Carbon Footprint

Dokumentacja

Mateusz Biedak

Michał Furmanek

1. Wstęp

Celem projektu było stworzenie aplikacji umożliwiającej liczenie footprint'u dla dowolnego problemu, a w szczególności dla problemu **„Carbon Footprint"**.

Aplikacja ta powinna z wszystkich możliwych scenariuszy wydobycia węgla, wybrać ten, który zostawił po sobie najmniejszy ślad węglowy. W przypadku dużego drzewa operacji sięgającego bardzo daleko wstecz, dużo czasu zajęłoby liczenie wszystkich możliwości, dlatego należy użyć do tego algorytmu ewolucyjnego, który zwróci w krótkim czasie względnie dobre wyniki.

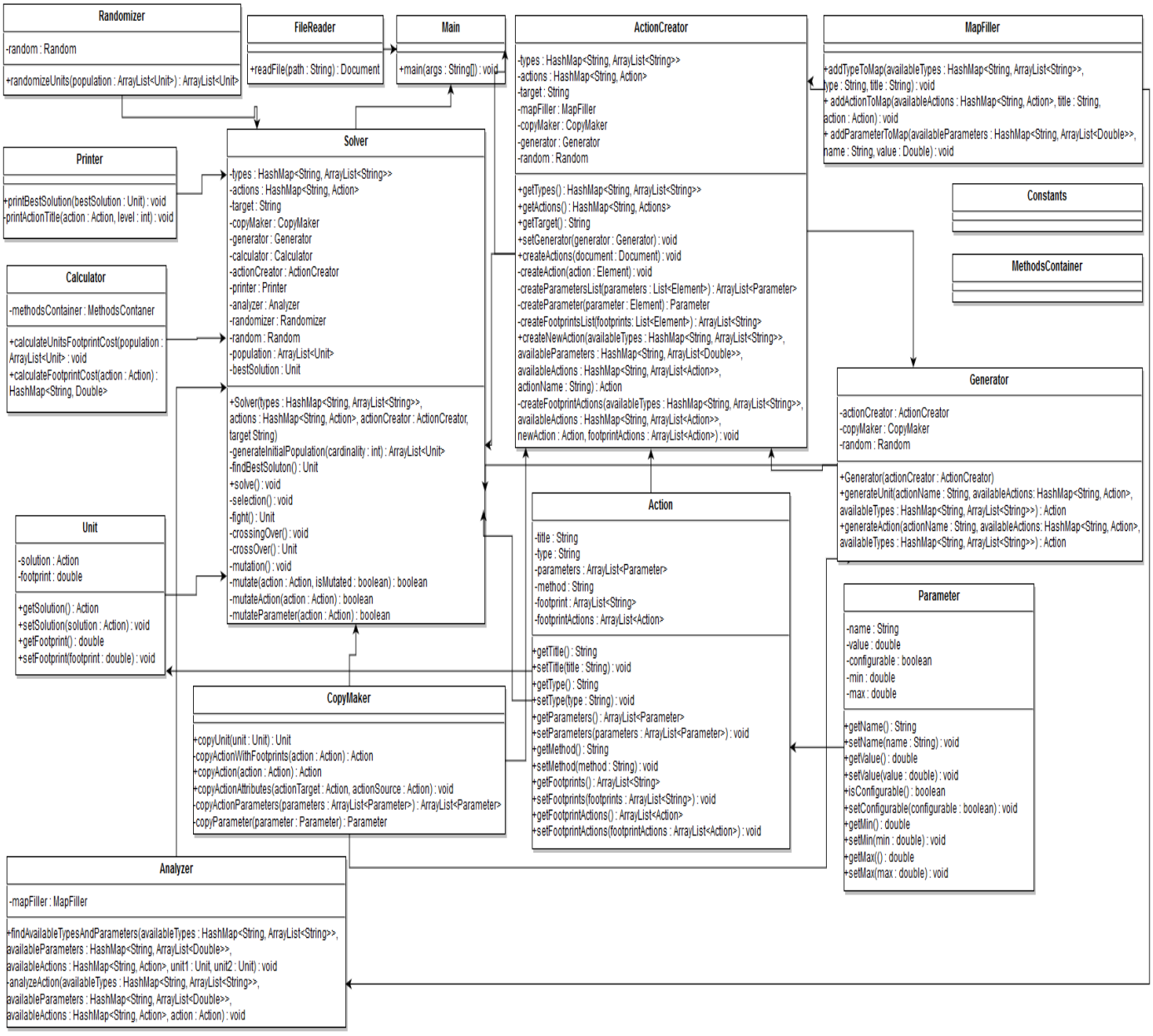
Aplikacja zatem składać się będzie z dwóch części. Części wczytującej plik xml, odpowiadający za dostarczenie danych o wszystkich możliwych akcjach do budowania drzewa akcji i potrafiącej przeliczać footprint dla tego drzewa, oraz części szukającej optymalnego rozwiązania przy pomocy algorytmu ewolucyjnego.

Obecne rozwiązanie pierwszej części programu jest wersją backendową, nie posiadającą GUI, zakładającą, że użytkownik zna się na programowaniu, ponieważ każda akcja zdefiniowana w pliku xml, posiada nazwę metody, która powinna być podczas przeliczania drzewa, dla tego węzła uruchomiona z poziomu jakiejś klasy – w tej wersji programu jest to wbudowana w projekcie klasa **MethodContainer**, a w przyszłych wersjach może bć to zrobione inaczej, o czym będzie w punkcie z propozycjami dalszego rozwoju.

Działanie programu w obecnym momencie wygląda tak, że na wejściu potrzebny jest nam plik xml z opisem wszystkich dostępnych w systemie akcji, ten kto podaje taki plik do odpowiedniego folderu, musi też w odpowiedniej klasie dopisać metody zdefinowane dla każdej akcji.

Wynik wyświetla się w konsoli w postaci drzewka akcji zrobionego tabulacjami, oraz wypisaniem wartości carbon footprint'u dla tego rozwiązania.

2. Diagram klas



Z racji słabej widoczności diagramu, załączony jest on do repozytorium i tam można go dokładniej obejrzeć.

Main – klasa uruchamiająca cały program.

FileReader – klasa wczytująca plik XML do dokumentu, który używany jest podczas tworzenia wszystkich obiektów opisanych w pliku.

ActionCreator – klasa tworząca wszystkie obiekty opisane w pliku XML.

Solver – klasa posiadająca algorytm ewolucyjny, uruchamia go na danych uprzednio stworzonych przez klasę **ActionCreator.**

CopyMaker – klasa tworząca kopie podanych obiektów, ten mechanizm jest potrzebny ponieważ chcemy uniknąć czytania z pliku za każdym razem gdy tworzymy osobnika, dlatego wszystkie akcje wczytane są do systemu jako wzrocowe, a następnie przy tworzeniu populacji - aby ich nie naruszyć podczas zmieniania parametrów – kopiowane, bo jeśli przypiszemy wiele zmiennych do tej samej referencji to zmiana jednej zmiennej spowodowałaby zmianę wzorca i innych osobników.

Action - klasa reprezentująca akcję opisaną w pliku XML.

Parameter – klasa reprezentująca parametr akcji opisany w pliku XML.

Unit – klasa reprezentująca osobnika populacji przechowującego swoje rozwiązanie oraz jego wartość.

MethodsContainer – klasa przetrzymująca metody, których nazwy zostały podane w pliku XMl dla każdego węzła, służą one do obliczania wartości footprint'u dla każdego węzła.

Generator – klasa generująca początkową populację.

MapFiller – klasa wypełniająca mapy parametrów, akcji oraz typów akcji, które mogą w danym momencie być użyte do tworzenia rozwiązania albo populacji.

Analyzer – klasa sprawdzająca jakie parametry oraz akcje są możliwe w danym momencie do użycia aby klasa **MapFiller** mogła je dodać do mapy.

Calculator – klasa zliczająca footprint dla całego drzewa.

Randomizer – klasa wybierająca dwa osobniki do wykonania danej operacji, czyli na przykład krzyżowania.

Printer – klasa wypisująca wyniki na konsolę.

Constants – klasa ze stałymi programu.

3. Wejście programu

```
<CarbonFootprint>
    <Actions>
        <Action>
            <Title>(Title)</Title>
            <Type>(Name of some type)</Type>
            <Parameters>
                <Parameter>
                    <Name>(Name)</Name>
                    <Value>(Value)</Value>
                    <Configurable>(True or False)</Configurable>
                    <Min>(Value)</Min>
                    <Max>(Value)</Max>
                </Parameter>
                ...
            </Parameters>
            <Method>(Method name)</Method>
            <Footprints>
                <Footprint>(Type of action)</Footprint>
                ...
            </Footprints>
        </Action>
        ...
    </Actions>
    <Target>(Title of action that is result of eariler actions)</Target>
</CarbonFootprint>
```

Opis znaczników przykładowego xml'a:

- **Actions** – ten znacznik odpowiada liście akcji możliwych do zaistnienia w problemie, jako akcję definiujemy jeden punkt w historii footprint'u taki jak naprzykład stworzenie fabryki, albo stworzenie ciężarówki.

- **Action** – ten znacznik to konkretna akcja, jaka będzie w programie reprezentowana obiektowo, posiadać ona może wiele atrybutów, niektóre są wymagane, niektóre opcjonalne, wszystko to będzie opisane w kolejnych znacznikach.

- **Title** – to pierwszy atrybut akcji, w to pole należy po prostu wpisać nazwę operacji czyli właśnie „Stworzenie ciężarówki” bądź inną, po której rozpoznawana będzie ta akcja w programie – jest to obowiązkowe pole, bo każda akcja ma jakąś nazwę.

- **Type** – jest to atrybut mówiący jakiego typu jest dana akcja – czyli na przykład czy jest to transport, czy zatrudnienie jakiegoś pracownika, czy coś innego, ale jest ważne by dany typ akcji zawsze nazywać w ten sam sposób, ponieważ akcje tego samego typu będą mogły być przez algorytm ewolucyjny podmieniane po to, by sprawdzić, która akcja danego typu będzie najbardziej optymalna dla footprintu i zostawi najmniejszy ślad.

- **Parameters** – lista parametrów, które mogą dotyczyć danej akcji, czyli na przykład przy tworzeniu fabryki można by chcieć uzależnić koszt footprintu od ilości hangarów, które będą w niej zbudowane czy czegoś innego, ten znacznik jest również opcjonalny, ponieważ dana akcja wcale nie musi mieć parametrów, może po prostu mieć jakiś stały koszt.

- **Parameter** – jest to konkretny jeden parametr, który ma też kilka atrybutów opisanych niżej.

- **Name** – to jest po prostu nazwa danego atrybutu, czyli „Ilość hangarów” albo „Pole powierzchni” i jest to atrybut wymagany.

- **Value** – to domyślna wartość jakiegoś parametru czyli na przykład 2 hangary i teraz w zależności od tego czy dany parametr może być zmienny i czy chcemy dać algorytmowi ewolucyjnemu możliwość sterowania tym parametrem w taki sposób aby sprawdzić jaka ilość hangarów będzie najbardziej optymalna, trzeba dodać jeszcze kolejne 3 atrybuty opisane niżej.

- **Configurable** – to atrybut, który po prostu mówi programowi czy cały parametr może być konfigurowalny, ma dwie wartości – True lub False, jest to atrybut opcjonalny, jesli go nie podamy, to nie możemy też jednak podać kolejnych dwóch.

- **Min** – to wartość minimalna jaką dany parametr może przyjąć.

- **Max** – to wartość maksymalna jaką dany parametr może przyjąć.

- **Method** – to nazwa funkcji jakiej należy użyć z poziomu języka Java, aby obliczyć wzór dla danej akcji, czyli jej wartość footprintu. Teoretycznie powina być ona napisana przez kogoś, kto pisze też tego xml'a, ale w wersji 1.0 naszego programu, po prostu my napiszemy wszystkie funkcje, a gdy ktoś kiedyś w wersji 2.0 zrobi odpowiedni interfejs użytkownika, to będzie można też tam pisać te wzory.

- **Footprints** – definiuje on jakie typy akcji składają się na daną akcję, jest on opcjonalny ponieważ możemy na pewnym poziomie footprintu skończyć szukanie śladu i nie chcemy już dalej definiować operacji składających się na daną akcję.

- **Footprint** – to już konkretny typ akcji zdefiniowany na liście akcji składających się na daną akcję. Ten znacznik jest nieobowiązkowy tak samo jak Footprints, ale jeśli istnieje Footprints, to co najmniej jeden Footprint musi też zaistnieć, bo po co inaczej definiować listę, która byłaby pusta.

- **Target** – to nasza akcja „Cel", czyli ta, której cały footprint chcemy policzyć, w szczególnym przypadku, jest to cel tego projektu, czyli policzenie Carbon Footprintu, dlatego najczęściej tą akcją będzie po prostu wydobycie węgla, czy dowiezienie węgla w konkretne miejsce. W tym znaczniku podajemy po prostu nazwę jednej z akcji zdefiniowanych w pliku XML. Znacznik ten jest obowiązkowy bo od czegoś trzeba to poszukiwanie zacząć.

Każda akcja powina mieć:

- tytuł .
- typ akcji (jakiego typu to jest akcja).

- wzór według, którego jest przeliczany footprint dla tej akcji.

Każda akcja może mieć:

- listę typów akcji składających się na daną akcję (znacznik Footprints) – czyli tak jak opisane wyżej, jeśli na tym poziomie jeszcze szukanie footprintu się nie kończy.

- listę parametrów jeśli dana akcja jakieś posiada (każdy parametr powinien posiadać nazwę i domyślną wartość, a jeśli jest dodatkowo konfigurowalny, to napisać, że jest konfigurowalny i podać minimalną granicę wartości i maksymalną granicę wartości).

4. Algorytm ewolucyjny

W problemie footprint'u rozwiązanie jest w postaci drzewa, w którym każdy węzeł to jedna akcja, która zaistniała aby kolejne mogły zaistnieć. Drzewo to rozpina się od korzenia – czyli akcji, dla której footprint jest liczony, przez kolejne węzły – czyli akcje się na daną akcję składające, aż do liści, gdzie ustanawiamy granicę liczenia footprint'u i zakładamy, że dane na dane akcje, żadne inne już się nie składały.

W algorytmie ewolucyjnym potrzebujemy populacji rozwiązań, więc na początku generujemy wstępną populację, a następnie przez pewną ilość iteracji będziemy poddawać ją trzem operacjom.

Są to:

- selekcja

- krzyżowanie

- mutacja

Dodatkowo po każdej iteracji szukamy najlepszego rozwiązania i sprawdzamy, czy jest ono lepsze od dotychczas najlepszego znalezionego, ponieważ element poszukiwawczy najlepszego rozwiązania jest elementem losowym, to nie zawsze nasza populacja będzie zmieniała się na lepsze, więc nie warto brać najlepszego rozwiązania z ostatecznej populacji.

Selekcja została tutaj zaimplementowana w taki sposób, że tworzymy nową populację – dwa razy mniejszą – wybierając losowo 2 osobniki, sprawdzając który jest lepszy, wybierając go do nowej populacji, a oba z nich zwracając spowrotem do puli. W ten sposób może zajść sytuacja, że do nowej populacji trafią same dobre osobniki, ale też mogą się powtarzać, co sprawi, że ten sam osobnik, może w przyszłych krokach zmutować się na kilka sposobów.

Krzyżowanie działa tutaj w ten sposób, że tworzymy nową populację tym razem spowrotem rozmiarów początkowej populacji, tak aby wraz z kolejnymi iteracjami rozmiar populacji nam nie malał. A tworzymy ją wybierając znów losowo 2 osobniki, rozkładając ich drzewa na części i składając w losowy sposób ich węzły tak, żeby powstało nowe drzewo –

w szczególnym przypadku może zbudować się jeden z rodziców nowego osobnika – co symulowałoby szansę na brak krzyżowania.

Mutacja była najcieższą operacją do zaimplementowania, ponieważ mutacja może w tym przypadku występować w postaci podmienienia węzła danego typu na inny węzeł danego typu, albo zmiany wartości parametru dla jakiegoś węzła. Dla każdego z osobników przechodzimy po każdym z jego węzłów dopóki nie zaistnieje jakakolwiek mutacja, czyli dla każdego węzła losujemy czy ma zaistnieć mutacja – jeśli tak to losujemy, czy będzie to mutacja węzła czy jego parametru, oczywiście mutacja ta może się nie powieść, bo dany węzeł może nie mieć alternatyw, albo parametr może nie być konfigurowalny – w tych przypadkach zwracana jest flaga, że mutacja nie zastniała i w następnej kolejności próbujemy mutować potomków danego węzła.

Na samym końcu algorytmu wypisujemy wynik najlepszego znalezionego rozwiązania na konslę.

5. Eksperymenty

Eksperymenty prowadzone były na poniższym pliku XML:

```
<?xml version="1.0"?>
<CarbonFootprint>
    <Actions>

        <Action>
            <Title>Carbon delivery</Title>
            <Type>Finish</Type>

            <Parameters>
                <Parameter>
                    <Name>Distance</Name>
                    <Value>100</Value>
                    <Configurable>True</Configurable>
                    <Min>20</Min>
                    <Max>1000</Max>
                </Parameter>
                <Parameter>
                    <Name>Cost</Name>
                    <Value>50</Value>
                    <Configurable>True</Configurable>
                    <Min>10</Min>
                    <Max>100</Max>
                </Parameter>
            </Parameters>

            <Method>calculateDeliveryCost</Method>

            <Footprints>
                <Footprint>Vehicle</Footprint>
            </Footprints>
        </Action>

        <Action>
            <Title>Truck creation</Title>
            <Type>Vehicle</Type>
            <Method>getTruckCreationCost</Method>
        </Action>

        <Action>
            <Title>Boat creation</Title>
            <Type>Vehicle</Type>
            <Method>getBoatCreationCost</Method>
        </Action>

    </Actions>

    <Target>Carbon delivery</Target>
</CarbonFootprint>
```

Metody liczące footprintdla każdego z węzłów były zaimplementowane tak:

```
public HashMap<String, Double> calculateDeliveryCost(HashMap<String, Double> ins, ArrayList<Parameter> parameters) {
    HashMap<String, Double> result = new HashMap<String, Double>();
    double cost = ins.get("TruckCreationCost");
    cost += parameters.get(0).getValue();
    cost += parameters.get(1).getValue();
    result.put("DeliveryCost", cost);
    return result;
}

public HashMap<String, Double> getTruckCreationCost(HashMap<String, Double> ins, ArrayList<Parameter> parameters) {
    HashMap<String, Double> result = new HashMap<String, Double>();
    result.put("TruckCreationCost", 4000.0);
    return result;
}

public HashMap<String, Double> getBoatCreationCost(HashMap<String, Double> ins, ArrayList<Parameter> parameters) {
    HashMap<String, Double> result = new HashMap<String, Double>();
    result.put("TruckCreationCost", 10000.0);
    return result;
}
```

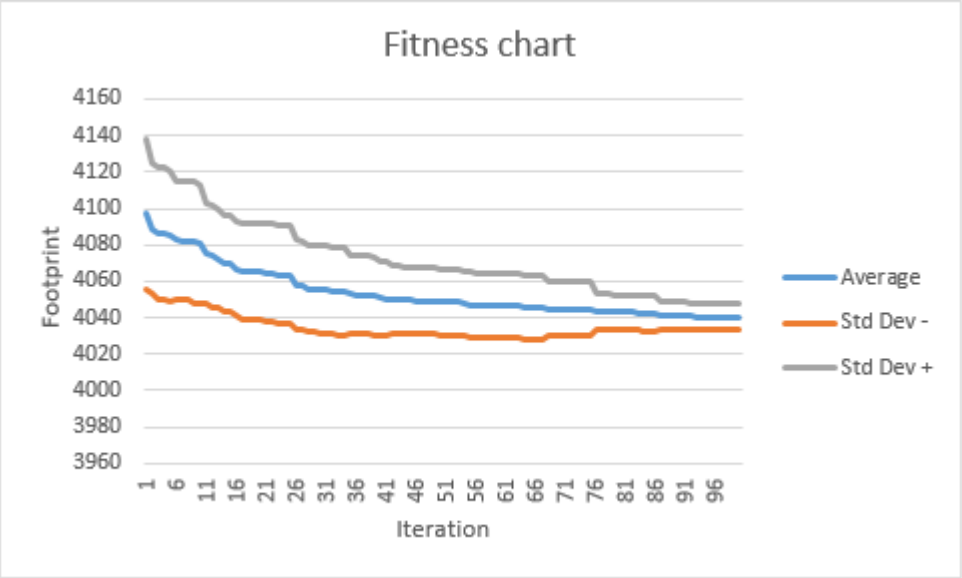
Na podstawie tych danych widać, że minimalna wartość jaką da się policzyć to 4030.0 a maksymalna to 11100.0.

Parametry algorytmu były ustawione w ten sposób:
- ilość iteracji = 100
- rozmiar populacji = 50
- szansa na mutację = 10%

Przykładowy wynik programu:

```
Best footprint is: 4031.5169599369
And best combination looks like:
Carbon delivery (Distance: 21.478691224562724, Cost: 10.038268712337231)
    Truck creation
```

Po 30-krotnym uruchomieniu programu i zgromadzonych danych, obliczyliśmy odchylenie standardowe, wyniki prezentują się następująco:



6. Możliwe usprawnienia oraz dalszy rozwój

Usprawnienia:

- kod jest dość dobrze podzielony na metody i klasy, ale być może ktoś wymyśliłby coś jeszcze lepszego – więc być może refactoring kodu

- dodanie obsługi błędów, ponieważ w tym momencie wszystko było zrobione tak aby projekt działał i liczył, a obsługa błędów tam gdzie są możliwe – czyli na przykład przy wczytywaniu xml'a albo odwoływaniu się do metod mechanizmem refleksji – została na poziomie e.printStackTrace()

- dodanie walidacji xml'a zgodnie z zasadami wypisanymi w punkcie **„Wejście programu"**, może być też zrobione poprzez wprowadzenie GUI wymienionego w podpunkcie **„Dalszy rozwój"**

- wprowadzenie propertiesów zamiast niektórych stałych, które znajdują się w klasie **Constants**, ponieważ narazie niektóre parametry zostały ustawione jako stałe – i tam są zmieniane, a są nimi :

  - CARDINALITY – czyli rozmiar populacji
  - NUMBER_OF_ITERATIONS – ilość iteracji algorytmu ewolucyjnego
  - MUTATION_CHANCE – szansa na zmutowanie węzła

- możliwe zmienianie wymienionych wartości poprzez GUI, zamiast propertiesów, chyba że być może jakieś podejście hybrydowe

Dalszy rozwój:
- stworzenie GUI do programu, ponieważ narazie jest tak zrobione, że trzeba ręcznie tworzyć xml'e i wstawiać w odpowiednie miejsce w projekcie i dopisywać do klasy **MethodsContainer** odpowiednie metody – ktoś nie będący informatykiem może takich rzeczy nie umieć

- dodanie walidacji do tego GUI gdy tworzone będą kolene węzły, tak aby sprawdzić czy drzewo może być poprawnie stworzone, albo czy zasady opisane wyżej są spełnione

- być może zamienić podejście z klasą MethodsContainer – które jest dość sztywne bo trzeba dodawać metody konkretnie do tej klasy – na jakieś inne, gdize użytkownik może podać już swoją klasę z zaimplementowanymi metodami, albo podać swój jar, albo metody te mogą być dopisywane z poziomu stworzonego GUI do odpowiedniej klasy

- dodać eksportowanie wyników do jakiegoś pliku tekstowego bądź wygenerować jakiś plik wizualny z obrazkiem całego drzewa wynikowego, ponieważ narazie wszystko jest wyświetlane w konsoli