Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie efektywnych algorytmów

Projekt nr 4 - Implementacja i analiza efektywności algorytmu genetycznego dla problemu komiwojażera

semestr zimowy 2023/2024

Autor:
Eryk Mika 264451

Prowadzący:
Dr inż. Marcin Łopuszyński

Spis treści

1. Wstęp teoretyczny[1]

Zgodnie z informacjami przedstawionymi w poprzednich sprawozdaniach, problem komiwojażera (*TSP*) jest problemem trudnym pod względem obliczeniowym. W tym opracowaniu zostanie omówione rozwiązanie tego problemu z wykorzystaniem algorytmu genetycznego. Jest to rodzaj algorytmu ewolucyjnego - jest wzorowany na biologicznej ewolucji oraz stosowany jest do optymalizacji oraz planowania.

Algorytm genetyczny jest heurystyką[2]. Symuluje on proces naturalnej selekcji poprzez ocenę adaptacji poszczególnych jednostek, eliminację słabszych osobników oraz krzyżowanie tych o największym przystosowaniu – w ten sposób powstają nowe osobniki w populacji. Każdy osobnik reprezentuje określony sposób rozwiązania problemu, który wyznacza dany *chromosom*. Osobniki oceniane są według pewnego kryterium – *funkcji oceny* – która, w przypadku problemu *TSP*, może być rozumiana jako funkcja przyporządkowująca koszt do danej trasy. Istotnym elementem algorytmu jest także mutacja, która polega na zmianie pewnych elementów rozwiązania według pewnego wzorca z określonym prawdopodobieństwem. Efektem tego procesu jest populacja jednostek, z których wybierane są te o najwyższym stopniu przystosowania. Zbiór informacji całej populacji określa się jako *genotyp*.

1.1. Metody krzyżowania

Krzyżowanie, realizowane poprzez *operator krzyżowania*, polega na kombinacji cech różnych osobników z populacji, co prowadzi do powstania nowych rozwiązań. Krzyżowanie zachodzi z pewnym ustalonym prawdopodobieństwem.

W zaimplementowanym i omawianym algorytmie zastosowano operator krzyżowania **PMX** (ang. *partially matched crossover*) – krzyżowanie z częściowym odwzorowaniem. W algorytmie realizującym ten operator wybierane są dwa punkty podziału, które wyznaczają tzw. sekcję dopasowania (ang. *matching section*). W ten sposób definiowane są punkty, które wyznaczają sposób transpozycji (zmianę miejsc) elementów danego rozwiązania[3] – szczegółowy opis algorytmu zawarty jest w opisie implementacji.

[1] https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy_genetyczne.pdf
[2] https://pl.wikipedia.org/wiki/Algorytm_genetyczny
[3] https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf

1.2. Metody mutacji[4]

Mutacja polega na wprowadzaniu losowych zmian do genotypu populacji. Ma to na celu zwiększenie różnorodności generowanych rozwiązań. Mutacja zachodzi z pewnym ustalonym prawdopodobieństwem, które z reguły jest niewielkie (≤1%), co ma na celu zachowanie równowagi pomiędzy przeszukiwaniem lokalnym (wokół pewnej grupy rozwiązań) oraz zwiększaniem przeszukiwanej przestrzeni rozwiązań[5].

W prezentowanym projekcie zastosowano dwa operatory mutacji: ***inverse*** oraz ***scramble***.

Operator *inverse* polega na odwróceniu kolejności elementów rozwiązania pomiędzy dwoma przyjętymi punktami w chromosomie (Rysunek 1.1).

Rysunek 1.1 Przykład zastosowania operatora inverse

Źródło: opracowanie własne

Operator *scramble* polega na losowym przestawianiu wybranych elementów z genotypu (Rysunek 1.2).

Rysunek 1.2 Przykład zastosowania operatora scramble

Źródło: opracowanie własne

1.3. Metoda selekcji

Selekcja w algorytmie genetycznym polega na wybieraniu osobników z populacji, które przejdą do następnego pokolenia (iteracji algorytmu). Możliwa jest realizacja selekcji na wiele sposobów. Do najważniejszych należą między innymi tzw. metoda ruletki oraz **metoda rankingowa[6]**, która została zaimplementowana i użyta w przedstawionym projekcie.

Metoda rankingowa w zaimplementowanej postaci polega na posortowaniu osobników w populacji rosnąco według przyjętej funkcji oceny – kosztu danej trasy. Następnie, zakładając, że do następnego pokolenia przechodzi *n* najlepszych osobników, pozostałe (gorsze) są usuwane (przykład przedstawia Rysunek 1.3).

Rysunek 1.3 Przykład zastosowania metody rankingowej. Populacja jest przedstawiona w postaci tablicy kosztów osobników.

[**55, 62, 74, 80, 91**, 100, 120, 182] → [55, 62, 74, 80, 91] *n = 5*

Źródło: opracowanie własne

[4] https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf
[5] https://www.baeldung.com/cs/genetic-algorithms-crossover-probability-and-mutation-probability
[6] https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)

1.4. Populacja początkowa

Populacja początkowa jest grupą osobników – rozwiązań, od których zaczyna swoje działanie algorytm. Wielkość tej populacji jest różna i zazwyczaj zależy ona od specyfiki rozwiązywanego problemu[7]. Często stosuje się wygenerowanie całości populacji w sposób losowy, jednakże spotyka się także podejście z wykorzystaniem „ziarna", które stanowią osobniki, o których wstępnie wiadomo, że mogą być obiecujące – są wygenerowane, na przykład, za pomocą metody zachłannej[8]. „Ziarno" to stanowi pewną część osobników populacji początkowej oprócz osobników wygenerowanych losowo. Ten sposób został wykorzystany w tym projekcie.

2. Opis implementacji algorytmów

W celu analizy efektywności omawianych algorytmów został napisany program w języku C++ z wykorzystaniem obiektowego paradygmatu programowania. Najistotniejszymi komponentami aplikacji są klasy *Graph* oraz *Route*, których pola (struktury danych) oraz metody są odpowiedzialne za realizację algorytmu. Wiele istotnych kwestii związanych z implementacją zostało wyjaśnionych w komentarzach w plikach źródłowych.

2.1. Klasa *Graph*

Klasa Graph jest główną klasą programu, która jest odpowiedzialna za przechowywanie struktury i metod grafu, na którym wykonywane są badane algorytmy. Pola prywatne klasy – dwuwymiarowa tablica *std::vector matrix* oraz *size* są użyte do przechowywania długości krawędzi w postaci macierzy kwadratowej – kosztów - *matrix* stopnia *size*. Oba pola przechowują liczby stałoprzecinkowe typu *int*. W macierzy komórka o współrzędnych *i*, *j* zawiera odległość pomiędzy wierzchołkami *i i j*.

Zaimplementowano konstruktor wczytujący instancję z pliku tekstowego, przeładowany operator przypisania oraz metodę wypisującą graf (macierz) na ekran. Została zaimplementowana metoda *calculateRouteCost()*, która służy do obliczania kosztu danej trasy komiwojażera w grafie – poprzez iterowanie po krawędziach w ścieżce i dodanie ich długości do kosztu, który jest przypisywany do odpowiedniego pola w obiekcie klasy *Route*.

2.2. Klasa Graph – metoda *generateInitialSolution()*

Metoda ta użyta do wyznaczenia rozwiązania początkowego w sposób **zachłanny**. Rozwiązanie to wykorzystane jest do tworzenia „ziarna" populacji początkowej.

[7] https://en.wikipedia.org/wiki/Genetic_algorithm
[8] https://medium.datadriveninvestor.com/population-initialization-in-genetic-algorithms-ddb037da6773

Algorytm:

1. Utworzenie listy *visited*, zainicjowanej odwiedzeniem korzenia (wierzchołka 0).
2. Rozpoczyna się pętla, która wykonuje się *size-1* razy, ponieważ trasa musi odwiedzić wszystkie wierzchołki oprócz korzenia.
3. Dla aktualnego wierzchołka *curIndex* na trasie, znajdowany jest najbliższy nieodwiedzony wierzchołek *dst* z najmniejszą wagą krawędzi. Wartość minimalnej wagi przechowywana jest w zmiennej *dstMin*.
4. Znaleziony wierzchołek *dst* jest dodany do listy visited i do wynikowej trasy *res* na odpowiedniej pozycji.
5. Po zakończeniu pętli, waga krawędzi powrotnej do korzenia, czyli od ostatnio odwiedzonego wierzchołka do korzenia (wierzchołek 0), jest dodawana do kosztu. Wygenerowana trasa jest zwracana z metody.

2.3. Klasa Graph – metoda *solveGA()*

Metoda ta użyta jest do rozwiązywania problemu komiwojażera za pomocą algorytmu genetycznego.

Tworzony jest wektor *population* przechowujący populację tras. Tworzona jest populacja początkowa. 10% osobników w tej populacji stanowią rozwiązanie uzyskane metodą zachłanną oraz osobniki pochodzące z tego rozwiązania poprzez wywoływania operatora *swap()* (zamiana elementów miejscami). Uzyskana populacja jest sortowana (Rysunek 2.1).

Rysunek 2.1 Algorytm tworzenia populacji początkowej

```
/*
   10% osobników w początkowej populacji wywodzi się z rozwiązania zachłannego
   razem z bazowym rozwiązaniem wygenerowanym zachłannie
*/
while(population.size() < (unsigned)(0.1 * initialPopulation))
{
/* Wygenerowanie osobników pochodzących z rozwiązania zachłannego (greedy) - losowe
przestawianie elementów trasy */
   Route r = greedy;
   r.procedureSwap(rand() % routeElements, rand() % routeElements);
   r.procedureSwap(rand() % routeElements, rand() % routeElements);
   calculateRouteCost(r);
   population.emplace_back(r);
}

// Reszta osobników jest wygenerowana losowo
while(population.size() < initialPopulation)
{
   Route r(routeElements);
   r.randomize();
   calculateRouteCost(r);
   population.emplace_back(r);
}

// Sortowanie populacji wg kosztów tras - podejście rankingowe
std::sort(population.begin(), population.end());
```

Źródło: opracowanie własne

Następnie, po inicjalizacji zmiennych związanych z obsługą pomiaru czasu, rozpoczyna się główna pętla algorytmu.