwiązań i poszukiwanie najlepszych wartości funkcji celu. Mutacja to proces samodoskonalenia polegający na zmianie genotypu dziecka. Dzięki niej możemy zdywersyfikować poszukiwania optymalnego rozwiązania i uniknąć utknięcia w minimum lokalnym.

```
1: popsize ← desired population size
2: P ← {}
3: for popsize times do
4:     P ← P ∪ {new random individual}
5: Best ← □
6: repeat
7:     for each individual P_i ∈ P do
8:         AssessFitness(P_i)
9:         if Best = □ or Fitness(P_i) > Fitness(Best) then
10:            Best ← P_i
11:    Q ← {}
12:    for popsize/2 times do
13:        Parent P_a ← SelectWithReplacement(P)
14:        Parent P_b ← SelectWithReplacement(P)
15:        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
16:        Q ← Q ∪ {Mutate(C_a), Mutate(C_b)}
17:    P ← Q
18: until Best is the ideal solution or we have run out of time
19: return Best
```

Rysunek 1: Pseudokod zaimplementowanego algorytmu.

2.2 Omówienie pseudokodu

Na początku określamy rozmiar popsize populacji. Program zostałnapisany dla reprezentacji ścieżkowej.

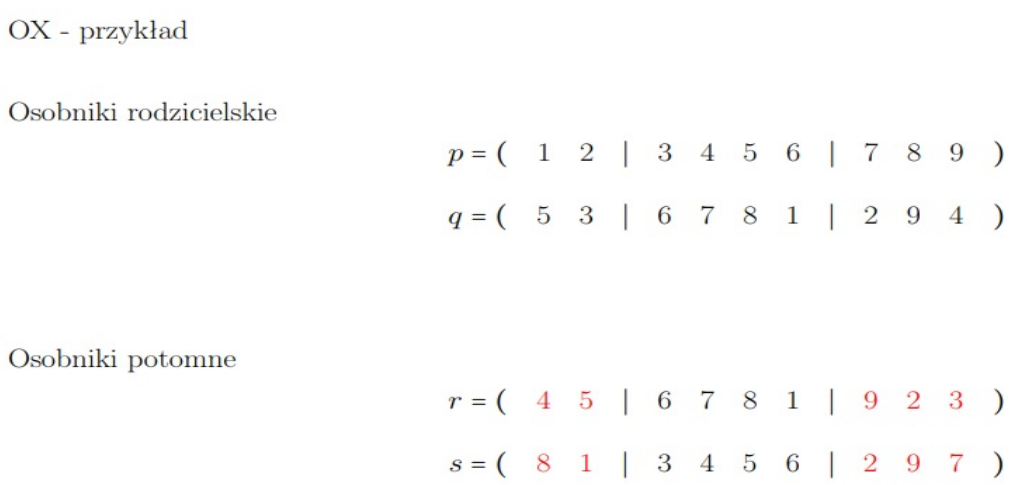
Kolejnym krokiem jest utworzenie pętli głównej programu, która może być wykonywana aż do znalezienia optymalnego rozwiązania, upływu zadanego czasu, bądź przez określoną liczbę iteracji. W prezentowanym programie wykorzystane zostało ostatnie rozwiązanie.

Na początku pętli konieczne jest ocenienie jakości wszystkich osobników populacji i zapamiętanie najlepszego dotychczasowego rozwiązania. Jakość osobników będzie miała znaczenie dla selekcji rodziców. Zapamiętanie najlepszego osobnika jest konieczne, aby program mógł pod koniec działania zwrócić go nam – w końcu poszukujemy najlepszego możliwego do znalezienia rozwiązania.

Najważniejszą częścią algorytmu jest druga pętla for w głównej pętli programu. To w niej wywoływane mamy funkcje selekcji, krzyżowania i mutacji, które określają przebieg poszukiwań minimum globalnego.

Funkcja SelectWithReplacement została zaimplementowana jako selekcja turniejowa. Jest to algorytm pozwalający na pseudolosowe poszukiwanie rozwiązań w zależności od rozmiaru turnieju. Rozmiar turnieju decyduje o tym, czy selekcja jest w pełni (pseudo)losowa, czy też faworyzowane są osobniki o najlepszych cechach oraz jak bardzo są faworyzowane.

Za pomocą funkcji selekcji wybieramy parę rodziców i możemy przystąpić do ich krzyżówki. Krzyżowanie Crossover zostało zaimplementowane za pomocą operatora OX. Jest to efektywny sposób pozwalający na uzyskanie dobrych wyników końcowych. Do rozmnażania jednak nie musi dojść w każdym przypadku. Przeważnie określa się je za pomocą parametru jego prawdopodobieństwa i tak też dzieje się w przedstawionym programie.



Rysunek 2: Przykład krzyżówki OX.

Funkcja Mutate została zaimplementowana za pomocą mutacji invert, poniważ dawała lepsze rezultaty od funkcji insertion(swap). Jest to prosta mutacja. Mutacja zachodzi z określonym prawdopodobieństwem. Polega ona na zamianie danego łańcucha miast w odwrotnej kolejności.

- *inversion* - wybiera losowo podciągi miast i zamienia ich kolejność.

$$p = (\ 1\ \ 2\ \mid\ 3\ \ 4\ \ 5\ \ 6\ \mid\ 7\ \ 8\ \ 9\ \ ) \ \rightarrow\ p' = (\ 1\ \ 2\ \mid\ 6\ \ 5\ \ 4\ \ 3\ \mid\ 7\ \ 8\ \ 9\ \ )$$

Rysunek 3: Przykład inwersji.

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem Intel Pentium, kartą graficzną zintegrowaną, 8GB RAM i DYSK SSD. Algorytmy operują na zmiennych integer 32bit. Pomiar czasu wykonany jest w mikrosekundach.

Pierwszym testowanym parametrem jest liczba iteracji, wyniki przedstawiono dla 17 miast i 100 osobników w populacji, parametrze turniejowym 2, prawdopodobieństwa mutacji 0.8 i prawdopodobieństwa krzyżówki 0.1.

Tablica 1:

iteracje	$t[us]$	*koszt*	*błąd*[%]
10	4.76815×10^{2}	3219	54
100	3.83569×10^{3}	2519	21
1000	2.31316×10^{4}	2313	11
10000	$3.60153\ \times10^{5}$	2179	5
100000	$3.63166\ \times10^{6}$	2210	7

Następnie przetestowano zachowanie algorytmu dla zmieniającej się populacji, przyjęto liczbę iteracji 100.

Tablica 2:

populacja	$t[us]$	*koszt*	*błąd*[%]
10	3.89577×10^{2}	2930	41
100	3.83569×10^{3}	2519	21
1000	3.94649×10^{4}	2346	13
10000	$4.46220\ \times10^{5}$	2229	7
100000	$6.64901\ \times10^{6}$	2120	2

Następnie algorytm genetyczny porównano z algorytmem programowania dynamicznego oraz symulowanego wyrzażania(SA) w tabeli nr 3, parametry dobrano tak, aby czasy zbytnio od siebie nie odbiegały, gdzie:

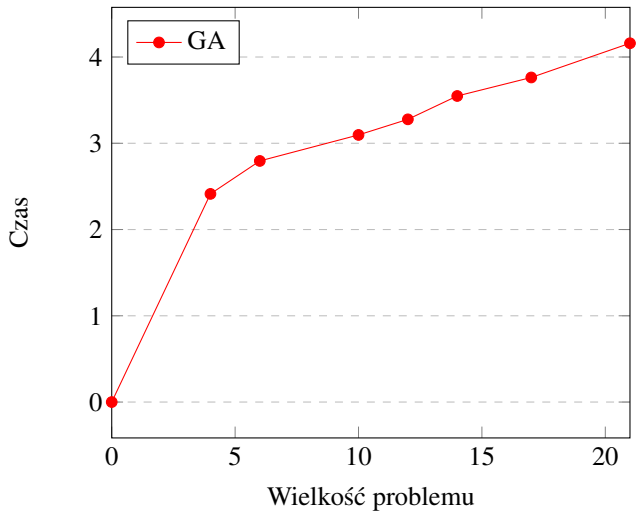
- n - liczba miast,
- D - koszt najkrótszej ścieżki,
- *BSA* - błąd znalezionej najkrótszej ścieżki znalezionej przez algorytm SA dla tempreratury chłodzenia 0,9999,

- *BGA* - błąd najkrótszej ścieżki znalezionej przez algorytm genetyczny dla liczby iteracji 10000 i poopulacji 5000 dla miast >=29, oraz 10000 i 1000 dla reszty
- $DP[us]$ - czas dla algorytmu programowania dynamicznego
- $SA[us]$ - czas dla algorytmu symulowanego wyżarzania dla tempreratury chłodzenia 0,9999
- $GA[us]$ - czas dla algorytmu genetycznego

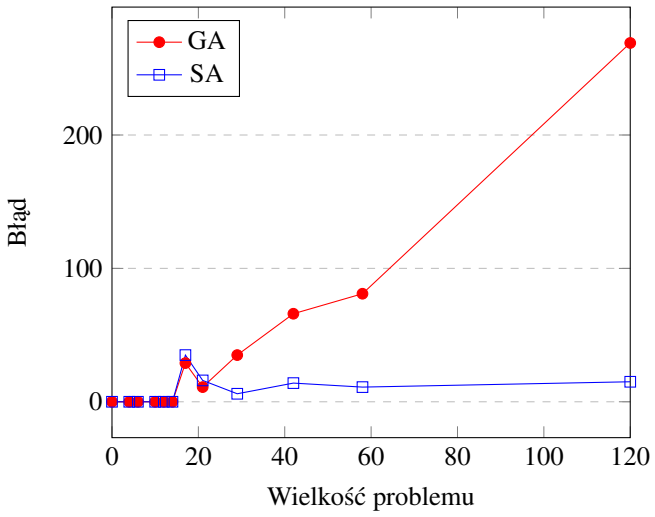
Wnioski - algorytm genetyczny działa w czasie liniowym, spisuje się dobrze dla małych instancji(<=21) od algorytmu symulowanego wyrzażania, jednak dla większych instancji działa o wiele gorzej.

Tablica 3: Czas obliczeń oraz błąd dla ustalonej liczby miast.

n	$DP[us]$	$GA[us]$	$SA[us]$	D	$BGA[\%]$	$BSA[\%]$
0	0	0	0	0	0	0
4	1.34×10^2	2.41316×10^6	2.90925×10^5	88	0	0
6	2.12×10^2	2.79511×10^6	5.26230×10^5	132	0	0
10	1.802×10^3	3.0966×10^6	1.39959×10^6	212	0	0
12	1.0383×10^4	3.2775×10^6	2.09477×10^6	264	0	0
14	5.2670×10^4	3.54814×10^6	2.84479×10^6	282	0	0
17	5.7411×10^5	3.76287×10^6	4.20395×10^6	2085	29	35
21	1.4324×10^7	4.15967×10^6	8.04416×10^6	2707	11	16
29		2.67573×10^7	1.53412×10^7	1610	35	6
42		4.02251×10^7	3.45926×10^7	699	66	14
58		4.98914×10^7	9.44869×10^7	25395	81	11
120		1.09387×10^8	5.62488×10^8	6942	269	15



Rysunek 4: Czas wykonywania się algorytmu w mikrosekundach w zależności od wielkości problemu



Rysunek 5: Błąd % znalezionego rozwiązania dla algorytmu genetycznego i SA.