Algorytmy Metaheurystyczne
Komiwojażer Genetycznie

Gabriel Budziński(254609)
Franciszek Stepek (256310)

Przedmowa

Na samym początku omówimy po krótce naszą implementacje, oraz podamy kilka informacji ogólnych. Następnie bardziej szczegółowo opiszemy poszczególne parametry allgorytmu, a na koniec przedstawimy wyniki i opis wykonywanych eksperymentów.

1 Informacje ogólne

1.1 Implementacja

Algorytmy implementujemy w języku `C/C++`, odległości między wierzchołkami są przechowywane jako pełne tablice dwuwymiarowe typu `int`, a trasy (pojedyncze osobniki) są w kontenerach `vector`, co ułatwia operacje odwracania i mieszania. Korzystaliśmy z kompilatora g++ wraz z użyciem flag -lSDL2 (używanej przy wizualizacji, wraz z odpowiednim dla danego systemu operacyjnego podlinkowania do folderu zawierającego) oraz -lpthread (przy korzystaniu z wielowątkowości).

Dodajmy jeszcze tylko, że jako generatora pseudolosowego użyliśmy typu `std::mt19937` zdefiniowanego przez `C++`.

1.2 Sprzęt

Programy były testowane na dwóch maszynach, laptopie *Lenovo* i komputerze stacjonarnym. Obie jednostki są wyposażone w procesor architektury `x86` marki `intel` oraz 16GB pamięci RAM.

- PC - Komputer stacjonarny posiada procesor sześciordzeniowy i5-10600K 4,1 GHz (o obniżonym napięciu operacyjnym).
- Laptop - Laptop posiada procesor czterordzeniowy i7-6700HQ 2,6 GHz

1.3 Instancje

Używaliśmy instancji przygotowanych przez TSPLIB, które dzielą się na 2 kategorie:

- 8 instancji symetrycznych:
  - berlin52.tsp
  - st70.tsp
  - eil76.tsp
  - bier127.tsp
  - kroA150.tsp
  - lin318.tsp
  - linhp318.tsp
  - pr439.tsp
- 8 instancji asymetrycznych:
  - ftv55.atsp
  - ftv64.atsp

- ftv70.atsp
- kro124p.atsp
- ftv170.atsp
- rbg323.atsp
- rbg358.atsp
- rbg443.atsp

W dalszych częściach, instancje będziemy oznaczać przez liczbę mówiącą o rozmiarze problemu (czyli np. st70.tsp oznaczymy jako $n = 70$, albo po prostu 70).

1.4 Metodologia/cel

Testy przeprowadzono za pomocą zaimplementowanych w tym celu funkcji ku jak największej automatyzacji. Dane o przeprowadzonych testach zapisywano do plików tekstowych w formacie CSV, a następnie poddane analizie. Testy i eksperymenty miały na celu zbadanie wydajności naszej implementacji, oraz znalezienie jak najbardziej optymalnych trybów/hiperparametrów dla przypadku ogólnego.

2 Opis parametrów

W opisie przejdziemy najpierw przez kolejne 'tryby' działania, a następnie omówimy także każdy hiperparametr występujący w naszej implementacji, ale zanim, to wspomnijmy jeszcze tylko, że każda operacja krzyżowania daje nam 2 nowe osobniki.

2.1 Tryby działania

- StartMode - sposób generowania populacji początkowej:
  - 0 - Każdy osobnik jest wybierany z 10 całkowicie losowych (Chodzi o losowe ermutacje dróg)
  - 1 - Każdy osobnik jest tworzny jako puszczenie algorytmu NearestNeighbor (czyli zachłanne szukanie najbliższego sąsiada z tych co pozostali w każdej iteracji) z losowego punktu startowego
  - 2 - Hybrydowe połączenie 2 poprzednich, gdzie stosunek losowych do NN wynosi 4:1 (Czyli około 20% populacji to osobniki 'względnie dobre').
- SelectionMode - sposób w jaki jest wykonywana selekcja osobników:
  - 0 - Turniejowa, czyli wyieramy najlepszych, a najgorszych odrzucamy
  - 1 - Kwadratowo ruletkowa - najlepszy osobnik przechodzi dalej, a wszystkim pozostałym przyporządkowywane są wagi względem kwadratu pozycji (Czyli jeżeli mamy populację wielkości 15, to najlepszy przechodzi dalej, kolejny ma wagę $14 * 14$, później $13 * 13$ itd., a ostatni ma wagę 1), a następnie zgodnie z nimi jest robione losowanie.
- MutMode - sposób przeprowadzenia mutacji (o jej hiperparametrach będzie później):
  - 0 - Mutacja typu *Invert*
  - 1 - Mutacja typu *Insert*
  - 2 - Mutacja typu *Swap*
  - 3 - W każdej iteracji (co to oznacza będzie powiedziane później) losowe wybranie spośród 3 poprzednich
- crossMode - używany operator do krzyżowania osobników:
  - 0 - *Order Based Crossover*
  - 1 - *Modified Order Based Crossover*
  - 2 - *Partially Mapped Crossover*
- crossType - sposób przeprowadzenia i selekcji osobników do krzyżowania:
  - 0 - Wszystkie osobniki są ustawione losowo, a następnie krzyżujemy ze sobą 1 z 2, 3 z 4.. itd. Jeżeli osobników było nieparzyście, to ostatni osobnik jest dublowany.
  - 1 - W każdej iteracji losowana jest para osobników z całej populacji (Ustalona liczba na sztywno)
  - 2 - Tak jak poprzednio, ale tym razem liczba losowań jest określona jako rozmiar problemu / 2 (czyli dla $n = 150$ mamy 75 losowań.

2.2 Hiperparametry

- time - czas działania algorytmu - w naszych eksperymentach każde 1 wywołanie trwa 30 sekund
- populationSize - rozmiar populacji początkowej (oraz co za tym idzie - rozmiar w każdej iteracji, ponieważ selekcja redukuje rozmiar do rozmiaru początkowego)
- mutationThreshold - określa z jakim prawdopodobieństwem zachodzi mutacja (mutacja może zajść podczas tworzenia nowych osobników, rozpatrywana dla każdego z osobna)
- mutationIntensification - górne ograniczenie na liczbę pojedynczych mutacji na jednym osobniku (jeżeli zajdzie mutacja, to następnie jest losowana jej intensyfikacja, co najmniej 1, definiuje liczbę iteracji przy mutacji - dlatego przy zastosowaniu MutMode3 może się okazać, że np. wykonają się 3 typu *Invert* oraz 1 typu *Swap*)
- crossSize - wielkość fragmentu podlegająca krzyżowaniu - przy naszych operatorach jest to wielkość 'wycinka', który definiuje operację krzyżowania
- crossCount - wykorzystywane tylko, gdy crossType 1, definiuje liczbę zachodzących krzyżowań

3 Opis eksperymentów

Jako wyniki eksperymentów będziemy pokazywali wartość funkcji celu dla podanych wykonań (Zazwyczaj jako minimum, oraz średnią z 4 wywołań), lub procentową wartość względem najlepszej znalezionej (informacja ze strony TSPLIB) - które będzie użyte wyniknie z kontekstu. Dodatkowo będziemy mówili również o liczbie wykonanych iteracji, oraz zastanowimy się później nad jej wpływem na ostateczny wynikiem.

3.1 Poszukiwanie I - Tryb

Na samym początku przeprowadzony został eksperyment, który miał na celu znalezienie jak najlepszego zestawu trybów dla naszego algorytmu. Użyliśmy tutaj następującego zestawu hiperparametrów (wybranych empirycznie po kilkunastu przetestowaniach algorytmu):

- populationSize = 20
- mutationThreshold = 0.05
- mutationIntensification = 5
- crossSize = 7
- crossCount = 20

Testy wykonaliśmy dla każdej możliwej kombinacji trybów (3*3*2*4*3 = 216), dla każdej z 16 badanych instancji.
Przypomnijmy, że czas działania ograniczyliśmy do 30 sekund, a żeby odrobinę zredukować losowy wkład metody, każde wywołanie powtórzyliśmy 4 razy.

3.2 Poszukiwanie II - hiperparametry

Po wyznaczeniu rokujących zestawów trybów, przeszliśmy do wyznaczenia jak najlepszych hiperparametrów. W tym celu wyznaczyliśmy 1 zestaw (najlepszy pod względem średniej) dla wariantu symetrycznego, 1 analogicznie dla asymetrycznego, oraz 1 wspólny, który dla obu był w pierwszej 3 (był tylko 1 taki zestaw).
Poprzez metodę losowego próbkowania dla każdego hiperparametru (losowy z zakresu podanego zaraz), wykonaliśmy 100 kombinacji, gdzie każdą testowaliśmy 4 razy.
Badane instancje:

- st70.tsp - sym.
- lin318.tsp - sym.
- ftv70.atsp - asym.
- rbg323.atsp - asym.

Badany zakres hiperparametrów:

- populationSize : $[10; 100]$
- mutationThreshold : $[0.0; 1.0]$
- mutationIntensification : $[1; 20]$
- crossSize : $[2; 20]$
- crossCount : $[10; 200]$

Z przeprowadzonych eksperymentów otrzymaliśmy następujące wyniki:

3.3 Badanie wpływu zastosowania 'lokalnej poprawy'

Przy zastosowaniu wyników z 2 poprzednich eksperymentów postanowiliśmy zbadać wpływ działania mechanizmu 'lokalnej poprawy' na wyniki.
Lokalna poprawa - z pewnym prawdopodobieństwem (określanym parametrem *enhanceChance*) po skończonej mutacji (czyli po szystkich iteracjach) na osobniku wykonujemy algorytm lokalnej poprawy, czyli iteracyjnie przechodzimy przez piątki kolejnych miast i tak modyfikujemy trasę, aby w każdej z tych iteracji przejście było minimalne - zatem najpierw 'poprawiamy' miasta 1-5, potem 2-6 itd. Takich możliwych przejść jest 6 (ponieważ zaczynamy zawsze z 1 i kończymy na 5), zatem jest to w miarę szybkie (dzieje się w czasie liniowym względem liczby miast) i nie powinno znacząco wpływać na liczbę wykonywanych iteracji.
W testach wykonaliśmy 10 powtórzeń dla każdej wartości parametru z zakresu $[0.05; 1.0]$ ze skokiem o 0.05. Wyniki przedstawmy na wykresie:

3.4 Badanie wpływu 'wieku' osobników

Eksperyment analogiczny do poprzedniego (w sensie metodologii), jednak tym razem badaliśmy wpływ zastosowania wieku na rozwiązanie. Oznaczyliśmy go jako *AgeMax*, przy czym oznacza to, że osobnik będący w populacji dłużej niż *AgeMax* w procesie selekcji jest 'usuwany' z populacji. Zastosowaliśmy tutaj jednak pewną formę elitaryzmu, ponieważ podczas każdej selekcji 'zerujemy' wiek najlepszego osobnika (Najlepszy ma prawo *Picia ze źródła wiecznej młodości*), dzięki czemu go nie tracimy.
W testach wykonaliśmy 10 powtórzeń dla każdej wartości z zakresu $[1; 20]$, a wyniki przedstawiamy na wykresie:

3.5 Porównanie najlepszych wyników z algorytmem TabuSearch

Aby jakoś porównać działanie naszego algorytmu, zestawimy go tutaj z zaimplementoanym algorytmem *TabuSearch*, gdzie Tabu będzie miało następujące parametry:

- par1
- par2

Natomiast *Genetic* otrzymane z eksperymentów z poszukiwaniami. Dla przypomnienia:

- par1
- par2

Testowaliśmy każdą z 16 badanych instancji, oraz każdemu z algorytmów daliśmy budżet czasu równy 90 sekund. Aby jednak dać jakąś szansę (i odrobinę zredukować losowość) algorytmowi genetycznemu, wykonaliśmy dla niego 3 iteracje po 30 sekund. W ten sposób jego wyniki mają szansę być nieco bardziej miarodajne. Wyniki przedstawmy w tabeli:
Oraz jeszcze zobaczmy, co Wilcoxon nam o tym mówi:

3.6 Badania nad Modelem Wyspowym

4 Drobne podsumowanie; Tabele dodatkowe

Zawrzemy tutaj niepokazane wcześniej tabele z wynikami, oraz pokusimy się o podsumowanie naszych eksperymentów.