Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Ewolucyjny algorytm dla nieliniowego zadania transportowego

Piotr Berezowski
Nr indeksu: 236749

Praca inżynierska napisana
pod kierunkiem
dr hab. Pawła Zielińskiego

Politechnika
Wrocławska
Wrocław 2019

Spis treści

Wstęp

Analiza problemu

2.1 Zadanie transportowe

Zadanie transportowe możemy zaliczyć do grupy zadań optymalizacyjnych z ograniczeniami. Rozwiązując je, staramy się przy użyciu n punktów nadania zaspokoić zapotrzebowanie m punktów odbioru w taki sposób, aby całkowity koszt transportu był minimalny. Zadanie wymaga określenia ilości towaru znajdującej się w każdym z punktów nadania, oraz zapotrzebowania na towar w każdym z punktów odbioru. Dodatkowo musimy określić koszt transportu pomiędzy poszczególnymi punktami. Klasyczne zadanie transportowe ogranicza się do transportu tylko jednego rodzaju towaru, dzięki czemu punkty odbioru mogą być zaopatrywane przez jeden lub więcej punktów nadania.

Zadanie transportowe nazywamy zbilansowanym, jeśli całkowita podaż towaru jest równa całkowitemu popytowi. W przeciwnym wypadku zadanie jest niezbilansowane. Rozwiązywanie zadania niezbilansowanego polega na sprowadzeniu go do zadania zbilansowanego, poprzez dodanie fikcyjnego dostawcy(w przypadku większego popytu), lub fikcyjnego odbiorcy(w przypadku większej podaży). Koszt transportu między fikcyjnym dostawcą a odbiorcami, lub między dostawcami a fikcyjnym odbiorcą najczęściej ustalany jest jako zerowy.

Załóżmy, że mamy n punktów nadania i m punktów odbioru. Początkowa ilość towaru w i-tym punkcie nadania jest równa $supply(i)$, a początkowe zapotrzebowanie w j-tym punkcie odbioru jest równe $demand(j)$. Jeśli x_{ij} jest ilością towaru dostarczanego przez i-ty punkt nadania do j-tego punktu odbioru, to zbilansowane zadanie transportowe możemy zdefiniować w następujący sposób:

$$min \sum_{i=1}^{n} \sum_{j=1}^{m} f_{ij}(x_{ij})$$

Przy spełnionych ograniczeniach:

$$\sum_{j=1}^{m} x_{ij} = supply(i), \text{ dla } i = 1, 2, \ldots, n$$

$$\sum_{i=1}^{n} x_{ij} = demand(j), \text{ dla } j = 1, 2, \ldots, m$$

$$x_{ij} \geqslant 0, \text{ dla } i = 1, 2, \ldots, n \text{ i } j = 1, 2, \ldots, m$$

Pierwszy zestaw ograniczeń mówi o tym, że całkowita ilość towaru transportowana z pojedynczego punktu nadania musi być równa jego początkowej ilości znajdującej się w tym punkcie. Z kolei drugi zestaw mówi o tym, że całkowita ilość towaru transportowana do pojedynczego punktu odbioru musi być równa jego początkowemu zapotrzebowaniu. W przypadku zadania niezbilansowanego równości w dwóch pierwszych zestawach ograniczeń należy zmienić na odpowiednie nierówności.

Zadanie jest liniowe, jeśli koszt transportu między punktami nadania i odbioru jest wprost proporcjonalny do ilości transportowanego towaru, tzn. jeśli $f_{ij}(x_{ij}) = cost_{ij}x_{ij}$, gdzie $cost_{ij}$ jest jednostkowym kosztem transportu między i-tym punktem nadania, a j-tym punktem odbioru. W przypadku kiedy zależność między kosztem transportu, a ilością transportowanego towaru wygląda inaczej mówimy o zadaniu nieliniowym.

2.1.1 Wersja liniowa

Zadanie transportowe w wersji liniowej możemy przedstawić jako problem programowania liniowego. Optymalne rozwiązanie możemy więc wyznaczyć przy pomocy znanych metod używanych przy tej klasy problemach, takich jak np. metoda sympleks, której opis znajduje się w sekcji *Klasyczne rozwiązania*.

$$[...TODO : ...]$$

2.1.2 Wersja nieliniowa

O ile liniowa wersja zadania jest stosunkowo łatwa w rozwiązaniu, o tyle dla wersji nieliniowej nie ma ogólnej metody rozwiązywania. Należy ono do problemów z kategorii NP-trudnych [TODO: - znalezc publikacje]. Do jego rozwiązania używa się różnych algorytmów wyznaczających rozwiązania przybliżone, takich jak algorytmy metaheurystyczne.

Wersja nieliniowa lepiej oddaje rzeczywisty problem planowania dostaw, gdzie koszty zależą od wielu czynników, które wpływają na to, że zależność między kosztem, a ilością transportowanego towaru nie jest liniowa. Niniejsza praca skupia się głównie na takich problemach.

2.2 Klasyczne rozwiązania

[TODO: Algorytm Sympleks]

2.3 Algorytmy metaheurystyczne

Metaheurystyki są to algorytmy, które definiują sposób w jaki ma być przeszukiwany zbiór dopuszczalnych rozwiązań zdefiniowanego problemu. Znajdują one zastosowanie w przypadkach kiedy nie znamy algorytmów, które wyznaczają rozwiązanie dokładne lub ich koszt jest zbyt duży. Do tego typu problemów możemy zaliczyć rozważane tutaj nieliniowe zadanie transportowe. Minusem stosowania algorytmów metaheurystycznych jest fakt, że nie dają one gwarancji na znalezienie wystarczająco dobrego rozwiązania.

2.3.1 Algorytmy ewolucyjne

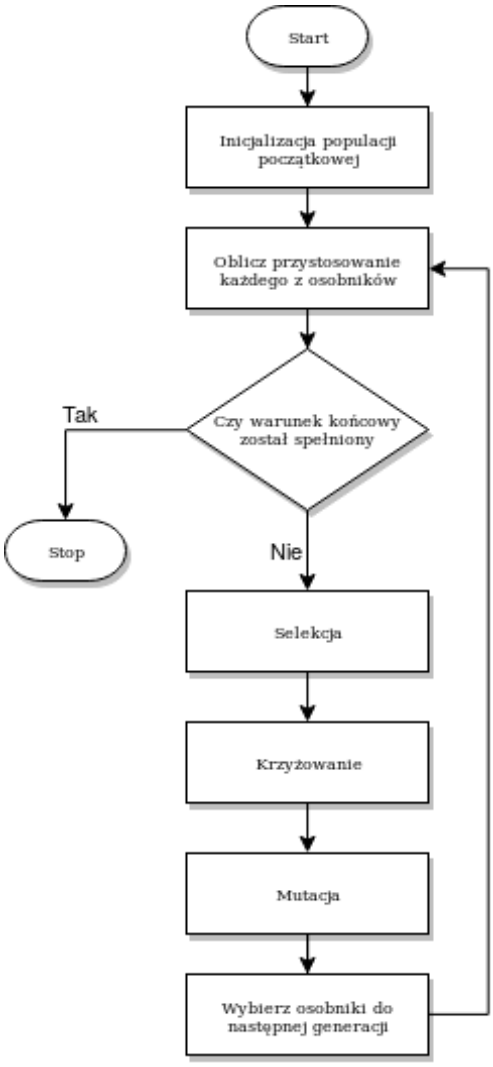
Algorytmy ewolucyjne stanowią podzbiór algorytmów metaheurystycznych. Sposób ich działania jest inspirowany przez zjawisko ewolucji występujące w naturze. Działają one na podzbiorach przestrzeni wszystkich rozwiązań, które nazywamy **populacjami**. Przy użyciu specjalnie zdefiniowanych operatorów, na podstawie jednej populacji tworzona jest kolejna zawierająca w sobie lepsze rozwiązania, nazywane dalej **chromosomami** lub **osobnikami**.

Na początku działania algorytmu generowana jest w sposób losowy populacja startowa. Procedurę generowania pojedynczego osobnika lub całej populacji nazywać będziemy **inicjalizacją**. Następnie na przestrzeni pokoleń(iteracji algorytmu) populacja ewoluuje generując coraz lepsze rozwiązania. W każdej iteracji pewna część osobników zostaje wybrana do reprodukcji. Procedurę wyboru rodziców do reprodukcji nazywać będziemy **selekcją**. Wybrane osobniki krzyżujemy między sobą, tworząc w ten sposób nowe, posiadające cechy wybranych wczesniej rodziców. Następnie losowo wybrane osobniki ulegają mutacji. Ostatecznie z otrzymanych osobników tworzona jest nowa populacja, która będzie stanowić bazę dla kolejnej iteracji algorytmu. Algorytm kończy działanie w momencie kiedy zostanie spełniony warunek końcowy, którym może być np. wygenerowanie wystarczająco dobrego rozwiązania lub przejście określonej liczby iteracji. Ogólna schemat

działania algorytmu ewolucyjnego przedstawiono w postaci pseudokodu 2.1 oraz schematu blokowego 2.1.

Pseudokod 2.1: Ogólny schemat działania algorytmu ewolucyjnego

```
1  P(t): Populacja w t-tej iteracji algorytmu;
2  O(t): Populacja dzieci w t-tej iteracji algorytmu;
3  t ← 0;
4  inicjalizacja P(t);
5  while Warunek końcowy nie został spełniony do
6  |   parents ← selekcja z P(t);
7  |   O(t + 1) ← zastosuj operator krzyżowania na parents;
8  |   O(t + 1) ← zastosuj operator mutacji na O(t + 1);
9  |   P(t + 1) ← wybierz osobniki do następnej generacji z O(t + 1);
10 |   t ← t + 1;
11 return najlepszy osobnik z P(t);
```



Rysunek 2.1: Ogólny schemat działania algorytmu ewolucyjnego

Projektowanie algorytmu ewolucyjnego możemy podzielić na kilka oddzielnych części, są to:

- **Reprezentacja** - określa sposób zakodowania rozwiązania w chromosomie(osobniku). Wybór repre-

zentacji chromosomu jest bardzo ważnym etapem projektowania algorytmu. Odpowiednia reprezentacja może w znacznym stopniu wpłynąć na szybkość i jakość rozwiązań znajdowanych przez algorytm, ponieważ to ona w dużej mierze określa sposób w jaki przeszukiwana będzie przestrzeń rozwiązań zadania. Jako reprezentacje bardzo często stosowane są wektory lub macierze genów, gdzie gen może być pojedynczą liczbą całkowitą lub rzeczywistą. Oczywiście jako sposób reprezentacji rozwiązania możemy wybrać dowolną strukturę danych, należy jednak pamiętać, że zdefiniowane później operacje mutacji i krzyżowania muszą być dostosowane do wybranej struktury.

- **Funkcja oceny** - określa stopień przystosowania danego osobnika. Bardzo często funkcja oceny jest równoważna funkcji celu, którą nasz algorytm ma minimalizować/maksymalizować, nie jest to jednak regułą.
- **Operator krzyżowania** - jest jednym z operatorów używanych do generowania kolejnego pokolenia w algorytmach ewolucyjnych. Z założenia przyjmuje on jako argumenty dwa lub więcej rozwiązań(rodziców) i generuje na ich podstawie nowe(dzieci), które łączą w sobie cechy rodziców. [TODO: przykład]
- **Operator mutacji** - jest drugim z operatorów używanych do generowania następnych pokoleń w algorytmach ewolucyjnych. Jego celem jest poszerzene obszaru przeszukiwanych rozwiązań. Ten operator powinien wprowadzać minimalną zmianę w rozwiązaniu, co zapobiega zbyt szybkiej zbieżności algorytmu i pozwala na wprowadzenie dodatkowej różnorodności w populacji. Należy pamiętać o tym, że wprowadzana zmiana nie może być za duża, bo może to prowadzić do odwrotnego rezultatu, czyli zamiast różnicować rozwiązania nasz operator może je niszczyć. [TODO: przykład]
- **Selekcja** - określa sposób wyboru rodziców na których użyjemy operatora krzyżowania. Istnieje wiele opisanych metod selekcji[3] takich jak np. metoda koła ruletki czy metoda rankingowa. Przy tworzeniu procedury selekcji należy pamiętać o tym, że rozwiązania lepiej przystosowane powinny mieć większe szanse na zostanie rodzicami dla kolejnego pokolenia. Zapewnia to większe szanse na wygenerowanie lepszych dzieci do następnej generacji. [TODO: przyklad]
- **Wybór następnego pokolenia** - ostateczny krok algorytmu, w którym wybieramy które osobniki wejdą w skład populacji początkowej w kolejnej iteracji algorytmu. Podstawową składową tej populacji powinny być oczywiście osobniki wygenerowane za pomącą krzyżowania. Często stosowaną praktyką jest również przepisywanie części najlepszych rozwiązań oraz kilku losowo wybranych z poprzedniego pokolenia.
- **Parametry algorytmu** - do standardowych parametrów należą wielkość populacji, prawdopodobieństwo krzyżowania oraz prawdopodobieństwo mutacji. Odpowiedni dobór parametrów ma kluczowe znaczenie dla efektywności oraz szybkości algorytmu.

Równoległą implementacja algorytmu ewolucyjnego

3.1 Użyte technologie

Do implementacji algorytmu zastosowano język Julia[1] w wersji 1.3. Julia jest stosunkowo nowym językiem programowania. Został zaprojektowany z myślą o zastosowaniach w obliczeniach numerycznych i analizie danych. Łączy on w sobie zalety języków niskopoziomowych i wysokopoziomowych takie jak szybkość i czytelność kodu. Testy pokazują, że program napisany w Julii może być równie szybki, jak odpowiadający mu program napisany w C[2]. Dodatkową zaletą jest możliwość bezpośredniego wywoływania bibliotek napisanych w C, Fortranie i kilku innych językach popularnych w dziedzinie obliczeń numerycznych bezpośrednio z Julii.

Ostatnie aktualizacje w znaczącym stopniu rozwinęły wsparcie języka dla obliczeń równoległych i rozproszonych. W tym momencie Julia oferuje wsparcie dla równoległości na poziomie wątków jak i procesów, co dodatkowo wpłynęło na wybór tego właśnie języka. Posiada własny protokół komunikacji między procesami, jednak istnieje również biblioteka implementująca najbardziej powszechny protokół MPI.

Julia jest językiem kompilowanym. Używa kompilatora JIT(just-in-time), który kompiluje program tuż przed jego wykonaniem, dzięki czemu jest szybsza niż języki interpretowane. Należy pamiętać o tym, że nie każdy program napisany w Julii będzie szybki. Wszystko zależy od jakości dostarczonego kodu. Głównym czynnikiem, który wpływa na szybkość są typy. Julia jest językiem dynamicznie typowanym, jednak podczas kompilacji tworzone są warianty tej samej funkcji dla różnych typów(o ile to możliwe). Pozwala to pominąć kontrolę typów podczas wykonywania kodu i tym samym znacząco przyspieszyć jego działanie. Dlatego pisząc kod w julii powinniśmy pamiętać o tym, żeby unikać miejsc, w których kompilator będzie zmuszony do konwersji zmiennych do konkretnego typu. Aby identyfikować tego typu miejsca możemy używać dostarczonych w bibliotece standardowej narzędzi, które pomagają analizować nasz kod. Warte wymienienia są tutaj:

- pakiet $Profile$, który zbiera informacje o czasie wywołania kolejnych fragmentów kodu, dzięki czemu w łatwy sposób możemy zidentyfikować fragmenty do dalszej optymalizacji. Pozwala on też śledzić liczbę ilość pamięci alokowanej przez konkretne fragmenty kodu, co również w wielu przypadkach może okazać się przydatną informacją.
- makro @$code\_warntype$, które zwraca strukturę AST(abstract syntax tree) dla wykonywanego kodu, dzięki czemu możemy zobaczyć możliwe typy dla wszystkich zmiennych. Dodatkowo miejsca w których kompilator nie jest w stanie jednoznacznie określić typu danej zmiennej jest zaznaczony na czerwono.

Julia udostępnia też środowisko uruchomieniowe $REPL$(read-eval-print loop), dzięki któremu możemy w bardzo łatwy sposób testować napisany kod. Dzięki dostępnym bibliotekom takim jak *Debugger.jl* oraz *Rebugger.jl* możemy w razie potrzeby debugować napisany kod z poziomu $REPL$ co znacznie przyspiesza znajdowanie błędów.

3.2 Reprezentacja chromosomu

W opisywanej implementacji jako reprezentacje rozwiązania przyjęto macierz:

$$V = (v_{ij}), \text{ gdzie } 1 \leqslant i \leqslant length(supply) \land 1 \leqslant j \leqslant length(demand)$$

Rozwiązanie jest zakodowane w taki sposób, że komórka macierzy o indeksie $[i, j]$ określa ilość transportowanego towaru między i-tym punketm nadania i j-tym punktem odbioru. Jest to jedna z najbardziej naturalnych reprezentacji rozwiązania dla zadania transportowego.

i\j	10.0	7.0	5.0	13.0	12.0
12.0	0.0	7.0	5.0	0.0	0.0
10.0	5.0	0.0	0.0	0.0	5.0
3.0	3.0	0.0	0.0	0.0	0.0
10.0	0.0	0.0	0.0	3.0	7.0
12.0	2.0	0.0	0.0	10.0	0.0

Tablica 3.1: Przykładowe rozwiązanie(pierwszy wiersz - wektor punktów odbioru, pierwsza kolumna - wektor punktów nadania).

Aby ograniczenia zadania zostały zachowane macierz rozwiązania musi spełniać następujące warunki:

$$\sum_{j=1}^{m} v_{ij} = supply[i], \text{ dla } i = 1, 2, \ldots, n, \text{ gdzie } n = length(supply)$$

$$\sum_{i=1}^{n} v_{ij} = demand[j], \text{ dla } j = 1, 2, \ldots, m, \text{ gdzie } m = length(demand)$$

$$v_{ij} \geqslant 0, \text{ dla } i = 1, 2, \ldots, n \text{ i } j = 1, 2, \ldots, m$$

3.3 Inicjalizacja chromosomu

Projektując procedurę inicjalizacji rozwiązania musimy pamiętać o tym, żeby generowane rozwiązania spełniały ograniczenia przedstawione w poprzedniej sekcji oraz obejmowały jak największą część przestrzeni wszystkich rozwiązań. Zaproponowana procedura przyjmuje jako argumenty wektory popytu i podaży. Iterując po kolejnych, losowych komórkach macierzy przypisujemy im wartość $val = min(supply[i], demand[j])$, gdzie i, j są indeksami wylosowanej komórki macierzy, a $supply[i]$ oraz $demand[j]$ odpowiadającymi im wartościami w wektorach popytu i podaży. Następnie zmniejszamy wartości w wektorach o wpisaną wartość val. W ten sposób ograniczenia zadania zostają spełnione. Wygenerowane rozwiązania są wierzchołkami sympleksu, opisującego wypukły brzeg przestrzeni dopuszczalnych rozwiązań.

Pseudokod 3.1: Procedura inicjalizacji chromosomu

Data: *supply* - wektor popytu rozmiaru n, *demand* - wektor podaży rozmiaru m
Result: V - zainicjalizowana macierz

```
1 V ← zeros(n, m);                    /* generujemy macierz zerową rozmiaru n × m */
2 indices ← lista wszystkich indeksów macierzy V w losowej kolejności;
3 for (s, d) ∈ indices do
4     val ← min(demand[d], supply[s]);
5     demand[d] ← demand[d] − val;
6     supply[s] ← supply[s] − val;
7     V[s, d] ← val;
8 return V
```

[TODO: Przykład zastosowania]

3.4 Operator krzyżowania

Operator krzyżowania został zdefiniowany jako kombinacja wypukła dwóch rodziców. W ten sposób w wyniku jednego krzyżowania powstają dwa nowe rozwiązania.

Pseudokod 3.2: Operator krzyżowania

Data: P_1, P_2 - rodzice wybrani do krzyżowania
Result: O_1, O_2 - otrzymane dzieci

```
1 c_1 ← rand(0,...,1);          /* losujemy liczbę z przedziału [0,1] */
2 c_2 ← 1.0 − c_1;
3 O_1 ← c_1 * P_1 + c_2 * P_2;
4 O_2 ← c_2 * P_1 + c_1 * P_2;
5 return (O_1, O_2);
```

Zdefiniowany tak operator krzyżowania nie narusza ograniczeń zadania, ponieważ przestrzeń rozwiązań jest wypukła. Wynika z tego, że jeśli rodzice spełniali ograniczenia, to otrzymane w ten sposób dzieci również muszą spełniać ograniczenia.

[TODO: Przykład zastosowania]

3.5 Operator mutacji

Operator mutacji opiera się na modyfikacji rozwiązania poprzez wybranie z niego podmacierzy i jej ponowną inicjalizacje(patrz 4). Załóżmy, że mamy n punktów nadania i m punktów odbioru. Wybierzmy jako kandydata do mutacji macierz $V = (v_{ij})$, gdzie $1 \leqslant i \leqslant n$ i $1 \leqslant j \leqslant m$. Podmacierz $W = w_{ij}$ jest tworzona w następujący sposób:

- Losujemy podzbiór k indeksów $\{i_1, \ldots, i_k\}$ ze zbioru $\{1, \ldots, n\}$ oraz podzbiór l indeksów $\{j_1, \ldots, j_l\}$ ze zbioru $\{1, \ldots, m\}$, $2 \leqslant k \leqslant n$ i $2 \leqslant l \leqslant m$.

- Tworzymy podmacierz W składającą się z takich elementów macierzy V, które zostały wylosowane, tzn. element $v_{ij} \in V$ zostaje włączony do podmacierzy W tylko jeśli $i \in \{i_1, \ldots, i_n\}$ oraz $j \in \{j_1, \ldots, j_l\}$.

Dla stworzonej macierzy W tworzymy nowe wektory $demand_W$ i $supply_W$ w następujący sposób:

$$supply_W[i] = \sum_{j \in \{j_1, \ldots, j_l\}} v_{ij}, \text{ dla } 1 \leqslant i \leqslant k$$

$$demand_W[j] = \sum_{i \in \{i_1, \ldots, i_k\}} v_{ij}, \text{ dla } 1 \leqslant j \leqslant l$$

Następnie na nowo inicjalizujemy podmacierz W macierzy V używając stworzonych wektorów $demand_W$ oraz $supply_W$ do określenia popytu i podaży w wybranych punktach nadania i odbioru. Po zakończeniu

inicjalizacji przepisujemy wartości z podmacierzy W z powrotem w odpowiadające miejsca macierzy V.

Pseudokod 3.3: Operator mutacji

Data: V - osobnik wybrany do mutacji wielkości $n \times m$, k, l - wielkość podmacierzy
Result: V - osobnik po mutacji

$supply\_idx \leftarrow$ wylosuj podzbiór długości k ze zbioru $\{1, \ldots, n\}$;
$demand\_idx \leftarrow$ wylosuj podzbiór długości l ze zbioru $\{1, \ldots, m\}$;
$W \leftarrow zeros(k, l)$; /* generujemy macierz zerową rozmiaru $k \times l$ */
for $i \in \{1, \ldots, k\}$ **do**
    for $j \in \{1, \ldots, l\}$ **do**
        $W[i, j] \leftarrow V[demand\_idx[j], supply\_idx[i]]$;
$supply\_vec \leftarrow zeros(k)$; /* generujemy wektor zerowy długości k */
$demand\_vec \leftarrow zeros(l)$; /* generujemy wektor zerowy długości l */
for $i \in supply\_idx$ **do**
    $supply\_vec[i] \leftarrow \sum_{j \in demand\_idx} V[i, j]$;
for $j \in demand\_idx$ **do**
    $demand\_vec[j] \leftarrow \sum_{i \in supply\_idx} V[i, j]$;
$W \leftarrow inicjalizacja(W, demand\_vec, supply\_vec)$;
for $i \in \{1, \ldots, k\}$ **do**
    for $j \in \{1, \ldots, l\}$ **do**
        $V[demand\_idx[j], supply\_idx[i]] \leftarrow W[i, j]$;
return V

Zdefiniowano dwa operatory mutacji. Różnią się one jedynie procedurą inicjalizacji. W pierwszym używamy tej samej procedury, którą inicjalizujemy nowe chromosomy podczas generowania populacji początkowej(patrz 2). Druga jest modyfikacją tej procedury. Modyfikacja polega na tym, że zamiast wybierać jako wartość pola $val = min(demand[j], supply[i])$ wybieramy liczbę z zakresu $[0, val]$. Zmiana ta powoduje, że otrzymana macierz może naruszać ograniczenia zadania, dlatego po wstępnym wyznaczeniu wartości naprawiamy rozwiązanie poprzez zrobienie wymaganych dodawań w taki sposób, żeby rozwiązanie spełniało ograniczenia.

Poniżej przedstawiono schemat zmodyfikowanej procedury inicjalizacji w postaci pseudokodu.

Pseudokod 3.4: Zmodyfikowana procedura inicjalizacji

Data: $supply$ - wektor podaży rozmiaru n, $demand$ - wektor popytu rozmiaru m
Result: V - zainicjalizowana macierz

$V \leftarrow zeros(n, m)$; /* generujemy macierz zerową rozmiaru $n \times m$ */
$indices \leftarrow$ lista wszystkich indeksów macierzy V w losowej kolejności;
for $(s, d) \in indices$ **do**
    $val \leftarrow$ wartość z przedziału $[0, min(demand[d], supply[s])]$;
    $demand[d] \leftarrow demand[d] - val$;
    $supply[s] \leftarrow supply[s] - val$;
    $V[s, d] \leftarrow val$;
for $(s, d) \in indices$ **do**
    $val \leftarrow min(demand[d], supply[s])$;
    $demand[d] \leftarrow demand[d] - val$;
    $supply[s] \leftarrow supply[s] - val$;
    $V[s, d] \leftarrow V[s, d] + val$;
return V

[TODO: Przykład zastosowania]

3.6 Funkcje oceny

W przypadku omawianego problemu funkcja oceny jest równoważna funkcji celu dla zadania transportowego. Powinna więc mieć postać:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} f(v_{ij})$$

, gdzie $f(v_{ij})$ jest dowolną funkcją przyjmującą jako argument ilość towaru transportowanego między punktami i oraz j.

[TODO: Przykłady funkcji oceny użyte w części eksperymentalnej + wykresy]

3.7 Metoda selekcji

W algorytmie zastosowano standardową metodę selekcji - metodę koła ruletki. W tej metodzie lepiej przystosowane osobniki mają odpowiednio większe szanse na to, że zostaną wybrane do puli rodziców dla następnej generacji. Polega ona na tym, że dla każdego osobnika z populacji przyporządkowujemy odpowiednio duży wycinek koła. Wielkość wycinka zależy od wartości funkcji przystosowania, jaką osiągnął dany chromosom. Następnie losujemy kołem tyle razy, ile rodziców chcemy otrzymać. Metoda ta pozwala na to, że jeden osobnik zostanie wybrany na rodzica kilkukrotnie.

Bardziej formalnie procedura została przedstawiona na poniższym pseudokodzie.

Pseudokod 3.5: Procedura selekcji

Data: *population* - populacja chromosomów, n - ilość rodziców do wybrania
Result: *parents* - wektor wybranych rodziców

```
1  parents ← stwórz pusty wektor długości n;
2  k ← length(population);
3  wheel ← zeros(k);                          /* Generujemy wektor zerowy długości n */
4  total ← Σ_{i=1}^{k} population[i].cost;
5  wheel[1] ← population[1].cost/total;
6  for i ∈ 2, ..., k do
7      wheel[i] ← population[i].cost/total + wheel[i − 1];
8  for i ∈ 1, ..., n do
9      num ← rand(0, ..., 1);                 /* losujemy wycinek koła */
10     selectedIdx ← wybierz pierwszy taki x, że x ⩽ num;
11     parents[i] ← population[selectedIdx];
12 return parents
```

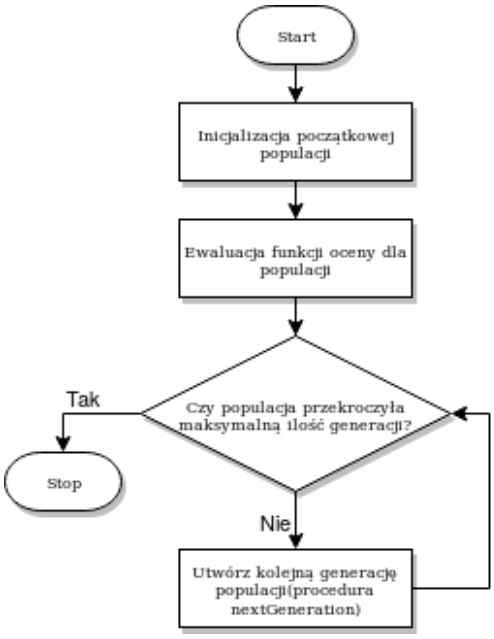
[TODO: obrazek ilustrujący algorytm]

3.8 Wersja równoległa

Algorytm może działać w dwóch trybach:

- Klasycznym
- Wyspowym

Na początku procedura inicjalizacji generuje losową populacje o określonej liczbie osobników i oblicza wartość funkcji oceny dla każdego z nich. Następnie przechodzimy do głównej pętli algorytmu, która kończy się w momencie kiedy populacja osiągnie maksymalną ilość generacji. Każdy z trybów różni się przebiegiem procedury *nextGeneration*(patrz 3.1), która tworzy nową generacje osobników.



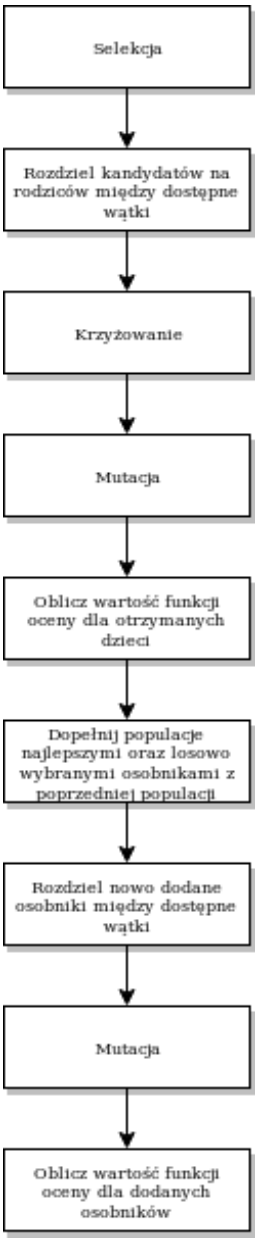
Rysunek 3.1: Przebieg zaimplementowanego algorytmu ewolucyjnego

Model klasyczny nie różni się wiele od standardowego algorytmu ewolucyjnego. Mamy tutaj jedną populacje, która ewoluuje przez określoną na starcie liczbę pokoleń. Wszystkie dostępne parametry algorytmu oraz ich dopuszczalne wartości opisane są w dalczej części pracy, w sekcji *Pliki konfiguracyjne*.

W modelu klasycznym zrównoleglenie odbywa się na poziomie pojedynczej iteracji(patrz 3.2). Ewolucję populacji możemy podzielić tutaj na dwie części:

- **Krzyżowanie** - w tej części między wątki rozdzielani są rodzice wybrani do krzyżowania. Następnie każdy z wątków generuje swoją część dzieci oraz z określonym prawdopodobieństwem stostuje na nich operator mutacji i ostatecznie oblicza dla nich wartość funkcji oceny. Dzieci są dodawane do kolejnej populacji.

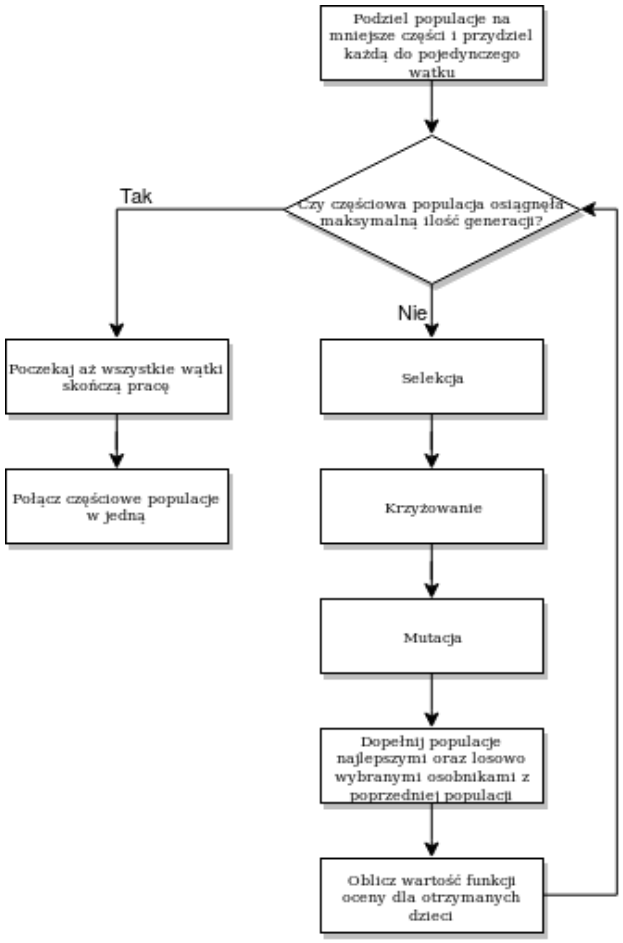
- **Dopełnienie populacji** - w tej części do kolejnej populacji przepisywana jest część najlepszych rozwiązań oraz losowo wybranych osobników z poprzedniej populacji. Następnie te dodane osobniki są poddawane mutacji i obliczana jest dla nich funkcja oceny.



Rysunek 3.2: Przebieg procedury nextGeneration dla modelu klasycznego

Model wyspowy różni się od klasycznego podejścia tym, że całkowita populacja jest tutaj rozdzielana na kilka mniejszych. Następnie każda z populacji częściowych ewoluuje niezależnie od innych przez określoną liczbę pokoleń. Po zakończeniu tego procesu wszystkie częściowe populacje są na nowo łączone w jedną. Następnie najlepsze rozwiązywanie jest zapisywane, a populacja zostaje na nowo podzielona na kilka mniejszych i cały proces się powtarza, do momentu w którym ilość generacji przekroczy określoną na początku liczbę. Na końcu najlepsze znalezione rozwiązanie jest zwracane.

W tym modelu zrównoleglenie obliczeń polega na tym, że przy każdym podziale populacji jeden wątek zarządza pojedynczą częścią populacji(patrz 3.3). Model ten skaluje się lepiej niż model klasyczny ze względu na to że podzadania przydzielane wątkom są większe. Minusem jest tutaj to, że populacja musi być odpowiednio duża, żeby jej podział na mniejsze części się sprawdził.



Rysunek 3.3: Przebieg procedury nextGeneration dla modelu wyspowego

W tym momencie każda z częściowych populacji w modelu wyspowym posiada takie same parametry. [...]

3.9 Pliki konfiguracyjne

Dodatkowo do algorytmu został dodany moduł obsługi plików konfiguracyjnych. Używają one formatu JSON. Moduł pozwala na zapisywanie i wczytywanie całej konfiguracji algorytmu, w skład której wchodzą wektory popytu i podaży, macierz kosztu, oraz wszystkie parametry programu. Definicja poszczególnych parametrów:

- *populationSize* - rozmiar całkowitej populacji. Powinien być dodatnią liczbą całkowitą.
- *eliteProc* - ułamek najlepszych rozwiązań, które zostają przepisane do następnego pokolenia. Wartość powinna znajdować się w przedziale $[0, 1]$. Testy pokazują, że najlepsze rozwiązania są generowane dla wartości parametru w przedziale $[0.1, 0.3]$.
- *mutationProb* - prawdopodobieństwo mutacji. Przyjmuje wartość z zakresu $[0, 1]$. Warto pamiętać o tym, że zalecane prawdopodobieństwo mutacji nie powinno przekraczać kilkanastu procent.
- *mutationRate* - wielkość mutacji, określa stosunek rozmiaru podmacierzy, wybieranej do ponownej inicjalizacji podczas mutacji, do macierzy rozwiązania. Przyjmuje wartości z zakresu $[0, 1]$. Tak jak w przypadku prawdopodobieństwa mutacji, jej wielkość nie powinna przekraczać kilkunastu procent, ponieważ zbyt duża mutacja może niszczyć znalezione rozwiązania.

- *crossoverProb* - prawdopodobieństwo krzyżowania. Przyjmuje wartości z przedziału $[0, 1]$. Zaleca się ustawialnie wartości z przedziału $[0.5, 0.9]$. Należy pamiętać o tym, że suma parametrów eliteProc i crossoverProb nie może być większa niż 1.
- mode - tryb w jakim ma działać algorytm. Przyjmuje wartości *regular*(w przypadku wyboru klasycznego modelu) lub *island*(w przypadku modelu wyspowego).
- *numberOfSeparateGenerations* - liczba naturalna określająca ilość iteracji jakie wykona algorytm pomiędzy rozdzieleniem populacji na mniejsze części, a ponownym jej scaleniem. Dla wyboru modelu klasycznego należy ustawić wartość parametru na 1.

Dodatkowo w pliku konfiguracyjnym określamy wektor popytu, podaży i macierz kosztów:

- *demand* - wektor popytu. Przyjmuje jako wartość listę elementów, które składają się z dwóch pól - i, które określa indeks wektora i val, które określa wartość wektora w miejscu i - $demand[i] = val$.
- *supply* - wektor podaży. Przyjmuje jako wartość listę elementów, o polach takich samych jak w przypadku wektora popytu. Pojedynczy element opisuje pole wektora $supply[i] = val$.
- $costMatrix$ - macierz kosztu, która może być wykorzystywana w funkcji celu. Przyjmuje jako wartość listę elementów, które składają się z trzech pól: s - określa indeks odpowiadający indeksowi wektora podaży, d - określa indeks odpowiadający indeksowi wektora popytu, oraz val - określa wartość macierzy w polu o podanych indeksach $costMatrix[d, s] = val$.

Przykładowy plik konfiguracyjny:

```
{
    "populationSize": 100,
    "eliteProc": 0.3,
    "mutationProb": 0.1,
    "mutationRate": 0.05,
    "crossoverProb": 0.2,
    "mode": "regular",
    "numberOfSeparateGenerations": 1,
    "demand": [
        {
            "val": 1.0,
            "i": 1
        },
        {
            "val": 10.0,
            "i": 2
        }
    ],
    "supply": [
        {
            "val": 1.0,
            "i": 1
        },
        {
            "val": 5.0,
            "i": 2
        },
        {
            "val": 5.0,
            "i": 3
        }
```

```
    ],
    "costMatrix": [
        {
            "val": 1.0,
            "s": 1,
            "d": 1
        },
        {
            "val": 2.0,
            "s": 2,
            "d": 1
        },
        {
            "val": 3.0,
            "s": 3,
            "d": 1
        },
        {
            "val": 10.0,
            "s": 1,
            "d": 2
        },
        {
            "val": 20.0,
            "s": 2,
            "d": 2
        },
        {
            "val": 30.0,
            "s": 3,
            "d": 2
        }
    ]
}
```

Wyniki eksperymentalne

4.1 Model klasyczny algorytmu genetycznego

4.2 Model wyspowy algorytmu genetycznego

4.3 Porównanie modelu klasycznego i wyspowego

Podsumowanie

W tym rozdziale znajdzie się podsumowanie pracy.

Bibliografia

[1] S. K. Jeff Bezanson, Alan Edelman, V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59:65–98, 2017.

[2] V. B. S. Jeff Bezanson, Stefan Karpinski, A. Edelman. Julia: A fast dynamic language for technical computing. *ArXiv:1209.5145*, September 2012.

[3] N. Saini. Review of selection methods in genetic algorithms. *International Journal of Engineering and Computer Science*, 6(12):22261–22263, Dec. 2017.

Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.