

Badania Operacyjne - Projekt

Michał Dziędzic, Kacper Sala, Paweł Tyszkowski, Mikołaj Zatorski

2021

1 Wstęp

Celem tego projektu jest zapoznanie się z metodami przybliżonymi i zbadanie ich skuteczności w rozwiązywaniu problemów trudnych obliczeniowo oraz takich, dla których nie istnieją algorytmy optymalne. Wymyślony przez nas problem (opisany w rozdziale drugim), nada się do tego celu, gdyż znalezienie optymalnego rozwiązania tradycyjnymi metodami jest niezwykle trudne. Użyjemy algorytmów genetycznych oraz rojowych w celu rozwiązania naszego problemu, opiszemy ich działanie, przedstawimy otrzymane wyniki i wreszcie postaramy się ocenić, czy są one dla nas satysfakcjonujące. Ten projekt jest realizowany w ramach zajęć laboratoryjnych z przedmiotu Badania Operacyjne, prowadzonego na kierunku Informatyka wydziału IET, AGH.

2 Opis zagadnienia

2.1 Ogólny opis problemu

Naszym celem jest rozwiązanie problemu przeprowadzki. Zakładamy, że mamy n przedmiotów do przewiezienia z punktu A do punktu B. Każdy przedmiot ma określoną masę. Przewozić przedmioty możemy za pomocą naszego samochodu osobowego (ma on oczywiście skończoną ładowność l_C), lub też wynajętego samochodu ciężarowego (ładowność l_T). Każdy przejazd ma swoją cenę, w przypadku samochodu osobowego kosztuje nas paliwo (p_C), w przypadku ciężarowego koszt wynajmu + paliwo (łącznie p_T).

Naszym celem jest znalezienie takiego rozłożenia **wszystkich** przedmiotów pomiędzy przejazdy samochodem osobowym lub ciężarowym, aby łączny koszt naszej przeprowadzki był jak najmniejszy.

2.2 Model matematyczny

2.2.1 Struktury danych

$S = \{s_1, s_2, \dots, s_n\}$ - zbiór przedmiotów
 $mass : S \rightarrow \mathbb{R}_+$ - funkcja zwracająca masę przedmiotu
 $l_T \in \mathbb{R}_+$ - ładowność ciężarówki

$l_C \in \mathbb{R}_+$ - ładowność samochodu osobowego
 $p_T \in \mathbb{R}_+$ - cena przejazdu ciężarówki
 $p_C \in \mathbb{R}_+$ - cena przejazdu samochodu osobowego

2.2.2 Reprezentacja rozwiązania

P - partycja zbioru S (podział przedmiotów na przejazdy odpowiednim samochodem)

$P = \{P_1, P_2, \dots, P_m\} \subset 2^S$, gdzie:

- $\forall i, j \in \{1, 2, \dots, m\} : i \neq j \implies P_i \cap P_j = \emptyset$
- $\bigcup_{i=1}^m P_i = S$
- $\forall P_i \in P : \text{mass}(P_i) = \sum_{s \in P_i} \text{mass}(s) \leq l_T$

2.2.3 Funkcja celu

Minimalizacja łącznego kosztu wszystkich przejazdów, tj.

$$\text{cost}(P) = \sum_{P_i \in P} \text{cost}(P_i) \rightarrow \min,$$

gdzie $\text{cost}(P_i)$ jest kosztem przejazdu i , tj.

$$\text{cost}(P_i) = \begin{cases} 0 & P_i = \emptyset \\ p_C & \text{mass}(P_i) \leq l_C \\ p_T & \text{mass}(P_i) \leq l_T \end{cases}$$

2.2.4 Odległość

Specjalny przypadek odległości edycyjnej definiowany dla partycji zbioru, opisany w [2].

2.2.5 Sąsiedztwo

Pojedyncza zamiana elementu pomiędzy klasami (w tym stworzenie nowej klasy dla tego elementu) tworzy sąsiada danej partycji.

2.2.6 Warunki ograniczające

- Skończone ładowności l_T oraz l_C ,
- Wszystkie przedmioty z S muszą zostać załadowane,
- *Implicit*e założenia o ładownościach i cenach:
 - $l_C < l_T$
 - $p_C < p_T$
 - $\forall s_i \in S : \text{mass}(s_i) \leq l_T$

3 Opis algorytmów

3.1 Generacja populacji rozwiązań

W generowaniu pojedynczego rozwiązania będziemy kierować się następującym algorytmem:

1. Losowe przemieszanie zbioru wejściowego S
2. Przypisanie przedmiotów s_1, s_2, \dots, s_i do zbioru car_items , gdzie:
 - $i = \lfloor n \cdot p \rfloor$
 - p jest zdefiniowane z góry (np. jako *zmienna losowa*), bądź domyślnie wynosi $\frac{p_C}{p_T}$
3. Przypisanie przedmiotów $s_{i+1}, s_{i+1}, \dots, s_n$ do zbioru $truck_items$
4. Przeniesienie ze zbioru car_items do zbioru $truck_items$ przedmiotów, dla których $mass(s) > p_C$
5. Wewnątrz zbiorów car_items oraz $truck_items$ podział na przejazdy wg *naiwnego algorytmu* (opisany poniżej)

Ponieważ krok 1. powyższego algorytmu zawiera element losowy, każde wygenerowane w ten sposób rozwiązanie jest inne. Co więcej, możemy zmieniać parametr p , który również w znacznym stopniu wpływa na rozwiązanie.

Algorytm naiwny do podziału przedmiotów na przejazdy:

Dane wejściowe:

S – zbior przedmiotów

c – ładownosc pojazdu

function divide(S , c):

```
trips = array()
current_trip_items = array()
current_capacity = c

for s in S:
    if current_capacity - mass(s) >= 0:
        current_trip_items.append(s)
    else:
        trips.append(current_trip_items)
        current_trip_items = array()
        current_trip_items.append(s)
        current_capacity = c
    end if

current_capacity -= mass(s)
```

```

end for

if length(current_trip_items) > 0:
    trips.append(current_trip_items)
end if

return trips
end function

```

3.2 Algorytm genetyczny

3.2.1 Reprezentacja genetyczna

Pojedynczy podział zbioru wejściowego S , $P = \{P_1, P_2, \dots, P_m\} \subset 2^S$, można reprezentować jako jeden chromosom, w którym genami są poszczególne podzbiory razem z informacją, jaki transport jest wymagany do ich przewozu.

Taki podział wymaga metody uporządkowania, której definicja zbioru nie zapewnia. Ten problem będzie rozwiązany przez określenie względnej sprawności genu.

3.2.2 Funkcja sprawności

Inspirując się [1], definiujemy funkcję sprawności zarówno dla całego rozwiązania, jak i pojedynczego genu. Sprawność rozwiązania będzie liczona jako zwykła przeciwność funkcji kosztu (tj. $fitness(P) = -cost(P)$ (może być ona ujemna lecz to nie jest istotne - zależy nam na jej maksymalizacji), natomiast sprawność genu jako dwie podobne funkcje:

$$\begin{aligned}
massFitness(P_i, trans_i) &= \begin{cases} \frac{mass(P_i)}{l_C} & trans_i = car \wedge mass(P_i) < l_C \\ \frac{mass(P_i)}{l_T} & trans_i = truck \wedge mass(P_i) < l_T \\ 0 & \text{w p.w.} \end{cases} \\
costFitness(P_i, trans_i) &= \begin{cases} \frac{mass(P_i)}{p_C} & trans_i = car \wedge mass(P_i) < l_C \\ \frac{mass(P_i)}{p_T} & trans_i = truck \wedge mass(P_i) < l_T \\ 0 & \text{w p.w.} \end{cases}
\end{aligned}$$

Definicja funkcji sprawności pozwala nam zdefiniować kolejność genów w chromosomie potrzebną do fazy krzyżowania. Obie funkcje są wykorzystywane ponieważ mają odmienne pożądane cechy:

- Funkcja costFitness ma lepszą szansę na korelację ze sprawnością całego rozwiązania i wykorzystuje więcej informacji o problemie,
- Funkcja massFitness zawsze ma wartości z przedziału $[0, 1]$.

3.2.3 Populacja początkowa

Populację początkową wygenerujemy za pomocą algorytmu opisanego w punkcie 3.1. Rozmiar początkowej (*population_size*) populacji ustawimy domyślnie na 100.

3.2.4 Selekcja

Algorytmów selekcji jest wiele, od tych bardziej losowych (np. algorytm ruletki) do tych bardziej deterministycznych (np. metoda rankingowa). My wykorzystamy tę drugą, ponieważ lepiej radzi sobie z odróżnianiem silnych osobników od słabych:

- Metoda rankingowa - Dla każdego chromosomu obliczamy sprawność, a następnie sortujemy je malejąco według niej. Ostatecznie wybieramy $\lfloor \frac{population_size}{10} \rfloor$ najlepszych osobników, które będą stanowić trzon następnej populacji. Pozostałe osobniki odrzucamy.

3.2.5 Krzyżowanie

Algorytm krzyżowania jest oparty na algorytmie *eager breeder*[1], z jedną modyfikacją która ma na celu zmniejszyć ilość przcy przy naprawianiu chromosomu potomka. Podczas krzyżowania dopuszczamy zmianę metody transportu zapisaną w genie. W tej fazie przez "sprawność genu" rozumiemy funkcję costFitness

Faza właściwego krzyżowania

1. Do dziecka dodaj wszystkie geny identyczne u obojga rodziców,
2. Dla każdego rodzica dodaj sprawniejszą połowę genów niedodanych w 1-szym kroku

Faza naprawy chromosomu

1. Dla każdego przedmiotu, który pojawia się w 2 różnych genach usuń go z genu o mniejszej sprawności; jeśli oba geny mają identyczną sprawność, usuń go z losowego
2. Dla każdego przedmiotu, który nie pojawia się:
 - dodaj go do najsprawniejszego genu, którego sprawność w wyniku takiej operacji wzrośnie
 - jeśli nie jest to możliwe, dodaj go do najmniej sprawnego genu, w którym przedmiot się zmieści
 - jeśli i to nie jest możliwe, umieść go w nowym genie

3.2.6 Mutacja

Algorytm mutacji polega na przejściu (być może kilka razy) do sąsiedniego rozwiązania. Przez "sprawność genu" rozumiemy tutaj funkcję `massFitness`.

- Wybieramy losowy gen stosując wagi odwrotnie proporcjonalne do sprawności
- Z podzbioru danego genu wybieramy losowy przedmiot
- Przedmiot ten przenosimy do innego losowego genu wg tych samych zasad, co przy naprawie po krzyżowaniu, z pominięciem genu z którego go usunęliśmy

3.3 Algorytm rojowy (pszczoły)

3.3.1 Streszczenie kodu

```
inicjalizacja_skautow()  
inicjalizacja_lak()  
while(not warunek_stopu):  
    wyszukiwanie_lokalne()  
    zmniejszanie_lak()  
    wyszukiwanie_globalne()
```

3.3.2 Inicjalizacja skautów

Dla naszych potrzeb skaut to po prostu rozwiązanie naszego problemu dlatego inicjalizacja polega na wygenerowaniu pożądanej liczby losowych rozwiązań.

3.3.3 Inicjalizacja łąk

Dla każdego wygenerowanego skauta generujemy łąkę (rozwiązania podobne do rozwiązania skauta). Nowe rozwiązania tworzymy poprzez wybranie przedmiotów które zmieniają swoje miejsce (każdy przedmiot ma takie samo prawdopodobieństwo aby zostać wybrany równe mp). Po wybraniu przedmiotów usuwamy je z rozwiązania skauta po czym dodajemy je w losowych miejscach tak aby ograniczenia problemu były spełnione.

3.3.4 Wyszukiwanie lokalne

W wyszukiwaniu lokalnym wybieramy ns najlepszych skautów dla których losujemy będziemy losować rozwiązania z łąk które zostały przez nich wygenerowane. Jeżeli spośród wylosowanych rozwiązań znajdziemy rozwiązanie lepsze od skauta który wygenerował łąkę znalezione rozwiązanie staje się nowym skautem. Dla ne najlepszych skautów ($ne < ns$) liczba losowanych rozwiązań to nre natomiast dla reszty spośród ns skautów liczba losowanych rozwiązań to nrs ($nrs < nre$).

3.3.5 Zmniejszanie łąk

Jeżeli w trakcie wyszukiwania lokalnego nie znaleźliśmy lepszego rozwiązania zmniejszamy łąkę o rn rozwiązań.

3.3.6 Wyszukiwanie globalne

Skauci którzy nie byli ns najlepszymi skautami są zastępowaniu przez nowe losowe rozwiązania.

3.3.7 Funkcja sprawności

Funkcja sprawności jest zdefiniowana analogicznie jak dla algorytmu genetycznego (3.2.2)

3.3.8 Populacja początkowa

Populację początkową wygenerujemy za pomocą algorytmu opisanego w punkcie 3.1. Rozmiar początkowej (*population_size*) populacji ustawimy domyślnie na 100.

4 Uruchamianie aplikacji

4.1 Wymagania

python3 w wersji co najmniej 3.8

4.2 Uruchamianie

Aby uruchomić aplikację należy wykonać poniższe komendy w terminalu:

```
git clone https://github.com/MikeyZat/badania-operacyjne.git
cd badania-operacyjne
pip3 install requirements.txt
python3 main.py [infile] [options]
```

, gdzie *infile* to ścieżka do pliku z danymi testowymi (domyślnie ex.json), a dostępne opcje to:

- -n - Ilość pokoleń, które chcemy wygenerować (domyślnie 1000)
- -u - Ilość pokoleń z rzędu bez poprawy, po których powinniśmy się zatrzymać (domyślnie 200)
- -k - Rozmiar populacji w każdym pokoleniu (domyślnie 100)

5 Eksperymenty

5.1 Dane testowe

Algorytm będziemy testować na czterech zestawach danych wejściowych:

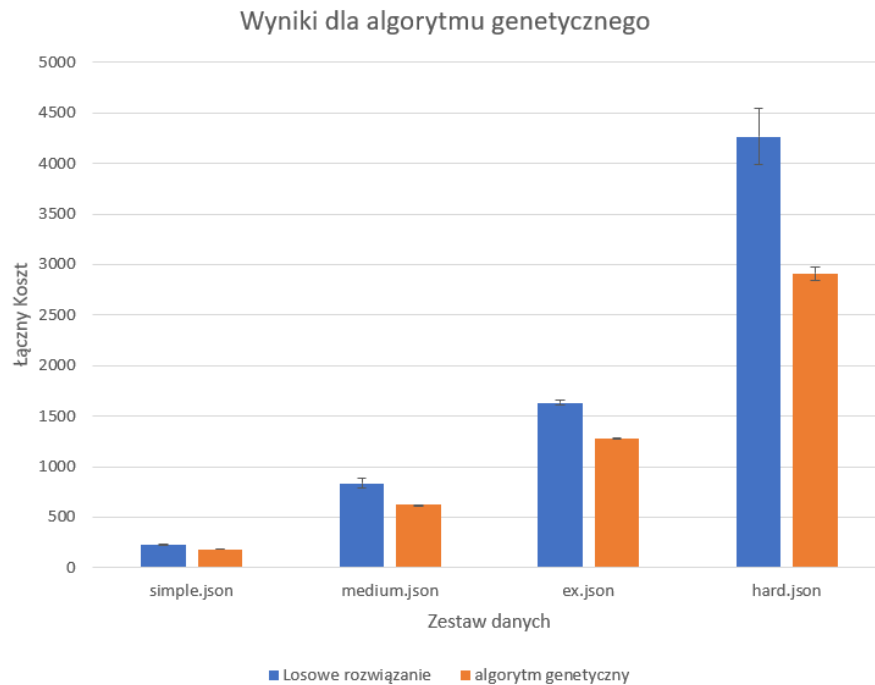
1. **simple.json** - $l_T = 40, l_C = 10, p_T = 50, p_C = 15, N = 15$
2. **medium.json** - $l_T = 30, l_C = 15, p_T = 60, p_C = 10, N = 50$
3. **ex.json** - $l_T = 30, l_C = 15, p_T = 60, p_C = 10, N = 100$
4. **hard.json** - $l_T = 100, l_C = 40, p_T = 150, p_C = 20, N = 200$

5.2 Rozwiązania losowe

Aby mieć jakikolwiek punkt odniesienia, będziemy losować rozwiązanie i zapisywać jego koszt. Takie algorytm jest oczywiście bardzo niedeterministyczne i mało skuteczne, ale pomoże nam w ocenie skuteczności pozostałych algorytmów. Z uwagi na losowość rozwiązania, do każdego zestawu testowego wykonamy 10 prób i jako wynik zapiszemy ich średnią, wraz z odchyleniem standardowym.

5.3 Algorytm genetyczny

Dla danych testowych wykonaliśmy po 10 prób i jako wyniki przedstawiliśmy średnią z pomiarów wraz z odchyleniem standardowym:



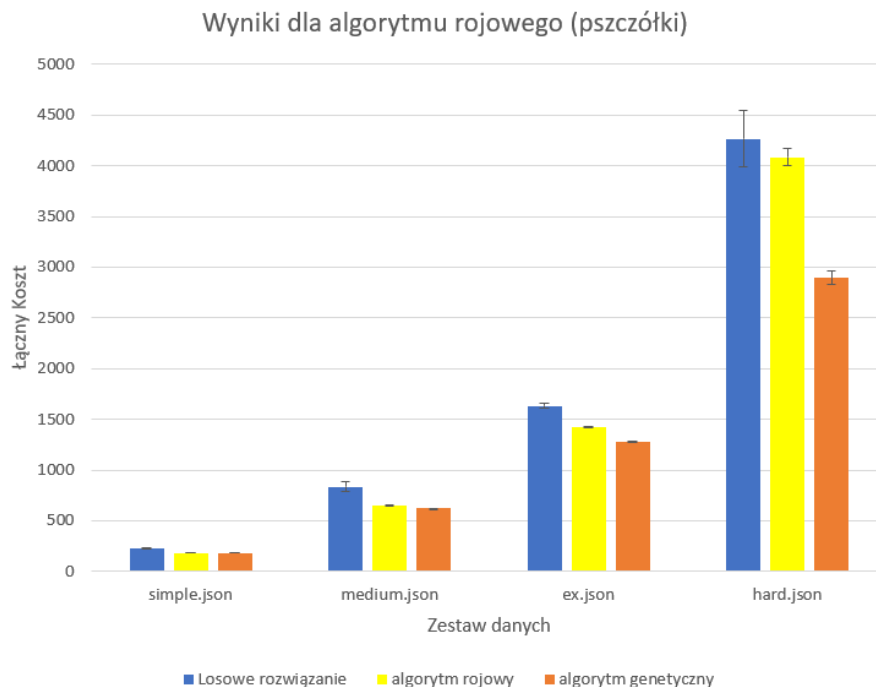
Rysunek 1: Porównanie wyników algorytmu genetycznego i losowych rozwiązań.

Komentarz

Jak widać, algorytm genetyczny dał zdecydowanie lepszy rezultat niż losowe rozwiązanie. Co więcej, odchylenie standardowe było mniejsze (w niektórych przypadkach algorytm kilkakrotnie zwracał ten sam, być może optymalny, wynik). Można zatem stwierdzić, że algorytm genetyczny okazał się być dużo bardziej deterministyczny. Warto tutaj nadmienić także, że algorytm losowy radzi sobie nie najgorzej i rozrzut wyników także nie jest dla niego bardzo duży. Dzieje się tak dlatego, że tylko permutacja przedmiotów jest całkowicie losowa, w następnym kroku algorytmem naiwnym przedmioty przydzielane są do przejazdów w miarę optymalny sposób.

5.4 Algorytm rojowy (pszczoły)

Dla danych testowych wykonaliśmy po 10 prób i jako wyniki przedstawiliśmy średnią z pomiarów wraz z odchyleniem standardowym:



Rysunek 2: Porównanie wyników algorytmu rojowego, genetycznego oraz losowych rozwiązań.

Komentarz

Jak widać, algorytm pszczółkowy dał gorsze wyniki niż genetyczny, jednak lepsze niż losowe rozwiązanie. Można również zaobserwować, że dla łatwiejszych przykładów był bardzo bliski wyników genetycznego, jednak dla najtrudniejszego przykładu różnica była już bardzo wyraźna. Po analizie wyników, nasuwa się wniosek, że algorytm ten nie jest zbyt dobrze przystosowany do naszego problemu - jego przewaga nad losowym rozwiązaniem jest dość niewielka.

6 Podsumowanie

Dzięki pracy nad naszym problemem poznaliśmy algorytmy służące do aproksymacji rozwiązania, które są szczególnie przydatne w przypadku gdy otrzymanie optymalnego rozwiązania jest niemożliwe do uzyskania w rozsądnym czasie. Dzięki porównaniu algorytmu pszczółek oraz genetycznego zauważyliśmy, że wybór algorytmu należy uzależnić od problemu który chcemy rozwiązać.

Uwagi dotyczące naszego problemu:

- Algorytm genetyczny, pomimo długich czasów dla pojedynczych pokoleń, był w stanie w niewielu pokoleniach dotrzeć do rozwiązań zauważalnie

lepszich od losowych

- Algorytm krzyżowania *eager breeder*[1], pomimo że był zaprojektowany dla innego problemu, dobrze sprawdził się w naszym przypadku
- Żaden z algorytmów nie dał porażająco lepszych wyników niż wybór losowy, lecz w obydwu przypadkach różnica była statystycznie wyraźna (z wyjątkiem algorytmu pszczelego dla największej instancji)
- Wydaje się, że przewaga algorytmu genetycznego nad pszczółkowym wynika z bardziej adekwatnego kryterium poprawy rozwiązania. Algorytm genetyczny skutecznie eliminuje ślepe uliczki i używa do poprawy rozwiązania rozwiązań aktualnie najlepszych. Algorytm pszczółkowy wydaje się nadmiernie losowy i podatny na ślepe, choć na pewnym etapie obiecujące, uliczki.

7 Bibliografia

- [1] W. A. Greene. Genetic algorithms for partitioning sets. *International Journal on Artificial Intelligence Tools*, 10, 2001.
- [2] D. C. Porumbel, J. K. Hao, and P. Kuntz. An efficient algorithm for computing the distance between close partitions. *Discrete Applied Mathematics*, 159, Styczeń 2011.