## Zastosowanie algorytmu roju (PSO) do rozwiązywania problemu komiwojażera

### Sprawozdanie 2 - Maciej Muszkowski, Maciej Roman

### Algorytm

Do rozwiązania problemu komiwojażera zastosowaliśmy algorytm PSO (Patricle Swarm Optimalization – roju). Rój składa się z n cząsteczek, z których każda poszukuje najlepszego rozwiązania (w wypadku problemu komiwojażera jest to minimalizacja długości przebytej drogi). Kierunek przeszukiwania przestrzeni rozwiązań jest podyktowany zarówno poprzez indywidualne doświadczenie cząsteczki (najlepsze znalezione przez tą cząsteczkę rozwiązanie) jak i przez doświadczenie całego roju (najlepsze rozwiązanie znalezione przez rój). W ten sposób cząsteczki wzajemnie na siebie oddziaływują, unikają „utknięcia” w minimum lokalnym, stopniowo przenosząc się do coraz lepszych obszarów przestrzeni rozwiązań.

Zapis algorytmu w pseudokodzie:

init\_population(); // losowe przypisanie położenia i prędkości każdej cząsteczce

t=0;

do

for i=1 to popSize do

if(f(Xi(t)) < f(Pi)) Pi = Xi(t);

Pg = min(Pall);

updateVelocity(); // uaktualnienie prędkości cząstki

updateSpeed(); // uaktualnienie położenia cząstki

end for;

t++;

until end\_criterium\_reached(); // póki nie osiągnięte kryterium stopu

Przyjęte oznaczenia:

* – iteracja.
* – funkcja oceny (długość drogi).
* – najlepsze rozwiązanie znalezione przez cząstkę i-tą. Wektor.
* - najlepsze rozwiązanie (globalne, roju). Wektor.
* – położenie i-tej cząstki w n-wymiarowej przestrzeni w iteracji t. Wektor.
* – prędkość i-tej cząstki w n-wymiarowej przestrzeni w iteracji t. Wektor.

Kryterium stopu zostanie opisane w następnych rozdziałach.

Położenie i prędkość każdej cząstki, są uaktualniane w oraz na podstawie wzorów:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Algorytm operuje na liczbach zmiennoprzecinkowych, aby przekształcić położenie cząstki na cykl Hamiltona, który może być przekazany do funkcji oceny, zostanie wykorzystana funkcja:

std::vector<int> toPermut(std::vector<float> values)

{

unsigned int i, n = values.size();

std::vector<int> res(n);

for(i=0; i < n; i++) // permutacja 0..n-1

res[i] = i;

for(i=0; i < n; i++) // zamiana o wartość w tablicy

std::swap(res[i], res[(int)(values[i])%n]);

return res;

}

Do zaimplementowania algorytmu został wybrany język C++.

### Algorytm w wersji dyskretnej

// TODO: przepisac z artykulu [1]

### Struktury danych

// TODO: tu przepisac to z artykulu

Położenie każdej cząstki reprezentowane jest przez n-elementową tablicę (n-ilość punktów, które mają zostać odwiedzone) zawierającą liczby zmiennoprzecinkowe. Ta tablica, może zostać przetworzona, przy pomocy powyższej funkcji, do tablicy zawierającej permutację liczb naturalnych, będących indeksami odwiedzanych punktów (indeksy od 0 do n-1).

Do reprezentacji grafu zawierającego odległości pomiędzy punktami, ze względu na to, że jest to graf pełny (możliwe jest połączenie dowolnego punktu z innym), wybrano reprezentację macierzową grafu.

### Projekty testów

Program podlega dwóm kategoriom testów – jakościowych oraz wydajnościowych. Z tego powodu konieczne było zaimplementowanie innych algorytmów, z którymi porównywane są uzyskane wyniki. Są to:

* Algorytm losowego przeszukiwania przestrzeni rozwiązań.
* Algorytm symulowanego wyżarzania.

Porównywana jest długość najkrótszego znalezionego cyklu oraz czas takich poszukiwań dla różnych grafów.

### Założenia programu

Kryteria stopu

// TODO: w artykule sa 4 rozne,mozna wybrac tylko niektore

#### Złożoność obliczeniowa

Złożoność obliczeniowa jest zależna od liczby cząsteczek roju oraz kryterium stopu. Jako takie kryterium przyjęliśmy sytuację, gdy w ciągu ostatnich iteracji nie otrzymaliśmy lepszego rozwiązania. Z tego powodu złożoność obliczeniowa jest trudna do oszacowanie, gdyż liczba iteracji głównej pętli algorytmu może być zmienna. // TODO: napisac jaka pesymistyczna w zaleznosci od liczby iteracji

#### Wejście/wyjście programu

Program implementujący/testujący jest aplikacją konsolową. Dane wejściowe mogą zostać podane na 2 sposoby, wczytane z pliku lub wygenerowane losowo z podaną maksymalną odległością pomiędzy 2 punktami. Na wyjściu wypisana zostanie długość najkrótszego znalezionego cyklu, czas wykonywania obliczeń oraz liczba punktów wejściowych. Opcjonalnie możliwe jest wypisanie macierzy odległości pomiędzy punktami oraz porównanie wyniku z wynikami działania innych algorytmów.

#### *Parametry*

Dla algorytmu symulowanego wyżarzania możliwe jest ustalenie liczby cząsteczek roju i maksymalnej liczby iteracji nieznajdujących lepszego rozwiązania. // TODO: są różne modyfikacje tego algorytmu

### Wykaz literatury

1. **Clerc, Maurice.** Discrete Particle Swarm Optimization. [Online] 2000. http://clerc.maurice.free.fr/pso/pso\_tsp/Discrete\_PSO\_TSP.htm.