POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Systemów Inteligentnych, Algorytmiki i Matematyki

Inteligentne Systemy Przetwarzania Danych

Bartosz Żuk

Nr albumu s18174

Ewolucyjna naprawa programów współbieżnych na przykładzie ARJi i algorytmu Petersona

Praca inżynierska
Napisana pod kierunkiem
Dr. Michała Knapika

Warszawa, luty, 2021

Streszczenie

Celem pracy jest zbadanie działania ARJi, narzędzia do ewolucyjnej naprawy programów w Javie, na przykładach uszkodzonych programów współbieżnych. W ramach badania podjęto próbę naprawy uszkodzonej wersji algorytmu Petersona, klasycznego rozwiązania problemu wzajemnego wykluczenia dla dwóch procesów. Przeprowadzono również próby naprawy uszkodzonej wersji tego samego algorytmu zaimplementowanej w sposób sekwencyjny, gwarantujący kontrolę nad wykonywanym przeplotem. Otrzymane wyniki poddano ocenie w celu określenia ich poprawności. Wybrane wygenerowane programy zostały zaprezentowane i omówione. Praca opisuje również dwa błędy w kodzie narzędzia ARJA, znalezione w trakcie przeprowadzania badań.

Słowa kluczowe

Automatyczna naprawa oprogramowania, Ewolucyjna naprawa oprogramowania, ARJA, Algorytmy genetyczne, Programowanie ewolucyjne, Programy współbieżne, Algorytm Petersona

Spis treści

Rozdział 1

Opis Pracy

Błędy są nieodłącznym produktem ubocznym tworzenia oprogramowania, zaś ich skutki mogą się wahać od nieznaczących do katastrofalnych. Przykładem poważnych konsekwencji niewielkiej usterki w oprogramowaniu jest nieudany start rakiety Ariane 5. W wyniku błędu, polegającego na zapisie 64-bitowej liczby do 16-bitowej zmiennej [17], doszło do eksplozji rakiety. Koszt tego błędu szacuje się na 8 milionów dolarów. Przykład ten nie jest odosobnionym przypadkiem.

Historie takie jak ta pozwalają na zobrazowanie, jak istotnym problemem mogą być błędy oprogramowania oraz jak kluczowym wyzwaniem jest ich efektywna identyfikacja i naprawa. Jedną z prób rozwiązania problemu jest próba automatyzacji tego procesu. Środowisko naukowe oraz podmioty komercyjne niestrudzenie pracują nad udoskonaleniem istniejących narzędzi, licząc że w niedalekiej przyszłości będą one w stanie zastąpić programistów w żmudnej pracy szukania i naprawy błędów.

Szczególnym rodzajem usterek, na których skupia się ta praca, są błędy wynikające ze współbieżności. Współbieżność jest kluczowym mechanizmem dla współczesnych systemów informatycznych, jednak błędy wynikające ze współdziałania wielu procesów lub wątków są niezwykle trudne do wykrycia i usunięcia.

W niniejszej pracy przyjrzano się jak ARJA [35], jedno z wiodących narzędzi do automatycznej naprawy, radzi sobie z uszkodzoną implementacją algorytmu Petersona - klasycznego rozwiązania problemu wzajemnego wykluczenia.

1.1 Cel pracy

Celem pracy jest przeprowadzenie badania polegającego na automatycznej naprawie wadliwej implementacji algorytmu Petersona przy pomocy ARJi. ARJA reprezentuje podejście "ewolucyjne", w którym naprawa oprogramowania postrzegana jest jako **zadanie optymalizacji**, zaś dla jego rozwiązania stosowane są **algorytmy ewolucyjne**. Dla porównania przeprowadzono analogiczne badanie wersji tego samego algorytmu, która jest efektywnie sekwencyjna, tzn. przeplot wykonywany przez algorytm może być zadany na wejściu.

Badanie ma na celu ocenę, czy niedeterministyczna natura błędów w programach współbieżnych uniemożliwia ich naprawę. Wygenerowane rozwiązania (*łatki* naprawiające program) zostają poddane omówieniu i ocenie. Ponadto zobrazowano wyzwania i trudności towarzyszące ewolucyjnej naprawie oprogramowania w kontekście programów współbieżnych.

1.2 Plan pracy

Praca składa się z **5 rozdziałów**.

1. Rozdział pierwszy przedstawia tematykę pracy oraz formułuje cel badania.

2. Rozdział drugi przedstawia algorytmy genetyczne i definiuje powiązane z nimi pojęcia wykorzystywane w dalszej części pracy. Przedstawiono również krótki zarys historii algorytmów ewolucyjnych i ich przykładowe zastosowania.

3. Rozdział trzeci poświęcony jest ewolucyjnej naprawie programów, a w szczegolności dokładnym opisie działania ARJi. Przedstawiono pseudokod wykorzystywanego w ARJi algorytmu genetycznego, omówiono zastosowane mechanizmy optymalizujące proces naprawy oraz pokazano w jaki sposób uruchomić narzędzie.

4. Rozdział czwarty prezentuje wyniki prób napraw programu współbieżnego implementującego algorytm Petersona. Początek rozdziału poświęcony jest krótkiemu opisowi współbieżności oraz problemów z nią związanych, następnie omówiono algorytm Petersona i zaprezentowano jego uszkodzoną wersję wraz ze wskazaniem błędnego przeplotu naruszającego zasadę wzajemnego wykluczenia. Dalsza część rozdziału skupia się na opisie przebiegu badania i analizie jego wyników.

5. Ostatni rozdział to konkluzje. Omówiono w nim napotkane trudności - w tym dwa błędy jakie zdołano znaleźć w kodzie narzędzia ARJA. Ponadto przedstawiono wnioski i obserwacje oraz zaproponowano dalsze kierunki badań związane z ewolucyjną naprawą programów współbieżnych.

Rozdział 2

Algorytmy genetyczne

W niniejszym rozdziale zaprezentowano podstawowe pojęcia dotyczące algorytmów ewolucyjnych. Przedstawiono również krótki przegląd literatury dotyczącej zastosowań algorytmów genetycznych w inżynierii oprogramowania.

2.1 Zadanie optymalizacji

Algorytmy ewolucyjne stosowane są do rozwiązywania zadań optymalizacyjnych. Celem zadania optymalizującego jest znalezienie najlepszego rozwiązania, ze zbioru wszystkich rozwiązań alternatywnych, nazywanego dalej **przestrzenią rozwiązań**. Do oceny jakości wybranego rozwiązania używa się funkcji celu. W rezultacie każde zadanie optymalizacyjne sprowadza się do znalezienia rozwiązania dla którego funkcja celu będzie osiągała swoje minimum (alternatywnie maksimum) globalne. Znane są grupy funkcji dla których istnieją metody analityczne na wyznaczanie ich ekstremów, jednak dla większości bardziej złożonych problemów używane są **metody heurystyczne**. Heurystyka w kontekście zadań optymalizacyjnych polega na przyjęciu pewnej strategii przeszukiwania przestrzeni rozwiązań. Strategia ta powinna umożliwiać, w akceptowalnym czasie, znalezienie rozwiązania zadowalająco zbliżonego do rozwiązania optymalnego. Często można spotkać się z heurystykami, które operują w sposób niedeterministyczny, tzn. zawierają pewien element losowości, który sprawia że za każdym jej użyciem możemy otrzymać różniące się rezultaty.

2.2 Algorytmy ewolucyjne - zarys historii i zastosowań

Algorytmy ewolucyjne są niedeterministyczną metodą heurystyczną. Powstały wskutek inspiracji mechanizmami ewolucji biologicznej, jak zaproponowany przez Darwina dobór naturalny oraz procesami znanymi z genetyki, jak mutacja i krzyżowanie organizmów. Algorytmy ewolucyjne są grupą wielu różnych typów algorytmów. Najczęściej można się spotkać ze **strategiami ewolucyjnymi**, **programowaniem genetycznym** i omawianymi w tej pracy **algorytmami genetycznymi**. Łączy je podejście do rozwiązywania zadanego problemu polegające na symulowaniu wybranych elementów procesu ewolucji na kolejnych zbiorach rozwiązań, nazywanych populacjami, w celu ich optymalizacji.

2.2.1 Zarys historii

Różne podejścia do algorytmów ewolucyjnych rozwijane były równolegle. Za twórcę algorytmów genetycznych uznawany jest John Henry Holland, który w [10] przedstawił model ewolucji populacji osobników, opisanych za pomocą binarnego kodu genetycznego. Na podstawie kodu danego osobnika przypisana zostaje mu wartość liczbowa wyrażająca jakość reprezentowanego przez niego rozwiązania. Wartość ta uzależnia prawdopodobieństwo z jakim dany osobnik będzie wybierany do następnej populacji. Oprócz tego Holland wprowadził mechanizm krzyżowania osobników oraz ich mutacji.

Ingo Rechenberg i Hans-Paul Schwefel [27, 25], opracowali metody zwane strategiami ewolucyjnymi. Różnią się one od algorytmów genetycznych zastosowaniem wyłącznie mutacji i selekcji jako technik przeszukiwania przestrzeni rozwiązań.

Zarówno algorytmy genetyczne, jak i strategie ewolucyjne były metodami rozwijanym niezależnie od lat 60. Z nadejściem lat 80 algorytmy ewolucyjne zaczęły zyskiwać na popularności, nastąpiła wymiana idei pomiędzy dotychczas odseparowanymi nurtami badań, co sprawiło że różnice pomiędzy poszczególnymi typami algorytmów ewolucyjnych zaczęły się zacierać. Proces ten nabrał intensywności wraz z nastaniem lat 90. Algorytmy genetyczne zapożyczyły mechanizm selekcji znany z strategii ewolucyjnych, z kolei strategie ewolucyjne zaczęły korzystać z krzyżowania [2, s. 19].

W 1990 roku John Koza [13] zaproponował ostatni z wymienionych wyżej typ algorytmów ewolucyjnych - programowanie genetyczne. Idea programowania genetycznego opiera się na sterowaniu procesem ewolucji programów komputerowych, które mają rozwiązać dane zadanie. Tym samym programowanie genetyczne zamienia przestrzeń rozwiązań konkretnego problemu na przestrzeń programów rozwiązujących ten problem. Jednym z najistotniejszych wkładów programowania genetycznego w dziedzinę metod ewolucyjnych było reprezentowanie osobników za pomocą struktur drzewiastych.

2.2.2 Zarys zastosowań (poza inżynierią oprogramowania)

Jak wspomniano, algorytmy ewolucyjne stosowane są do zadań optymalizacyjnych. W rzeczywistości oznacza to zazwyczaj rozwiązywanie zadań związanych z planowaniem lub harmonogramowaniem złożonych, często dynamicznych procesów, na które wpływa duża liczba zmiennych i dla których nie opracowano bardziej formalnych metod znajdywania satysfakcjonujących rozwiązań. Arabas [2] przytacza kilka przykładów wykorzystania technik ewolucyjnych w szeroko pojętym przemyśle, m.in. [2, s. 20]:

- projektowanie kształtu komory silnika odrzutowego;
- optymalizacje rozłożenia elementów na płytce krzemowej, w mikroelektronice;
- harmonogramowanie kolejności dostarczania przesyłek przez firmy kurierskie.

Oprócz przykładów przytoczonych przez Arabasa, algorytmy ewolucyjne stosowane są również w szeroko pojętym uczeniu maszynowym. Przykładem jest ich zastosowanie przy programie AlphaStar [28], sztucznej inteligencji opartej o sieci neuronowe, która zdołała pokonać profesjonalnego gracza w popularnej grze strategicznej czasu rzeczywistego Starcraft II. Algorytmy ewolucyjne zostały tam użyte do optymalizacji struktury sieci neuronowych [3].

Powyższe przykłady pokazują, że dzięki swej uniwersalności i efektywności algorytmy ewolucyjne znajdują zastosowanie w szerokiej gamie różnych dziedzin problemowych.

2.3 Podstawowe pojęcia

W tej sekcji, bazując na definicjach z [2], opisano podstawowe pojęcia dotyczące algorytmów genetycznych i przedstawiono ich ogólną postać. Na końcu omówiono optymalizacje wielokryterialna oraz wprowadzono twierdzenie o schematach, będące próbą teoretycznego uzasadnienia zbieżności algorytmów genetycznych.

2.3.1 Populacja, osobnik, chromosom

Najogólniej ujmując, algorytmy ewolucyjne polegają na iteracyjnym tworzeniu kolejnych populacji czyli grupy osobników.

Genotyp i fenotyp Każdy osobnik reprezentuje rozwiązanie danego zadania i posiada swój genotyp i fenotyp. Są to terminy zaczerpnięte z genetyki. Pierwszy termin oznacza sposób w który osobnik, reprezentujący wybrane rozwiązanie problemu, będzie kodowany. Kodowanie to powinno umożliwiać odtworzenie fenotypu osobnika czyli zbioru jego cech. Przyjmijmy za przykład trywialne zadanie optymalizacyjne, polegające na znalezieniu prostokąta o największym możliwym polu, przy ustalonym obwodzie. Dla tak sformułowanego zadania można w prosty sposób zakodować rozwiązania. Genotypem danego osobnika może być wektor dwóch liczb odpowiadających długości boków prostokąta, zaś fenotypem interesujące nas cechy prostokąta, które możemy wyliczyć na podstawie kodowania, czyli jego pole i obwód.

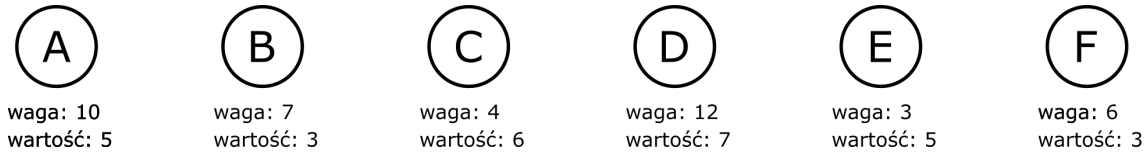
Chromosomy W każdym genotypie zawiera się przynajmniej jeden chromosom. Chromosom jest kolejnym terminem zaczerpniętym z genetyki. W kontekście algorytmów ewolucyjnych jest to fragment kodowania, wydzielony ze względu na rodzaj przechowywanej informacji. Każdy genotyp musi posiadać chromosomy kodujące fenotyp osobnika. Oprócz tego zdarza się, że konkretna implementacja algorytmu ewolucyjnego wymaga zapamiętywania innych informacji o osobniku przy wykorzystaniu dodatkowych chromosomów.

Kodowanie osobników Spójrzmy teraz na dwa popularne sposoby kodowania osobników. Pierwszym sposobem jest kodowanie genotypu za pomocą wektora. Kodowanie to zostało wykorzystane w przykładzie z prostokątami. Szczególnie popularną wersją kodowania wektorowego jest **kodowanie binarne**. Polega ono na użyciu wektora zer i jedynek. Klasycznym wykorzystaniem kodowania binarnego jest reprezentacja rozwiązań problemu plecakowego. **Problem plecakowy** (ang. *knapsack problem*) [5, s. 431], jest zadaniem optymalizacyjnym, które można sformułować następująco. Istnieje skończony zbiór przedmiotów, z których każdy ma swoją wagę i wartość. Celem zadania jest zapakowanie plecaka w taki sposób, by nie przekroczyć jego limitu dopuszczalnej wagi i jednocześnie zmaksymalizować wartość zabranych przedmiotów. **Decyzyjna wersja problemu plecakowego** sprowadza się do odpowiedzi na pytanie, czy możliwe jest osiągniecie danej wartości przedmiotów w plecaku przy zachowaniu określonego limitu wagi. Jest to jeden z 21 problemów NP-zupełnych wymienionych na liście Karpa [12]. Nie jest znany algorytm, który byłby w stanie znaleźć optymalne rozwiązanie dla tego problemu w czasie wielomianowym, stąd też konieczne jest użycie metod heurystycznych do szybkiego znajdywania rozwiązań o zadowalającej jakości.

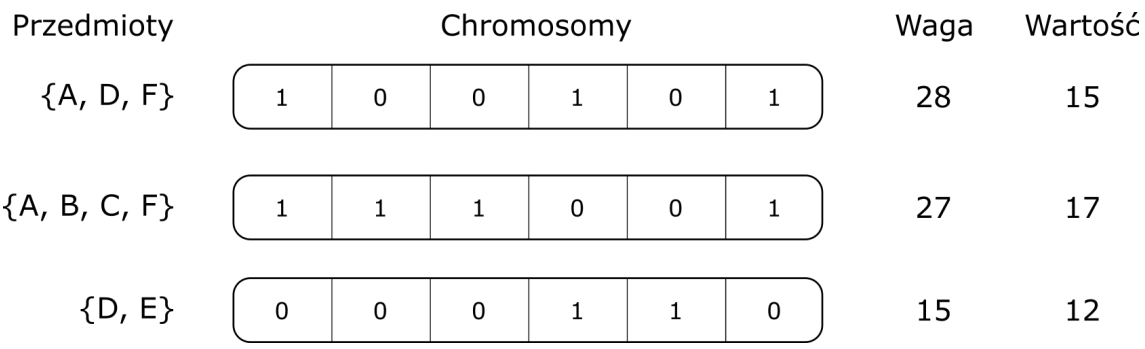
Rozwiązaniem problemu plecakowego jest podzbiór przedmiotów, które zostały zapakowane. Kodowanie rozwiązań wykonywane jest przy pomocy wektora charakterystycznego pozbiorów. Niech $X = \{x_0, \ldots, x_n\}$ będzie zbiorem wszystkich przedmiotów. Każdy podzbiór $Y \subseteq X$ można opisać za pomocą wektora charakterystycznego $(b_0, \ldots, b_n)$ gdzie:

$$b_i = \begin{cases} 1 & \text{jeśli } x_i \in Y \\ 0 & \text{jeśli } x_i \notin Y \end{cases}, \qquad dla \;\; i = 0, \ldots, n \tag{2.1}$$

W ten sposób każdy osobnik będzie posiadał genotyp, składający się z jednego binarnie zakodowanego chromosomu, natomiast fenotypem są łatwo obliczalne z genotypu sumaryczna waga i wartość plecaka.



(a)

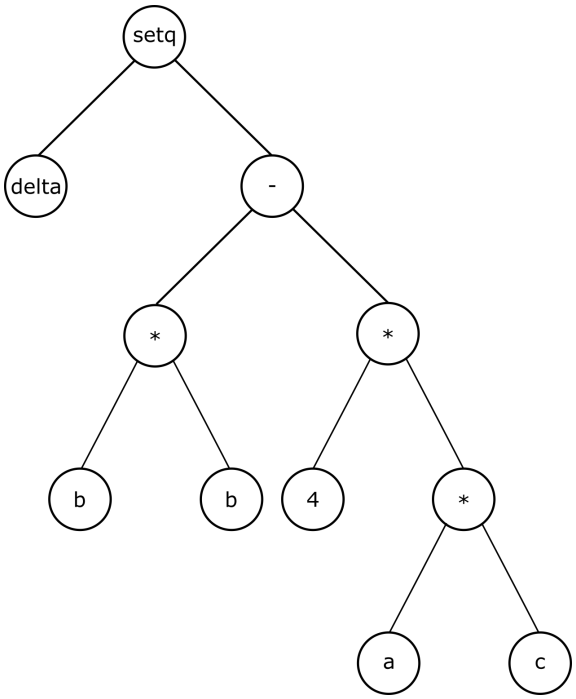


(b)

Rysunek 2.1: Ilustracja kodowania rozwiązań dla problemu plecakowego: (a) przykładowy zbiór przedmiotów wraz z wagami i wartością, (b) chromosomy binarne kodujące trzy wybrane rozwiązań wraz z obliczoną sumaryczną wagą i wartością

Kolejnym sposobem reprezentowania osobników używanym w algorytmach ewolucyjnych jest użycie **struktur drzewiastych**. Jak wspomniano na początku rozdziału, reprezentacja drzewiasta została spopularyzowana dzięki programowaniu genetycznemu. W programowaniu genetycznym ewoluujemy populacje programów komputerowych. Każdemu programowi przypisujemy genotyp w postaci drzewa AST (ang. *abstract syntax tree*), nazywanym też drzewem składniowym. Reprezentuje ono składniową strukturę programu. Każdy węzeł drzewa AST zawiera pewne wyrażenie, zaś dzieci tego węzła są znaczącymi

składowymi tego wyrażenia [30]. Dzięki strukturze drzewa AST możemy zakodować dowolny program. W szczególności, łatwość manipulacji drzewiastymi reprezentacjami programów napisanych w języku LISP [13, 20] uczyniły go częstym celem badań w dziedzinie programowania genetycznego. Poniższy przykład ilustruje drzewo AST prostego programu napisanego w języku LISP obliczającego wyróżnik równania kwadratowego.



Rysunek 2.2: Drzewo AST wyrażenia $(seqt\ (delta\ -(*(b\ b)\ *(4\ *(a\ c)))))$ [2, s. 97]

2.3.2 Algorytm genetyczny

Algorytmy genetyczne są jednym z typów algorytmów ewolucyjnych. Podobnie jak reszta algorytmów ewolucyjnych polegają na iteracyjnym generowaniu populacji coraz lepiej przystosowanych osobników reprezentujących alternatywne rozwiązania danego problemu. Wyróżniają się od innych algorytmów ewolucyjnych tym, że oprócz mutacji używają również mechanizm krzyżowania. Poniżej przedstawiony został algorytm genetyczny w ogólnym zarysie, by w dalszej części wprowadzić uszczegółowienie definicji i metod. Należy odnotować, iż poniższy schemat ma wiele wariantów [20, 2]; dla uproszczenia rozważań prezentowana jest wersja bliska tzw. prostemu algorytmowi genetycznemu (ang. *simple genetic algorithm*). Prosty algorytm genetyczny możemy podzielić na dwa etapy.

- **Etap pierwszy** to **inicjacja i ocena populacji bazowej** B^0. Inicjacja populacji odbywa się w sposób losowy, po czym dokonywana jest początkowa ocena populacji przy pomocy tzw. **funkcji przystosowania**. Każdemu osobnikowi funkcja ta przydziela wartość liczbową, reprezentującą jakość opisywanego przez niego rozwiązania.
- **Drugim etapem** algorytmu jest pętla w której następuje **symulowanie ewolucji populacji bazowej**, aż do osiągnięcia określonego **warunku stopu**. Warunek stopu jest dobierany w zależności od rodzaju problemu rozwiązywanego przez algorytm genetyczny. Może to być osiągnięcie określonej liczby obrotów pętli algorytmu lub

wygenerowanie populacji zawierającej zadowalająco przystosowanego osobnika. Dla każdego obrotu pętli populacja bazowa jest poddawana następującym operacjom:

1. Na początku dokonywany jest wybór osobników do reprodukcji. Proces ten nazywany jest **selekcją**. Każdy osobnik może zostać wybrany z prawdopodobieństwem uzależnionym od jego poziomu przystosowania. Lepiej przystosowany osobnik będzie miał większą szanse na bycie wybranym. Następnie tworzone są kopie wyselekcjonowanych osobników, które w rezultacie tworzą **populację tymczasową** T^i, zwaną też **populacją rodzicielską**. Dzięki selekcji, w populacji tymczasowej zazwyczaj przeważają ponadprzeciętnie przystosowani osobnicy.
2. Osobniki z populacji tymczasowej poddawane są **krzyżowaniu**. Krzyżowanie polega na wymianie genotypów pomiędzy dwoma osobnikami rodzicielskimi, w wyniku której powstają nowi osobnicy. Każda rozdzielna para osobników rodzicielskich poddawana jest krzyżowaniu z ustalonym prawdopodobieństwem. Jeśli dojdzie do krzyżowania, to potomkowie osobników rodzicielskich dołączają do nowej populacji P^i zwanej **populacją potomną**. W przeciwnym wypadku do populacji potomnej P^i trafiają wybrani osobnicy rodzicielscy.
3. Następnie populacja potomna poddawana jest **mutacji**. Mutacja osobnika polega na perturbacji jego genotypu. Każdy gen w genotypie osobnika może zostać zmodyfikowany z zadanym prawdopodobieństwem, zaś efektem modyfikacji jest zmiana jego wartości na inną (np. zmiana 1 na 0 i vice versa w kodowaniu binarnym).
4. Nowo stworzona populacja potomna P^i zostaje poddana ocenie przy pomocy funkcji przystosowania. Na końcu pętli następuje selekcja osobników z populacji potomnej P^i i populacji bazowej B^i - etap ten nazywamy **sukcesją**. Najprostszym rodzajem sukcesji jest skopiowanie populacji potomnej. Jednak inne schematy sukcesji mogą być bardziej złożone i wykorzystywać populacje bazową. W wyniku sukcesji otrzymywana jest nowa populacja bazowa B^{i+1}.

Algorytm 1 Ogólny algorytm genetyczny [2, s. 66]

$i = 0$
inicjacja populacji B^0
ocena B^0
while not warunek stopu **do**
  T^i = reprodukcja B^i
  P^i = krzyżowanie i mutacja T^i
  ocena P^i
  B^{i+1} = sukcesja $(B^i,\ P^i)$
  $i = i + 1$
end while

2.3.2.1 Generowanie populacji początkowej

Jak już wspomniano osobnicy populacji początkowej generowani są w sposób losowy. Przykładowo, rozwiązując problem plecakowy opisany w sekcji 2.3.1, generowanie populacji początkowej polegać może na stworzeniu μ losowych wektorów charakterystycznych, gdzie μ to liczność populacji.

Pożądaną cechą populacji początkowej jest zróżnicowanie jej osobników. Jak wskazują autorzy [7] przy zadaniach optymalizacyjnych, gdzie nie znamy optymalnego rozwiązania, odpowiednio duża różnorodność populacji początkowej powinna przyspieszyć tempo generowania przez algorytm wysokiej jakości rozwiązań.

Rozmiar populacji jest czynnikiem wpływającym na różnorodność populacji oraz na efektywność i szybkość działania algorytmów genetycznych. Utrzymanie niewielkiej populacji początkowej argumentowane jest tym, że algorytm genetyczny w trakcie optymalizacji rozwiązań jest w stanie szybciej opuszczać maksima lokalne funkcji przystosowania. Wynika to z tego, że aby opuścić maksimum lokalne często musimy wpierw dopuścić do reprodukcji gorzej przystosowanych osobników. W liczniejszych populacjach każdy taki osobnik będzie konkurował z większą liczbą osobników najlepiej przystosowanych, co sprawi że rzadziej dojdzie do jego selekcji i w rezultacie reprodukcji. Z kolei, jak wskazuje Arabas [2, s. 227], zwolennicy liczniejszych populacji początkowych (liczonych w tysiącach) powołują się na **twierdzenie o schematach**, które krótko opisane zostanie w sekcji 2.3.4. Jednym z założeń tego twierdzenia jest nieskończoność populacji początkowej. Fakt ten mógłby przemawiać za podstawami do produkowania liczniejszych populacji początkowych. Dodatkowo wraz z zwiększeniem liczności populacji początkowej rośnie prawdopodobieństwo że będzie do niej należał osobnik znajdujący się w "obszarze przyciągania" maksimum globalnego, co znacząco przyspieszy generowanie rozwiązań wysokiej jakości.

Istnieją zadania gdzie przed inicjacją populacji początkowej można określić pewien obiecujący obszar przestrzeni rozwiązań od którego warto zacząć przeszukiwania. W takim wypadku możemy wyznaczyć zbiór rozwiązań należących do tego obszaru, nazywany **zbiorem zainteresowań**. Gdy zbiór zainteresowań może być określony, generowanie populacji początkowej obywa się w taki sposób aby każdy osobnik reprezentował rozwiązanie należące do tego zbioru.

2.3.2.2 Funkcja przystosowania i metody selekcji

Funkcja przystosowania jest narzędziem używanym w algorytmach ewolucyjnych do symulowania nacisku środowiska na populację osobników. Funkcja przystosowania przypisuje każdemu genotypowi wartość liczbową odzwierciedlającą jakość zakodowanego rozwiązania. Im lepsze rozwiązanie tym funkcja przyjmuje większą wartość, osiągając wartość maksymalną dla genotypu kodującego rozwiązanie optymalne. Wyznaczenie odpowiedniej funkcji przystosowania jest kluczowe dla efektywnego działania każdego algorytmu genetycznego.

Dla omawianego problemu plecakowego, gdzie każde rozwiązanie reprezentowane jest przy pomocy chromosomu binarnego, możemy zaproponować następującą funkcję przystosowania:

$$\Phi(X) = \begin{cases} 0 & \text{jeśli} \quad \sum_{i=1}^{n} w_i X_i > W \\ \sum_{i=1}^{n} p_i X_i & w\ przeciwnym\ wypadku \end{cases} \tag{2.2}$$

gdzie:

$X_i \in \{0, 1\}$	wartość i-tego genu w chromosomie,
$p_i \in \mathbb{R}$	wartość i-tego przedmiotu,
$w_i \in \mathbb{R}$	waga i-tego przedmiotu,
$W \in \mathbb{R}$	dopuszczalna waga plecaka.

Na podstawie wartości przypisanych przez funkcje przystosowania dokonuje się selekcji. Selekcja występuje w dwóch fazach algorytmu genetycznego - reprodukcji oraz sukcesji. W reprodukcji wybiera się osobniki, które zostaną następnie poddane krzyżowaniu i mutacji. Z kolei w sukcesji wybieramy osobników, z populacji potomnej i bazowej, którzy utworzą nową populacje.

Ważnym zagadnieniem selekcji jest tzw. **nacisk selektywny**, czyli intensywność z jaką będziemy premiować najlepiej przystosowanych osobników populacji. Zbyt duży nacisk selektywny będzie sprawiał że algorytm genetyczny szybko zatrzyma przeszukiwanie przestrzeni rozwiązań. Dzieje się tak ponieważ kopie pierwszego lepiej przystosowanego osobnika, tworzone w procesie reprodukcji, w krótkim czasie zdominują populacje - cała populacja będzie składała się wyłącznie z jego kopii. Stąd w każdej metodzie selekcji wprowadza się element losowości, by gorzej przystosowane osobniki również miały szanse na reprodukcję i sukcesję. Z drugiej strony, kładąc zbyt mały nacisk selektywny ryzykujemy, że najlepiej przystosowani osobnicy nie zostaną wybrani do następnej populacji. Stąd głównym zadaniem każdej metody selekcji jest zbalansowanie jakości wybranych osobników z ich różnorodnością. Poniżej przedstawiono trzy popularne schematy reprodukcji wykorzystujące różne podejścia do selekcji osobników.

- **Reprodukcja proporcjonalna**, zwana też ruletkową, polega na przypisaniu osobnikom prawdopodobieństwa wylosowania w sposób proporcjonalny do wartości ich funkcji przystosowania. Zazwyczaj, zarówno przy reprodukcji jak i sukcesji, mówimy o losowaniu ze zwracaniem, tj. dany osobnik może wziąć udział w wielu krzyżowaniach w bieżącej fazie.

- **Reprodukcja rangowa** różni się od reprodukcji proporcjonalnej tym, iż każdemu osobnikowi przypisywana jest **ranga**, tj. pozycja zajmowana przez osobnika w pokoleniu posortowanym nierosnąco względem wartości funkcji przystosowania. Następnie, podobnie jak w reprodukcji proporcjonalnej, przypisuje się każdemu osobnikowi prawdopodobieństwo które w tym wypadku powinno być proporcjonalne względem jego rangi. Możemy zauważyć że reprodukcja rangowa kładzie mniejszy nacisk selektywny niż reprodukcja proporcjonalna. By to zilustrować, przyjmijmy że mamy do czynienia z populacją zawierającą osobnika znacząco lepiej przystosowanego od pozostałych. W reprodukcji proporcjonalnej osobnik ten będzie miał zauważalnie większe prawdopodobieństwo bycia wylosowanym niż drugi najlepiej przystosowany osobnik. Z kolei w reprodukcji rangowej różnica ta będzie niewielka, bo choć rozbieżność w ich przystosowaniu będzie duża to różnica ich rang wynosić będzie tylko jeden.

- **Reprodukcja turniejowa** jest ostatnim z rozważanych tu sposobów reprodukcji. Polega ona na sukcesywnym losowaniu z populacji podzbiorów o określonej liczności t, będącej parametrem algorytmu. Liczba tak wylosowanych zbiorów równa jest wielkości populacji bazowej μ. Następnie, dla każdego podzbioru urządzany jest turniej, czyli po prostu wybór najlepiej przystosowanego osobnika.

Kontynuując przykład z problemem plecakowym możemy użyć reprodukcji rangowej o następującej funkcji $p_r(X)$ określającej prawdopodobieństwo selekcji:

$$p_r(X) \;=\; 1 \;-\; \frac{r(X)}{r_{max}} \tag{2.3}$$

gdzie:

$r(X) \in \mathbb{N}$ ranga chromosomu X
$r_{max} \in \mathbb{N}$ maksymalna ranga w populacji

Sukcesja jest kolejną fazą gdzie dokonuje się wyboru osobników. Metody selekcji wykorzystywane w sukcesji są zazwyczaj mniej złożone niż te wykorzystywane w reprodukcji i często poświęca się im mniej uwagi. Jednak należy pamiętać że sukcesja jest istotną fazą algorytmu genetycznego, zapewniającą jego poprawne działanie. Arabas [2, s. 129 - 134] mówi o trzech często stosowanych schematach sukcesji:

- **Sukcesja z całkowitym zastępowaniem** jest po prostu zastąpieniem całej populacji bazowej populacją potomną.
- **Sukcesja z częściowym zastępowaniem** polega na określeniu współczynnika wymiany η, gdzie $0 < \eta < 1$ a następnie zastąpieniu $\eta\%$ populacji bazowej osobnikami z populacji potomnej. Istnieją różne schematy zarówno usuwania osobników z populacji bazowej jak i wyboru ich następców.
- **Sukcesja elitarna** ma na celu zapewnienie przeżycia b najlepiej przystosowanych osobników z populacji bazowej. W tym celu przenosi się ich bezpośrednio do następnej populacji. Pozostałych osobników wybiera się do kolejnego pokolenia z połączenia populacji bazowej B^i i populacji potomnej P^i.

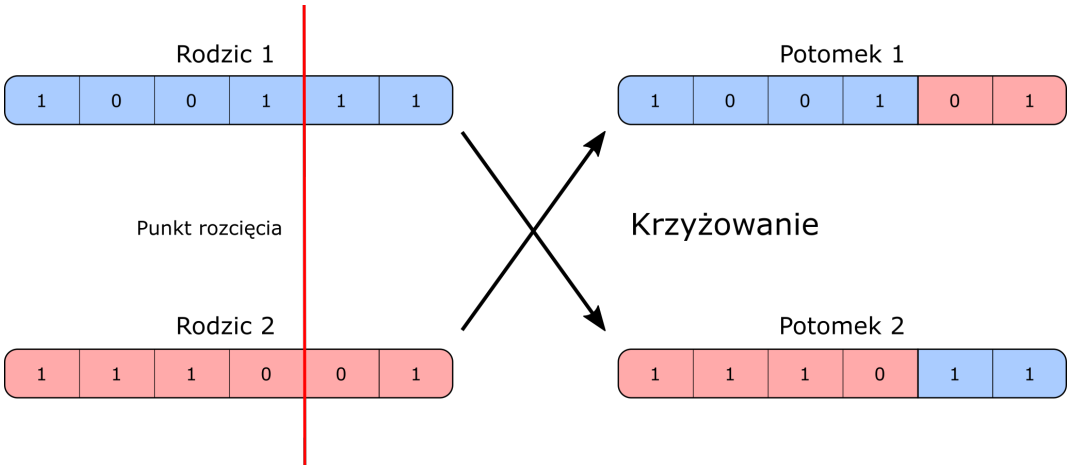
2.3.2.3 Krzyżowanie i mutacja

Krzyżowanie i mutacja są operacjami genetycznymi przekształcającymi genotypy osobników wyselekcjonowanych w fazie reprodukcji.

Krzyżowanie W krzyżowaniu najczęściej uczestniczy para osobników (nazywana też rodzicami), która generuje parę, lub, w innych podejściach, jednego potomka. Dla każdej pary rodziców krzyżowanie jest wykonywane z wybranym prawdopodobieństwem p_c, będącym parametrem algorytmu. Jeśli dojdzie do krzyżowania, wygenerowani potomkowie trafiają do populacji potomnej; w przeciwnym wypadku do populacji potomnej kopiowani są rodzice. Poniżej krótko opisano trzy popularne metody krzyżowania.

- **Krzyżowanie jednopunktowe**, jest najpopularniejszą metodą, polegającą na wylosowaniu punktu rozcięcia chromosomu. Punkt rozcięcia dzieli chromosomy obydwu

rodziców na dwie części, po czym odpowiednio pierwsza część chromosomu pierwszego rodzica jest łączona z drugą częścią chromosomu drugiego rodzica i na odwrót. W ten sposób otrzymujemy dwa nowe chromosomy, które będą składały się na genotyp dwóch osobników potomnych. Rys. 2.3 ilustruje krzyżowanie jednopunktowe w przestrzeni chromosomów kodowanych binarnie.

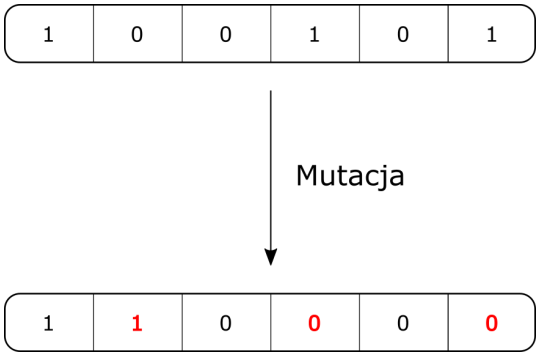


Rysunek 2.3: Ilustracja krzyżowania jednopunktowego dwóch chromosomów binarnych

- **Krzyżowanie równomierne** jest sposobem krzyżowania osobników w którym każdy gen w chromosomie osobnika potomnego wybierany jest spośród odpowiadających genów osobników rodzicielskich. Podstawowy wariant tej metody krzyżowania produkuje jednego potomka. Gen od pierwszego rodzica jest przekazywany z prawdopodobieństwem p_e. Gen od drugiego rodzica wybierany jest w przypadku, gdy nie przekazał go pierwszy rodzic. Zazwyczaj ustala się że $p_e = \frac{1}{2}$. W wariancie produkującym parę potomków, drugi potomek otrzymuje geny rodziców, które nie zostały przekazane potomkowi pierwszemu.

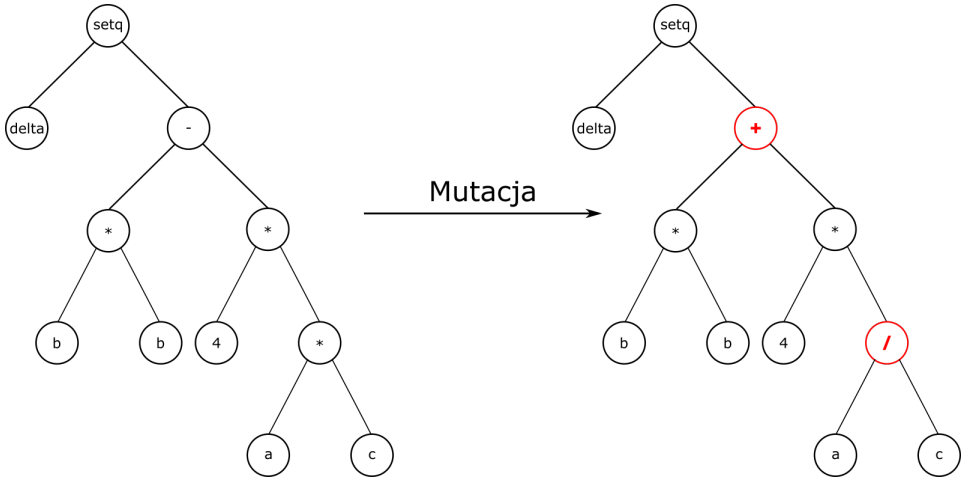
Mutacja Mutacja jest mechanizmem umożliwiającym algorytmom genetycznym przeszukiwanie przestrzeni rozwiązań w celu znalezienia nowych ekstremów lokalnych. Ściślej - mutacja, jako operacja czysto stochastyczna, może pozwolić na uniknięcie zbieżności obliczeń do danego ekstremum lokalnego i pokierować je potencjalnie w stronę ekstremów globalnych (o ile takie istnieją). Osobnicy z populacji potomnej poddani mogą być mutacji. Jak opisano w sekcji 2.3.2, mutacja polega na permutacji genotypu danego osobnika.

- **Mutacja Flip Bit** stosowana jest w przypadku reprezentacji binarnej. Polega na negacji bitu każdego genu z określonym prawdopodobieństwem p_m (będącym parametrem algorytmu). Rys. 2.4 ilustruje przykładową mutację Flip Bit dla chromosomu kodowanego binarnie.



Rysunek 2.4: Ilustracja mutacji Flip Bit dla chromosomu binarnego

- **Mutacja równomierna** stosowana jest dla kodowania wektorowego z liczbami rzeczywistymi. Polega na zmodyfikowaniu zastanej wartości na nową wylosowaną liczbę z danego przedziału. Przedział liczbowy z którego możemy losować nową wartość jest parametrem algorytmu. Dla mutacji równomiernej wykorzystuje się rozkład jednostajny.
- **Mutacja reprezentacji drzewiastej** polega na wykonywaniu jednej z predefiniowanych modyfikacji struktury drzewa (w przypadku programów w LISP może to być np. podmiana operatora w drzewie podformuły).



Rysunek 2.5: Ilustracja mutacji chromosomu w postaci drzewiastej

Spójność przestrzeni genotypów Warunkiem koniecznym do określenia poprawnych operatorów genetycznych jest zachowanie spójności przestrzeni genotypów. Aby ją zachować należy upewnić się, że dla każdego chromosomu należącego do zbioru zainteresowań istnieje ciąg operacji krzyżowania oraz mutacji który będzie w stanie wygenerować go z dowolnej populacji początkowej. Niespełnienie tego warunku może oznaczać że projektowany przez nas algorytm genetyczny jest z góry skazany na porażkę, gdyż rozwiązanie optymalne nie znajduje się w dostępnej dla nas przestrzeni genotypów.

Funkcja operatorów Oba operatory są istotne dla efektywności z jaką przeszukiwana jest przestrzeń rozwiązań. Krzyżowanie ma celu szybsze znalezienie najlepszego rozwiązania w obrębie "obszaru przyciągania" odkrytego ekstremum. Stąd krzyżowanie, podobnie jak metody selekcji, jest operacją która może przyspieszyć zbieżność algorytmów genetycznych, poprzez zmniejszenie różnorodności osobników w populacji. Odwrotny efekt daje operator mutacji. Jest on głównym mechanizmem dostarczającym do populacji osobników "egzotycznych", mocno różniących się od reszty populacji. Dzięki temu umożliwia na opuszczanie ekstremów lokalnych.

2.3.3 Parę słów o zbieżności algorytmów genetycznych

Jedno z teoretycznych uzasadnień efektywności działania algorytmów genetycznych oparte jest na **twierdzeniu o schematach**, sformułowanym przez Johna Hollanda [10] w roku 1975.

Schematy Schematy definiowane są jako ciągi złożone z zer, jedynek oraz znaków #, gdzie ostatni symbol oznacza dowolną wartość. Mówimy, że dany chromosom jest opisywany przez schemat, gdy jest z nim zgodny na wszystkich pozycjach na których nie występuje #. Schemat można więc traktować jako zbiór chromosomów przez niego opisywanych. **Rzędem schematu** $o(S)$ jest liczba znaków różnych od # w danym schemacie S, zaś **długością schematu** $d(S)$ nazwiemy odległość pomiędzy skrajnymi znakami rożnymi od #. Na przykład dla $S = \#\#100\#\#11$ mamy $o(S) = 5$ oraz $d(S) = 7$. **Wartością przystosowania schematów** $\Phi(S)$ jest średnia z wartości przystosowania wszystkich chromosomów opisywanych przez schemat.

Twierdzenie o Schematach Do sformułowania twierdzenia o schematach, prócz wyżej zdefiniowanych pojęć, przyjmuje się następujące założenia [2, s. 74]:

- populacja bazowa musi zawierać nieskończoną liczbę osobników,
- każdy możliwy schemat opisuje przynajmniej jednego osobnika należącego do populacji,
- algorytm genetyczny korzysta z selekcji proporcjonalnej,
- osobnicy reprezentowani są przy pomocy kodowania binarnego, oraz wykorzystywane jest krzyżowanie jednopunktowe i mutacja Flip Bit, której prawdopodobieństwo jest bliskim zeru.

Dla tych założeń możliwe jest oszacowanie oczekiwanej liczby chromosomów w następnej populacji bazowej $E(|S|_{B_{t+1}})$, które będą opisywane przez schemat S. Co przedstawia poniższy wzór [2, s. 74] (**Twierdzenie o Schematach**).

$$E(|S|_{B_{t+1}}) \ \geq \ |S|_{B_t} \frac{\Phi(S)}{\overline{\Phi}} \left(1 \ - \ p_c \frac{d(S)}{n \ - \ 1} \ - \ o(S) p_m\right) \tag{2.4}$$

gdzie:

$\|S\|$	liczebność schematu S
$\Phi(S)$	wartość przystosowania schematu S
$\overline{\Phi}$	średnie przystosowanie populacji
$d(S)$	długość schematu S
$o(S)$	rząd schematu S
n	długość chromosomu
p_c	prawdopodobieństwo krzyżowania
p_m	prawdopodobieństwo mutacji

Często formułowana interpretacja twierdzenia o schematach mówi, że jeśli istnieją w populacji osobnicy opisywani przez krótkie schematy (niskiego rzędu) które kodują pewną pozytywną cechę lub ich zestaw, to ich liczba będzie wzrastała w kolejnych populacjach.

Wbudowana równoległość Dzięki temu że jeden osobnik może być opisywany przez wiele schematów, zauważono że istnieje pewna równoległość w ich przetwarzaniu. Oznacza to, że w każdej populacji przetwarza się kilka schematów naraz. Stąd twierdzi się, że osobnicy ze sprzyjającymi cechami będą rozprzestrzeniać się w sposób wykładniczy w kolejnych populacjach. Obserwacja o równoległości przetwarzania schematów jest kolejną przesłanką wynikającą z twierdzenia o schematach przemawiającą za efektywnością algorytmów genetycznych. Jednocześnie należy nadmienić, iż związek twierdzenia o schematach i algorytmów genetycznych nie jest oczywisty i bywa kwestionowany [20].

2.3.4 Optymalizacja wielokryterialna

W wielu problemach optymalizacyjnych chcemy ocenić jakość danego rozwiązania pod względem wielu, czasami wykluczających się nawzajem, kryteriów. Mówimy wtedy że mamy do czynienia z zadaniem optymalizacyjnym wielokryterialnym. W tego typu zadaniach funkcja celu f jest zestawem m różnych funkcji oceniających poszczególne kryterium:

$$f(X) = (f_1(X), \ f_2(X), \ ..., \ f_m(X)). \tag{2.5}$$

Praktycznym celem tego typu zadań jest równoczesne zminimalizowanie wszystkich funkcji oceniającej każde kryterium. Jako przykład można podać zadanie produkcji komponentu hardware'owego w którym występuje kryterium ceny i wielkości produktu. Oczywiście niezwykle rzadko zdarza się, by rozwiązanie było równoczesnym minimum dla wszystkich kryteriów. Aby umożliwić porównanie rozwiązań w zadaniach wielokryterialnych zdefiniowano relację dominacji, którą oznaczamy przez symbol $\succ$. Rozwiązanie X dominuje nad rozwiązaniem Y wtedy, gdy dla przynajmniej jednego kryterium rozwiązanie X jest oceniane lepiej niż rozwiązanie Y, będąc jednocześnie ocenianym nie gorzej niż rozwiązanie Y w pozostałych kryteriach. Formalnie, mamy $X \succ Y$ wtedy i tylko wtedy gdy:

$$\forall k \ = \ 1, \ldots, \ m \ \ f_k(X) \ \leq f_k(Y) \ \wedge \ \exists k \ \ f_k(X) \ < f_k(Y). \tag{2.6}$$

Dla tak zdefiniowanej relacji dominacji, cel zadania wielokryterialnego można sformułować jako zadanie znalezienia zbioru rozwiązań niezdominowanych, czyli tzw. **zbioru Pareta** (często nazywanego też frontem Pareta).

2.4 Zastosowania algorytmów genetycznych w inżynierii oprogramowania

Algorytmy genetyczne, ze względu na swoją uniwersalność i efektywność, znajdują zastosowanie w szeregu różnych dziedzin, również inżynierii oprogramowania. Używane są miedzy innymi do automatycznego testowania oprogramowania oraz do identyfikacji i naprawy błędów programu.

Fuzzing Rodzajem zautomatyzowanego testowania oprogramowania jest fuzzing. Polega on na przekazywaniu do programu losowo wygenerowanych zestawów danych wejściowych w celu znalezieniu luk w programie. Wkład algorytmów genetycznych w fuzzing polega na generowaniu zestawów danych wejściowych, które maksymalizują pokrycie testowanego kawałka kodu. W ten sposób z kolejnymi generacjami wybrany program będzie testowany w coraz dokładniejszy sposób. Warto zauważyć że w algorytmie genetycznym używanym do fuzzingu kładziony jest duży nacisk na utrzymanie wysokiej różnorodności populacji w celu przetestowaniu programu pod każdym możliwym względem. Jednym z popularniejszych programów do fuzzingu, używającym algorytmów genetycznych, jest **AFL** (z ang. *American Fuzzy Lop*). Został on stworzony przez Michała Zalewskiego, i jest z sukcesem używany do wyszukiwania błędów m.in. w projektach takich jak Firefox, OpenSSL czy PHP [29].

Naprawa oprogramowania Idea zautomatyzowanej naprawy oprogramowania przy pomocy algorytmów genetycznych polega na potraktowaniu problemu jako zadania optymalizacyjnego. Celem zadania jest znalezienie łatki (ang. *patch*), która rozwiąże wszystkie testy podane przez użytkownika. W ten sposób przestrzeń rozwiązań możemy przedstawić jako przestrzeń wszystkich możliwych łatek, zaś algorytm genetyczny jest narzędziem jej przeszukiwania. Jednym z pionierskich implementacji tego podejścia jest GenProg [16](2009), natomiast bardziej współczesnym narzędziem jest omawiana w tej pracy ARJA [35] (2018). Niestety, rozwiązania korzystające z podejścia ewolucyjnego nie zdołały do tej pory osiągnąć skuteczności wystarczającej dla zastosowań komercyjnych. Jednak gigantyczne zapotrzebowanie związane z automatyczną identyfikacją i naprawą błędów sprawia że jest to dynamicznie rozwijająca się dziedzina badań.

Rozdział 3

Ewolucyjna Naprawa Programów

Głównym tematem niniejszego rozdziału jest przedstawienie procesu ewolucyjnej naprawy programów na przykładzie współczesnego narzędzia ARJA [35]. W dalszym ciągu opisane zostaną wybrane komponenty programu i kluczowe koncepty jego działania. Przedstawimy również krótki przegląd innych popularnych narzędzi.

3.1 Naprawa programów przy pomocy algorytmów genetycznych

Ilość zasobów (w tym czasu) przeznaczanych we współczesnych projektach informatycznych na identyfikację i naprawę błędów, sprawiła że wizja automatyzacji tego procesu zyskała zainteresowanie środowiska naukowego. Wraz z pojawianiem się nowych narzędzi uformowały się dwa podejścia do automatycznej naprawy oprogramowania:

- **Podejście semantyczne** – skierowane na zidentyfikowanie przez narzędzie źródła defektu i na podstawie zdobytej wiedzy skonstruowanie łatki naprawiającej wadliwy program.
- **Podejście "generate-and-validate"** – polegające na generowaniu dużej ilości zmodyfikowanych wersji wadliwego oprogramowania a następnie sprawdzaniu ich poprawności.

Narzędzia używające algorytmów genetycznych zaliczają się do podejścia "generate-and-validate". Łatki do oprogramowania generowane są za pomocą przeprowadzanych operacji genetycznych modyfikujących strukturę wyrażeń w programie. Walidacja odbywa się przy pomocy **wyroczni**. W przypadku rozwiązań używających algorytmów genetycznych, wyrocznia to **zbiór testów jednostkowych** dołączonych do wadliwego fragmentu kodu. Dzięki wyroczni możemy dokonać oceny jakości wygenerowanej łatki i tym samym efektywniej przeszukiwać przestrzeń rozwiązań programu. Jak łatwo zauważyć wymogiem koniecznym dla każdego takiego zbioru jest posiadanie przynajmniej jednego testu dla którego podany program nie wykonuje się poprawnie. Co ciekawe, możliwa jest naprawa oprogramowania przy wyroczni składającej się jedynie z testów wadliwie, co wykazana w testach na zbiorze QuickBugs [34].

Przetrenowanie łatki Istotnym konceptem odnoszącym się do ewolucyjnej naprawy oprogramowania jest przetrenowanie łatki. Łatka, która naprawia program tak, że przechodzi

on wszystkie testy nazywana jest **łatką adekwatną** (z ang. *adequate patch*). Podobnie jednak jak w innych gałęziach uczenia maszynowego, może okazać się, że łatka zmienia program w sposób tak istotny, iż oddala się on od swoich pierwotnych funkcji, zaś jego głównym celem staje się uzyskanie dobrego wyniku w testach. Łatkę taką nazywa się *przetrenowaną*. Ocena przetrenowania łatki musi zazwyczaj być dokonywana w sposób manualny, co może być przeszkodą w automatyzacji procesu naprawy. W praktyce, aby zmniejszyć prawdopodobieństwo wystąpienia przetrenowanej łatki należy upewnić się, że podany zbiór testów jest wystarczająco liczny i wystarczająco zróżnicowany, tzn. pokrywa większość kodu programu.

Założenie nadmierności Mówiąc o ewolucyjnej naprawie oprogramowania nie sposób nie wspomnieć o założeniu nadmierności (z ang. *redundancy assumption*). To na nim opierają się narzędzia naprawy programów używające algorytmów genetycznych. Założenie to mówi, że w większości przypadków aby naprawić błąd w programie wystarczy użyć kawałków kodu już w nim zawartych przy pomocy operacji takich jak np. zamiana kolejności linijek kodu. Ponieważ używane algorytmy genetyczne jedynie modyfikują istniejącą strukturę programu, ich efektywne funkcjonowanie zależy od prawdziwości założenia nadmierności. Istnieją próby empirycznego potwierdzenia założenia nadmierności, jak praca [19], analizująca commity z repozytoriów kodu dużych projektów informatycznych. Celem tej analizy jest sprawdzenie, czy commity, których celem jest naprawa błędów oprogramowania, zawierają nowe fragmenty kodu - wcześniej niespotykane w projekcie. Eksperyment wykazał, że 52% zbadanych commitów złożone jest wyłącznie z istniejącego już kodu. Wyniki te wydają się, przynajmniej częściowo, potwierdzać zasadność założenia nadmierności.

3.1.1 Istniejące narzędzia

Zarówno podejście semantyczne jak i "generate-and-validate" zrodziły pewną liczbę narzędzi do automatycznej naprawy błędów, z których jednak żadne (według obecnego stanu wiedzy autora) nie znalazło zastosowania komercyjnego. Poniżej przedstawimy listę wybranych narzędzi badawczych.

- **Nopol** [33] jest wiodącym narzędziem do automatycznej naprawy programów, reprezentującym podejście semantyczne. Nopol jest przeznaczony do naprawy wadliwie skonstruowanych instrukcji warunkowych. Nopol przyjmuje na wejściu wadliwy program oraz wyrocznię złożoną z testów jednostkowych. Program wykorzystuje podejście opierające się na wyliczaniu tzw. *wartości anielskich* (z ang. *angelic values*), tzn. wartościowań parametrów, które umożliwią poprawne wykonanie testów jednostkowych. Przy pomocy wartości anielskich Nopol przekształca naprawę programu na problem z zakresu teorii spełnialności modulo (z ang. *Satisfiability Modulo Theories*, SMT). Rozwiązanie tego problemu jest następnie zamieniane na kod tworzący łatkę naprawiającą. Nopol jest narzędziem, które zdołało naprawić największą liczbę błędów w [8, Figure 2], jednym z największych testów porównawczych (2141 błędów), jakie przeprowadzono na wybranych narzędziach automatycznej naprawy.
- **GenProg** [16] jest jednym z najstarszych i najpopularniejszych narzędzi do automatycznej naprawy programów. Wykorzystuje algorytmy genetyczne do przeszukiwania przestrzeni łatek, stąd zaliczany jest do podejścia "generate-and-validate". GenProg również używa wyroczni złożonej z zestawu testów jednostkowych w celu wyliczania

przystosowania osobników. Pierwotnie program stosował drzewiastą reprezentację łatek (w postaci kodu AST naprawianego programu), jednak w nowszych wersjach łatki reprezentowane są w postaci wektorowej. GenProg stał się punktem odniesienia dla wielu bardziej współczesnych narzędzi takich jak ARJA czy Kali.

- **Kali** [23] również opiera się na algorytmach genetycznych. Narzędzie zostało stworzone w celu zweryfikowania hipotezy postawionej przez twórców narzędzia, mówiącej, że większość łatek generowanych przez GenProg opiera się na operacji usunięcia wadliwego stwierdzenia w kodzie. Stąd też Kali bazuje w dużym stopniu właśnie na operatorze usuwania. Prócz usuwania stwierdzeń Kali zamienia również warunki w instrukcjach *if* oraz umieszcza instrukcje *return*.

- **RSRepair** [22] jest kolejnym narzędziem stworzonym jako odpowiedź na sukces GenProg. W odróżnieniu od GenProg RSRepair wykorzystuje przeszukiwanie losowe zamiast algorytmów genetycznych. Autorzy narzędzia postawili za cel wykazanie, że narzędzia zgodne z podejściem "generate-and-validate" niepotrzebnie korzystają z algorytmów genetycznych, które obarczone są dużo większym kosztem obliczeniowym niż przeszukiwanie losowe. RSRepair miał być równie skuteczny jak GenProg i jednocześnie potrzebować mniej czasu na wygenerowanie łatek, co wydają się potwierdzać wyniki z esperymentów przedstawionych w [8, Figure 2]. RSRepair rzeczywiście osiąga lepsze wyniki niż GenProg. Warto jednak zauważyć, że nowsze podejścia korzystające z algorytmów genetycznych, takie jak ARJA i Kali okazują się bardziej skuteczne niż RSRepair.

3.2 ARJA

Analiza przeprowadzona w [24] wykazała, że znacząca część łatek wygenerowanych przez GenProg zawierała błędy. Sukces RSRepair, podważający zasadność wykorzystywania algorytmów genetycznych, był kolejnym powodem przemawiającym za potrzebą nowego narzędzia. Skuteczność GenProg również pozostawiała wiele do życzenia. W teście porównawczym [8], przytaczanym w poprzedniej sekcji, dwie różne implementacje GenProg (GenProg-A i jGenProg) naprawiły odpowiednio 77 oraz 65 błędów z 2141 możliwych.

Powyższe problemy i ograniczenia stały się bezpośrednią motywacją dla autorów ARJi. ARJA jest narzędziem do automatycznej naprawy oprogramowania, wykorzystującym podejście "generate-and-validate". Podobnie jak GenProg i Kali ARJA stosuje algorytmy genetyczne do przeszukiwania przestrzeni łatek. Twórcy nowego narzędzia przeprowadzili wnikliwą analizę GenProg, która umożliwiła uniknięcia błędów swojego poprzednika; wprowadzono również liczne nowe rozwiązanie i udoskonalenia. W rezultacie ARJA regularnie osiąga najlepsze wyniki [8] [34] spośród narzędzi reprezentujących podejście "generate-and-validate", a w szczególności wyróżnia się wśród narzędzi opartych o algorytmy genetyczne. W dalszej części tego rozdziału, opierając się na informacjach z [35], opisane zostaną poszczególne elementy programu.

3.2.1 Przestrzeń łatek

Proces tworzenia przestrzeni łatek przez opisywane narzędzie możemy podzielić na trzy etapy. Pierwszym jest zidentyfikowanie miejsc w kodzie, które potencjalnie zawierają błąd. Następnie ARJA wybiera wyrażenia, za pomocą których podejmie próbę naprawy błędów

w danym miejscu. Na koniec ARJA koduje ciąg operacji na wyrażeniach wybranych w pierwszych dwóch etapach. Przy pomocy tych operacji modyfikowany będzie oryginalny kod, tworząc łatkę oprogramowania.

Punkty modyfikacji Pierwszym krokiem w celu skonstruowania przestrzeni łatek jest wybranie potencjalnie błędnych linijek kodu. W tym celu każdej linijce kodu przyporządkowana jest jej **"podejrzaność"** $susp \in [0,\ 1]$, reprezentująca szansę, że dana linijka zawiera błąd. Aby znaleźć wadliwe stwierdzenia ARJA korzysta z metody Ochiai [1]. Wykorzystując tę metodę podejrzaność linijki lc wyliczana jest przy pomocą poniższego wzoru [35, Równanie 2].

$$susp(lc)\ = \frac{N_{CF}}{\sqrt{N_F\ \times\ (N_{CF}\ +\ N_{CS})}} \tag{3.1}$$

gdzie:

N_{CS} liczba poprawnie wykonywanych testów pokrywających linijkę lc,
N_{CF} liczba wadliwie wykonywanych testów pokrywających linijkę lc,
N_F liczba wadliwie wykonywanych testów

Po obliczeniu podejrzaności dla każdej linijki, dokonywany jest ich wybór na podstawie dwóch parametrów - dolnego progu podejrzaności γ_{min} oraz maksymalnej liczby podejrzanych linijek n_{max}. Parametry oceniane są niezależnie: ARJA selekcjonuje wszystkie linijki lc dla których $susp(lc) > \gamma_{min}$, oraz n_{max} najbardziej podejrzanych linijek. Do dalszej analizy kierowane są wyniki procesu, który zwrócił mniejszą liczbę linijek. Wybrane linijki są miejscami w których algorytm genetyczny będzie dokonywał modyfikacji, stąd nazywane są **punktami modyfikacji**.

Odsiew wyrażeń Po ustaleniu punktów modyfikacji ARJA dokonuje dopasowania zbioru wyrażeń, które posłużą do ich modyfikacji. Zgodnie z **założeniem nadmierności** na początku za potencjalnych kandydatów brane są wszystkie linijki kodu. Następnie, wyrażenia poddawane są odsiewowi (z ang. *ingredient screening*) w celu wykluczenia tych, których użycie doprowadziłoby do błędu kompilacji. Działając w sposób statyczny, ARJA koncentruje się tu na często występującym problemie, który stosunkowo łatwo wykryć, tj. zastosowaniu wyrażenia które używa metod lub zmiennych niedostępnych w zakresie danego punktu modyfikacji.

Na przykład dla fragmentu kodu przedstawionego na listingu 3.1 nie jest możliwe użycie linijki 6 do zastąpienie linijki 10, ponieważ zmienna *newRadius* nie znajduje się w zakresie metody *getRadius()*. W trakcie odsiewu wyrażeń każda linijka poddawana jest podobnej analizie. Aby wyrażenie nie zostało odsiane musi używać wyłącznie zmiennych oraz wywołań metod które dostępne są w zakresie danego punktu modyfikacji. Pod koniec całego procesu każdy punkt modyfikacji powiązany jest ze zbiorem wyrażeń, których użycie jest bezpieczne pod względem zakresu zmiennych i metod.

```java
public class Circle {

    private double radius;

    public void setRadius(double newRadius) {
        this.radius = newRadius;
    }

    public double getRadius() {
        return radius;
    }
}
```

Listing 3.1: Przykładowy fragment kodu w języku Java)

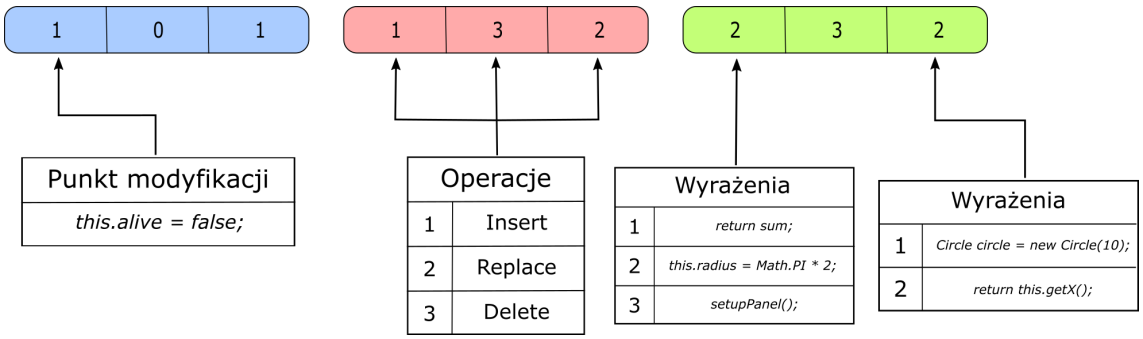
Reprezentacja łatki ARJA korzysta z **postaci wektorowej** do reprezentacji łatek. Każda łatka opisana jest przy pomocy trzech wektorów o tej samej długości.

1. Pierwszy wektor jest wektorem binarnym o długości n, gdzie n jest liczbą wybranych punktów modyfikacji. Każdy bit w wektorze przypisany jest do określonego punktu modyfikacji i koduje czy ma być on edytowany.

2. Drugi wektor zawiera informacje o rodzajach operacji, jakiej poddane mają zostać punkty modyfikacji. Każdej operacji przypisywany jest unikalny identyfikator w postaci liczby całkowitej. Przykładowo, jeśli i-ty punkt modyfikacji został wybrany do edycji w pierwszym wektorze, to zostanie on poddany operacji zakodowanej poprzez identyfikator w i-tym elemencie drugiego wektora. Podobnie jak w GenProg rozpoznaje się trzy rodzaje operacji na wyrażeniach: zastąpienie, usunięcie oraz wstawianie (*Replace*, *Delete*, *Insert*). Działanie dwóch pierwszych jest zgodne z intuicją, natomiast operacja wstawiania polega na umieszczeniu wybranego wyrażenia przed punktem modyfikacji.

3. Trzeci wektor koduje wyrażenia wykorzystywane do operacji zastąpienia i wstawienia. Każdy punkt modyfikacji ma swój własny zbiór bezpiecznych wyrażeń otrzymanych w wyniku odsiewu. Podobnie jak przy kodowaniu operacji, każde wyrażenie ma przypisany identyfikator - unikalny w obrębie wyrażeń danego punktu modyfikacji. W przypadku operacji usunięcia wyrażenie zakodowane w trzecim wektorze jest ignorowane.

Prezentowana na Rys. 3.1 łatka dokonuje dwóch operacji w oryginalnym kodzie oprogramowania. Dla pierwszego punktu modyfikacji, zawierającego wyrażenie *this.alive = false;* wykonywana jest operacja wstawienia. Jak możemy odczytać z trzeciego wektora, w odpowiednie miejsce wstawiony zostanie fragment *this.radius = Math.PI * 2;* Drugi punkt modyfikacji nie jest edytowany. Trzeci punkt modyfikacji poddany jest operacji zastąpienia: wyrażenie odpowiadające punktowi modyfikacji podmienione zostanie linijką *return this.getX();*



Rysunek 3.1: Przykładowa reprezentacja łatki w ARJA

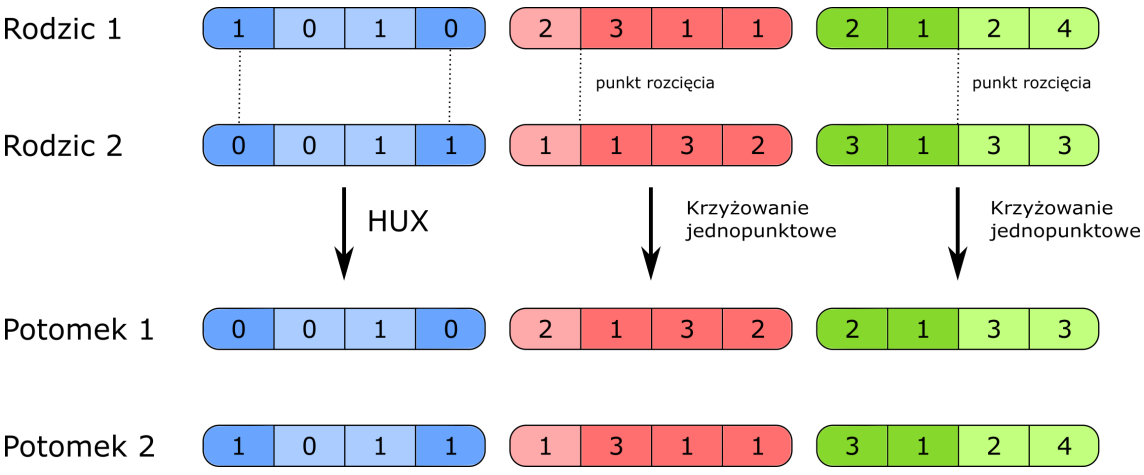
3.2.2 Operatory genetyczne

W niniejszej sekcji przedstawione zostaną dwa operatory genetyczne - krzyżowanie i mutacja - wykorzystywane do generowania populacji potomnych.

Krzyżowanie ARJA stosuje połączenie dwóch opisanych poniżej metod krzyżowania.

- Dla pierwszego wektora binarnego, mówiącego o tym czy dany punkty modyfikacji będzie edytowany, wykorzystywane jest **krzyżowanie HUX** (ang. *Half Uniform Crossover*). W metodzie tej wybrana w sposób losowy połowa genów różniących oba osobniki podlega wymianie, jak zilustrowano na Rys. 3.2 (lewa część).
- Dla pozostałych dwóch wektorów składających się na reprezentacje łatki, stosowane jest **krzyżowanie jednopunktowe**, opisane w poprzednim rozdziale.

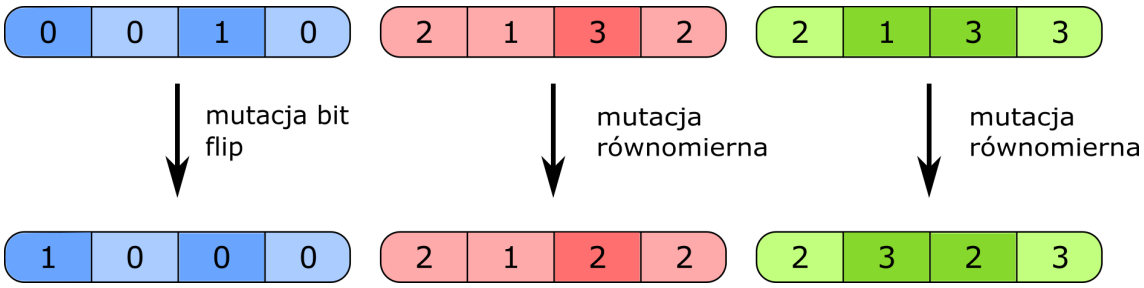
Para rodziców produkuje dwóch potomków w wyniku krzyżowania.



Rysunek 3.2: Przykładowe krzyżowanie łatek w ARJA

Mutacja Mutacja stosowana jest bezpośrednio po zakończeniu krzyżowania osobników. Analogicznie jak przy krzyżowaniu, pierwszy wektor poddawany jest innemu rodzajowi mutacji niż dwa pozostałe.

- Binarny wektor punktów modyfikacji ulega mutacji **Flip Bit**, polegającej na iterowanej negacji jego bitów z zadanym prawdopodobieństwem.
- Dwa pozostałe wektory poddawane są lekko zmodyfikowanej **mutacji równomiernej**, przy czym:
  - W przypadku wektora drugiego nowe wartości genów losowane są ze zbioru identyfikatorów dostępnych operacji, tj. $\{1, 2, 3\}$.
  - Trzeci wektor przypisuje punktom modyfikacji zbiory wyrażeń. Mutacja (jeśli zachodzi) musi być ograniczona do wyboru identyfikatora ze zbioru wyrażeń dostępnych dla danego elementu. W przypadku zilustrowanym na Rys. 3.1 można dostrzec, że dla pierwszego elementu w trzecim wektorze kandydat na nowy indentyfikator dla pierwszego punktu modyfikacji pochodzi ze zbioru $\{1, 2, 3\}$, zaś dla punktu trzeciego ze zbioru $\{1, 2\}$.



Rysunek 3.3: Przykładowa mutacja łatki w ARJA

3.2.3 Funkcja przystosowania

Po wygenerowaniu nowej populacji, każdy jej osobnik zostaje oceniony przy pomocy funkcji przystosowania. Twórcy ARJi formułują zadanie automatycznej naprawy jako problem wielokryterialny, gdzie każda łatka oceniana jest pod względem dwóch kryteriów - jej poprawności oraz długości. W tym celu zdefiniowano **dwuwymiarowa funkcję przystosowania** $F(x) = (f_1(x),\ f_2(x))$, gdzie x jest łatką oprogramowania, zaś funkcje składowe opisane są poniżej.

- Wartość $f_1(x)$ to liczba operacji, jakim poddany zostanie w wyniku zaaplikowania łatki oryginalny program. W omawianym przypadku jest to po prostu suma bitów pierwszego wektora łatki:

$$f_1(x) = \sum_{i}^{n} b_i \tag{3.2}$$

- Wartość $f_2(x)$ (zob. [35, Równanie 4]) wyraża ważony stosunek liczby wadliwych testów programu będący wynikiem zmodyfikowania przez łatkę x oryginalnego kodu do liczby wszystkich testów zawartych w wyroczni:

$$f_2(x) = \frac{|\{t \in T_f \mid x \text{ przechodzi } t\}|}{|T_f|} + w \times \frac{|\{t \in T_c \mid x \text{ nie przechodzi } t\}|}{|T_c|} \tag{3.3}$$

gdzie T_f to zbiór testów oryginalnie wadliwych, T_c to zbiór testów oryginalnie poprawnych, zaś w to uprzedzenie (z ang. *bias*).

Zbiory T_f i T_c w równaniu 3.3 wyznaczane są przed wygenerowaniem populacji początkowej, poprzez przetestowanie oryginalnej wersji programu. Należy wspomnieć że ARJA implementuje kilka metody mających na celu przyspieszenie późniejszego procesu oceny przystosowania osobników. W ich wyniku część testów może zostać odfiltrowana ze zbioru T_c. Wspomniane metody zostaną dokładniej omówione w sekcji 3.2.5 poświęconej metodom optymalizacji, jakie wykorzystuje ARJA. Warto tu zwrócić uwagę na trzy przypadki szczególne.

1. Pierwszym jest sytuacja gdy $f_1(x) = 0$, co oznacza, że łatka nie dokonuje żadnych modyfikacji oryginalnego kodu. Z punktu widzenia naprawy oprogramowania utrzymywanie takiej łatki w populacji mija się z celem, stąd też nadawana jest jej wartość $+\infty$. Powoduje to wyeliminowanie problematycznej łatki w następnej fazie reprodukcji, lub ewentualnie sukcesji (w zależności od używanej metody).

2. Kolejnym przypadkiem szczególnym jest wystąpienie łatki, która prowadzi do niepowodzenia w kompilacji programu lub przekroczenia zadanego limitu czasowego wykonania testów. W celu usunięcia jej z populacji postępuje się jak w przypadku pierwszym.

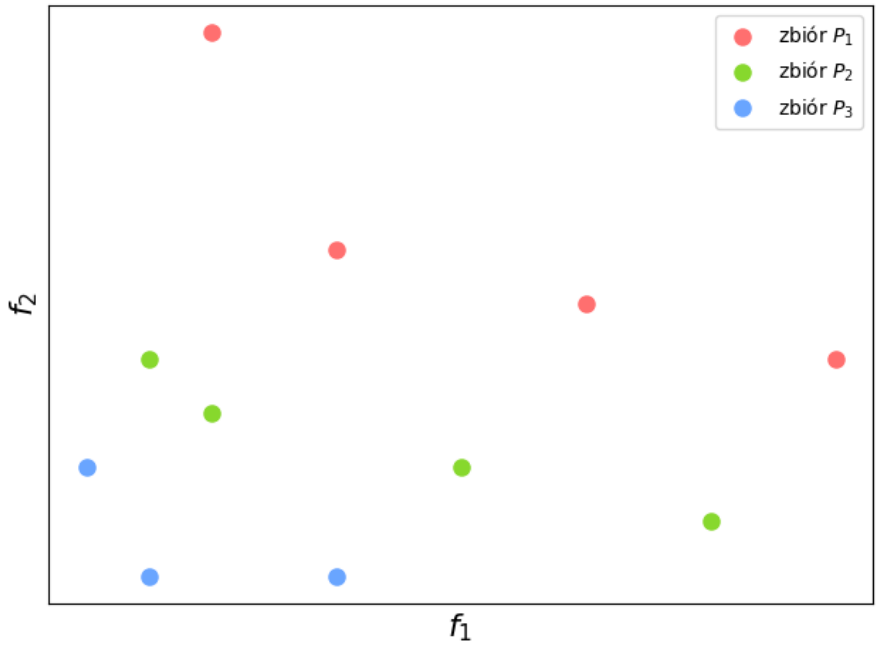
3. Trzeci przypadek szczególny zachodzi, gdy łatka przejdzie wszystkie testy pomyślnie, czyli gdy $f_2(x) = 0$. W efekcie, w każdej następnej generacji znajdą się tylko rozwiązania należące do zbioru $\{x \in P\colon f_2(x) = 0\}$, gdzie P to zbiór Pareta (zbiór rozwiązań niezdominowanych) wygenerowanej populacji.

3.2.4 NSGA-II

ARJA wykorzystuje algorytm genetyczny NSGA-II [6] (*Nondominated Sorting Genetic Algorithm II*), w celu przeszukiwania przestrzeni łatek. Algorytm ten został zaprojektowany z myślą o zadaniach wielokryterialnych. W tym celu twórcy NSGA-II zaprojektowali szybkie niedominujące sortowanie - wyspecjalizowaną metodę sortowania rozwiązań. Wykorzystali również techniki przeciwdziałające koncentracji przeszukiwań przestrzeni wokół jednego optimum. Metody te używane są by wspomóc proces selekcji w wyborze niezdominowanych i zróżnicowanych rozwiązań.

Szybkie niedominujące sortowanie Celem tej metody sortowania jest podzielenie zbioru rozwiązań na zbiory Pareta kolejnych poziomów. Wpierw wyznaczany jest zbiór Pareta najwyższego poziomu P_1, tzn. zbiór rozwiązań niezdominowanych. Aby wyznaczyć zbiór drugiego poziomu P_2 usuwa się wszystkie rozwiązania należące do zbioru P_1 i powtarza procedurę wyznaczania rozwiązań niezdominowanych. Cały proces kontynuuje się, dopóki każde rozwiązanie nie będzie przypisane do któregoś z wyznaczonych zbiorów Pareta. Na Rys. 3.4 przedstawiono przykładowe podzielenie zbioru rozwiązań na zbiory Pareta trzech kolejnych stopni.

Opisana procedura ma złożoność obliczeniową $O(MN^2)$ [6, 26], gdzie M jest liczbą kryteriów pod względem jakich rozwiązanie będzie oceniane, a N to rozmiar populacji. Jest to złożoność będąca na równi z innymi wiodącymi metodami niedominującego sortowania. Należy zauważyć, że choć rozwiązania zostaną rozdzielone do odpowiednich zbiorów



Rysunek 3.4: Przykładowe podzielenie rozwiązania na zbiory Pareta

Pareta, to wciąż trzeba je posortować w ich obrębie. W tym celu twórcy NSGA-II wykorzystują opisany dalej operator "Crowded-Comparison".

Operator "Crowded-Comparison" Operator oznaczany symbolem $\succ_n$, jest relacją porządku częściowego na przestrzeni rozwiązań. Umożliwia on posortowanie rozwiązań mając na uwadze zarówno ich zróżnicowanie, jak i przydział do odpowiednich zbiorów Pareta. Operator wykorzystuje dwie wartości wyliczane dla każdego rozwiązania r:

- rangę rozwiązania $r.rank$, tzn. poziom zbioru Pareta do jakiego dane rozwiązanie należy;
- miarę $r.distance$ (nazywaną "crowding-distance"). Dla danego rozwiązania reprezentuje ona zagęszczenie rozwiązań w jego sąsiedztwie. Proces wyliczania miary "crowding-distance" (Alg. 2) jest następujący. Jeśli istnieje kryterium według którego dane rozwiązanie jest skrajnym, to ustawiana jest wartość $r.distance = \infty$. W przeciwnym wypadku dla danego rozwiązania $R[i]$ wartość $r.distance$ jest równa sumie długości znormalizowanych przedziałów pomiędzy najbliższymi sąsiadami $R[i]$ względem wszystkich kryteriów. Złożoność czasowa funkcji "crowding-distance" to $O(MNlogN)$.

Używając tych dwóch wartości możemy zdefiniować relację $\succ_n$:

$$\begin{aligned} r_1 \succ_n r_2 \iff & r_1.rank < r_2.rank \vee \\ & (r_1.rank = r_2.rank \ \wedge \ r_1.distance > r_2.distance) \end{aligned} \tag{3.4}$$

Intuicyjnie, spośród dwóch rozwiązań lepszym jest dominujące; w przypadku zaś, gdy nie ma takiego - mniej zagęszczone.

Algorytm 2 Algorytm wyliczania "crowding-distance" [6, s. 4]

▷ Uwaga: $f_k^{max/min}$ to maks./minim. wartość kryterium k w zbiorze rozwiązań
$N = |R|$
for $i = 1$ to N **do**
  $R[i].distance = 0$
end for
for each kryterium k **do**
  $R = \text{sort}(R, k)$
  $R[1].distance = R[N].distance = +\infty$
  for $i = 2$ **to** $(N - 1)$ **do**
    $R[i].distance$ += $(R[i+1].k - R[i-1].k)$ / $(f_k^{max} - f_k^{min})$
  end for
end for

Główna pętla Algorytm NSGA-II wprowadza kilka modyfikacji w stosunku do ogólnego algorytmu genetycznego zaprezentowanego w sekcji 2.3.2. Po pierwsze, dla każdej populacji bazowej B^i pamiętana jest poprzednia populacja B^{i-1} (z wyjątkiem populacji początkowej B^0). Po drugie, faza sukcesji odbywa się na początku pętli i jest wykonywana na populacjach B^i oraz B^{i-1}. Po trzecie, zarówno faza sukcesji jak i selekcji wykorzystują szybkie niezdominowane sortowanie oraz relację "crowded-comparison" $\succ_n$. Algorytm 3 pokazuje sposób w jaki NSGA-II generuje kolejne populacje.

Algorytm 3 Algorytm NSGA-II

$Q = \emptyset$
$i = 0$
inicjacja populacji B^0
ocena B^0
while not warunek stopu **do**
  B^{i+1} = sukcesja (B^i, Q)
  T^{i+1} = reprodukcja B^{i+1}
  B^{i+1} = krzyżowanie i mutacja T^{i+1}
  ocena B^{i+1}
  $Q = B^i$
  $i = i + 1$
end while

Zmodyfikowany schemat sukcesji został przedstawiony w algorytmie 4. Na początku wykonywane jest szybkie niedominujące sortowanie na sumie populacji bazowej B^i i populacji poprzedniej (zapamiętywanej w zmiennej Q). W rezultacie otrzymujemy zbiór P składający się ze zbiorów Pareta kolejnych poziomów $(P^1, P^2, P^3, ...)$. Do nowej populacji B^{i+1} sumowane są kolejne zbiory Pareta aż do momentu gdy dany zbiór P^t nie zmieści się w całości w populacji. W międzyczasie obliczamy też "crowding-distance" dla poszczególnych zbiorów P^t. Zbiór, który nie pomieścił się w populacji B^{i+1}, zostaje posortowany przy pomocy operatora "crowded comparison" $\succ_n$. Następnie najlepsze rozwiązania z posortowanego zbioru trafiają do populacji B^{i+1} aż do jej zapełnienia. Wynikiem procesu sukcesji jest nowa populacja B^{i+1}.

Algorytm 4 Schemat sukcesji w NSGA-II [6, s. 5]

P = szybkie niedominujące sortowanie($B^i \cup Q$)
$B^{i+1} = \emptyset$
$t = 1$
while $|B^{i+1}| + |P^t| \leq N$ **do**
    wylicz "crowding-distance"(P^t)
    $B^{i+1} = B^{i+1} \cup P^t$
    $t = t + 1$
end while
sort(P^t, $\succ_n$)
$B^{i+1} = B^{i+1} \cup P^t[1 : (N - |B^{i+1}|)]$

Używany schemat reprodukcji również jest lekko zmodyfikowany. Przebiega on tak samo jak **reprodukcja turniejowa** z jedną różnicą - zwycięzca turnieju jest wybierany przy pomocy operatora "crowded comparison". Wykorzystywane operatory genetyczne oraz funkcja przystosowana dostarczane są już przez samą ARJA i zostały opisane w poprzednich sekcjach.

3.2.5 Parę słów o optymalizacjach

Jak wspomniano w sekcji 3.2.3, ARJA używa metody, które mają za zadanie optymalizować czas obliczania przystosowania osobników. Prócz optymalizacji pracy funkcji przystosowania, ARJA stara się również ograniczyć, w inteligentny sposób, przestrzeń łatek. Ograniczona przestrzeń pozwoli NSGA-II na efektywniejsze przeszukiwanie rozwiązań.

Optymalizacja pracy funkcji przystosowania Na początku omówiony zostanie wspomniany już mechanizm filtracji testów. Jego działanie jest stosunkowo proste.

W trakcie pierwszego wykonania testów z wyroczni (jeszcze na oryginalnej wersji programu), ARJA zapamiętuje wszystkie linijki pokryte przez poszczególny test. Jeśli dany pozytywny test nie pokrywa żadnej linijki uznanej za podejrzaną (czyli nie pokrywa żadnego punktu modyfikacji) zostaje on odfiltrowany. Ponieważ żadna zmiana wprowadzona przez łatkę nie wpłynie na rezultat takiego testu, operacja ta jest całkowicie bezpieczna – przy założeniu, że program wykonuje się w sposób deterministyczny.

Dodatkowo ARJA dostarcza kolejne dwie metody pozwalające w dalszym stopniu przyspieszyć fazę oceny osobników. Po pierwsze, możliwe jest ograniczenie liczby operacji wykonywanych przez łatkę. Jeśli łatka przekroczy ustalony limit operacji n_o to wszystkie operacje nadmiarowe zostają usunięte. Pozostawiane są operacje wykonywane na punktach modyfikacji z największą podejrzanością. Drugą metodę ARJA zapożycza od narzędzia GenProg, zaś wykorzystywana jest, gdy zbiór testów pozytywnych T_c dalej ma zbyt duży rozmiar. Metoda ta polega na losowaniu za każdym razem innego podzbioru testów ze zbioru T_c. Następnie, przy pomocy wylosowanego podzbioru wyliczana jest funkcja f_2 (przedstawiona w równaniu 3.3), będąca składową funkcji przystosowania. Dopiero, gdy dla jakiegoś podzbioru łatka osiągnie $f_2(x) = 0$, ponownie wylicza się przystosowanie już na pełnym zbiorze T_c.

Ograniczenie przestrzeni łatek Twórcy ARJA określili 14 reguł mających przeciwdziałać wykonywaniu operacji mijających się z celem, niewpływających na poprawność programu

lub prowadzących do błędów kompilacji. Dzięki doprecyzowaniu dozwolonych operacji ogranicza się liczbę możliwych łatek, a tym samym przestrzeń rozwiązań. Reguły podzielono na trzy kategorie.

1. Do pierwszej kategorii należą dwie reguły [35, tabela 1]. Wyróżniają się tym, że **dostosowują typ operacji** do wybranych punktów modyfikacji. Pierwsza reguła zakazuje usuwania wyrażeń zawierające deklaracje zmiennych - wykonanie takiej operacji prawie zawsze prowadzi do nieskompilowania programu. Druga reguła zakazuje usuwania wyrażeń zawierających słowa kluczowe *return/throw* we wszystkich metodach, z wyjątkiem metod zadeklarowanych jako *void*. Brak zwracania wartości dla tego typu metod będzie prowadzić do błędu kompilacji.

2. Druga kategoria składa się z reguł **ograniczających typy wyrażeń** wykorzystywanych do edytowania punktów modyfikacji. Na kategorię składa się 6 reguł [35, tabela 2]. Dotyczą one w większości używania stwierdzeń zawierających słowa kluczowe, których wykorzystanie ma sens tylko w konkretnym kontekście. Mowa tu o słowach kluczowych takich jak *break*, *continue* czy *case*. Przykładowo, druga reguła z tej kategorii określa, że słowo kluczowe *break* może być wykorzystane tylko wewnątrz wyrażenia *switch* lub pętli.

3. Ostatnia kategoria składa się z reguł **zakazujących konkretnych operacji**. Również zawiera 6 reguł [35, tabela 3], mówiących głownie o wykonywaniu operacji, które nie mają wpływu na poprawność programu. Między innymi zastępowanie wyrażenia samym sobą lub wyrażeniem posiadającym takie samo drzewo AST (reguła 1). Dodatkowo podano też dwie reguły odnoszące się do słowa kluczowego *return*, np. regułę mówiącą o niewstawianiu *return* w sposób doprowadzający do pojawienia się nowych wyrażeń nieosiągalnych.

3.2.6 Uruchomienie ARJA: wymagania i opis środowiska

Kod źródłowy ARJA jest publicznie dostępny na platformie Github [1]. Do uruchomienia programu konieczny jest system operacyjny Mac OS X lub Linux. Niezbędny będzie również Java JDK kompatybilny z lekko przestarzałą wersją 1.7. Po spełnieniu tych dwóch warunków możliwe jest skompilowanie i uruchomienie narzędzia.

ARJA jest programem uruchamianym z wiersza poleceń. Minimalnie, należy przekazać cztery parametry:

1. Parametr *-DsrcJavaDir* określający ścieżkę do kodu źródłowego wadliwego programu.

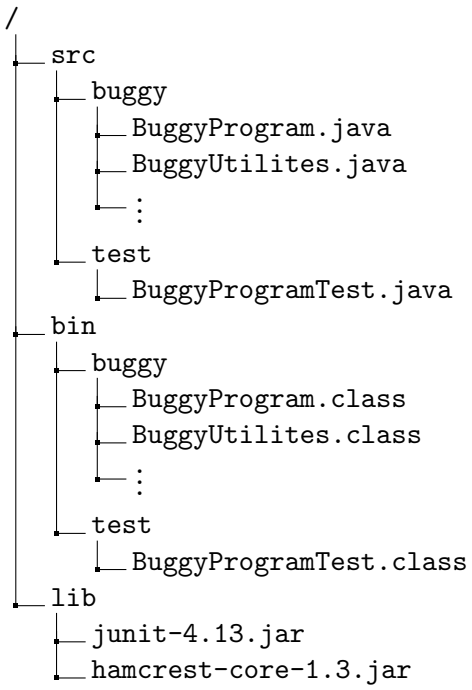
2. Parametr *-DbinJavaDir* określający ścieżkę do skompilowanego kodu wadliwego programu.

3. Parametr *-DbinTestDir* określający ścieżkę do skompilowanego kodu testów jednostkowych (wyroczni).

4. Parametr *-Ddependences* określający ścieżki do jakichkolwiek zależności wadliwego programu (pliki .jar).

[1] ARJA, https://github.com/yyxhdy/arja

Jeśli do skompilowania wadliwego programu wymagany jest więcej niż jeden plik .jar, to należy oddzielić odpowiednie ścieżki symbolem ":". Należy również zaznaczyć, że ARJA w tej chwili wspiera jedynie **ścieżki absolutne**. Każda łatka pomyślnie przechodząca wszystkie testy zapisywana jest domyślnie do folderu `arja/patches_id`, gdzie id to losowo generowany ciąg czterech liter. Przy pomocy parametru *-DpatchOutputRoot* można wskazać wybrany folder, w którym zapisywane będą łatki. Poniżej zamieszczono przykładową strukturę drzewa folderów dla wadliwego programu oraz komendę uruchamiającą próbę jego naprawy przy pomocy ARJA.



Rysunek 3.5: Drzewo folderów dla przykładowego wadliwego programu

```
$ java -cp lib/*:bin us.msu.cse.repair.Main Arja \
        -DsrcJavaDir <Absolute path>/src \
        -DbinJavaDir <Absolute path>/bin \
        -DbinTestDir <Absolute path>/bin \
        -Ddependences <Absolute path>/lib/junit-4.13.jar:<Absolute
           path>/lib/hamcrest-core-1.3.jar
```

Listing 3.2: Komenda uruchomienia ARJA dla programy z rysunku 3.5

Warto zauważyć, że ARJA wykryje tylko testy jednostkowe zrealizowane przy pomocy popularnej biblioteki JUnit. Każdy plik zawierający odpowiednie adnotacje, dostarczane przez bibliotekę, zostanie automatycznie zidentyfikowany jako część wyroczni. Wszystkie inne pliki we wskazanym folderze zostaną uznane za część wadliwego programu i będą mogły zostać poddane modyfikacji. ARJA dostarcza szeregu parametrów, przy pomocy których możliwe jest dostosowanie jej pracy. Poza opisanymi wcześniej, warto tu wspomnieć o parametrze *-DmaxGenerations*, określającym maksymalną liczbę generacji. Osiągnię-

cie limitu generacji jest warunkiem stopu dla algorytmu genetycznego wykorzystywanego przez ARJA. Strona Github projektu dostarcza więcej informacji o wszystkich dostępnych parametrach.

Rozdział 4

Próby Naprawy Programów Współbieżnych

Niniejszy rozdział poświęcony jest próbom naprawy programów współbieżnych przy pomocy narzędzia ARJA [35]. Część teoretyczna tego rozdziału oparta jest w większości na [4, 9]. Na wstępie omówiono pojęcie współbieżności oraz rodzaje problemów jakie z niej wynikają. W szczególności, zarysowano problem wzajemnego wykluczenia. W dalszej części opisany został, rozwiązujący ten problem, algorytm Petersona wraz z jego sekwencyjną wersją, przedstawiono ich uszkodzoną implementację oraz zaprezentowano wyniki z prób ich automatycznego naprawienia przez ARJA.

4.1 Współbieżność w zarysie

Przetwarzanie współbieżne polega na równoczesnym (pozornym, lub faktycznym) wykonywaniu czynności przez wiele procesów współdzielących pewne zasoby. W przetwarzaniu współbieżnym jeden proces może rozpocząć swoje działanie przed zakończeniem pracy innego. We współczesnych systemach współbieżność realizowana jest przy pomocy dwóch rodzajów przetwarzania.

- **Przetwarzanie w przeplocie** jest naprzemiennym wykonywaniem instrukcji procesów. Opisana funkcjonalność realizowana jest przy pomocy **zegara procesora**. Co ustalony okres, nazywany **kwantem czasu**, zegar wysyła sygnał pod wpływem którego system operacyjny wstrzymuje wykonywanie dotychczasowego procesu oraz dopuszcza do procesora inny oczekujący proces. Operacja ta nazywana jest **wywłaszczeniem**. Jak łatwo zauważyć, w opisywanym rodzaju przetwarzania nie mamy do czynienia z prawdziwie równoległym wykonywaniem procesów - instrukcje realizowane są w sposób sekwencyjny. Jednak wystarczająco szybkie przełączanie się pomiędzy procesami stwarza, z perspektywy użytkownika, iluzję równoległego działania. Zaletą przetwarzania w przeplocie jest możliwość znaczącej poprawy wykorzystywania zasobów obliczeniowych procesora. Przykładowo, jeśli dany proces oczekuje na zakończenie operacji wejścia/wyjścia inny proces może w tym czasie wykonywać własne obliczenia.

- **Przetwarzanie równoległe** jest możliwe, gdy posiadany procesor ma wiele rdzeni lub gdy dysponujemy systemem z wieloma procesorami. Jest to przetwarzanie praw-

dziwie równoległe. Instrukcje poszczególnych procesów wykonywane są w tym samym czasie przez różne procesory lub przez różne rdzenie procesora. Współcześnie większość komputerów posiada procesor wielordzeniowy lub korzysta z architektury wieloprocesorowej i tym samym umożliwia ten rodzaj przetwarzania. Należy podkreślić, że przetwarzanie równoległe nie wyklucza użycia przetwarzania w przeplocie. Często będzie się zdarzać, że wszystkie rdzenie będą zajęte własnymi obliczeniami. Sytuacja gdy jeden z nich będzie wykorzystywał zasoby obliczeniowe oczekując na długotrwałą reakcje użytkownika, jest tak samo nieuzasadnionym marnotrawstwem jak w przypadku systemu z jednym procesorem. Korzyści płynące z przetwarzania w przeplocie - tzn. efektywniejsze wykorzystanie pracy procesora - sprawiają, że ten rodzaj przetwarzania znajduje zastosowanie również w architekturach wieloprocesorowych.

Dalsze rozważania o programach współbieżnych będą opierały się na **czterech założeniach** dotyczących środowiska w którym owe programy operują. Podobne założenia będą obowiązywały w trakcie analizy prób napraw wybranych programów współbieżnych.

1. Przyjmuje się założenie, że kilka procesów może wykonywać się niezależnie, z dowolną prędkością. Innymi słowy zakłada się, że procesy nie korzystają ze wspólnego zegara. Choć w rzeczywistości może zdarzyć się inaczej, to fakt ten nie zmienia sposobu w jaki projektowane są programy współbieżne. Programista nie wie, czy tworzony przez niego program będzie wykonywany na architekturze jedno- czy wieloprocesorowej. W przypadku tej drugiej, każdy procesor może przydzielać różne kwanty czasu na wykonanie instrukcji procesów. Nie posiadamy również informacji, ile czasu zajmuje samo wywłaszczenie. Stąd też w dalszej analizie będzie zakładane, że procesy przetwarzane są w sposób niezależny.

2. Przyjmuje się, że procesy te mogą współdzielić pewne obszary pamięci, w celu komunikacji i synchronizacji. W szczególności, programy współbieżne omawiane w dalszej części rozdziału będą opierały się na wykorzystaniu **wątków**. Wątki są fragmentami programu wykonującymi się współbieżnie w obrębie jednego procesu. Wyróżniającą je cechą jest współdzielenie pamięci adresowej przydzielonej procesowi w ramach którego funkcjonują. Podobnie jest z innymi zasobami, takimi jak na przykład gniazda. W porównaniu do wątków, współdzielenie zasobów pomiędzy procesami odbywa się rzadziej i jest dużo bardziej restrykcyjne. W dalszej części tego rozdziału wszystkie stwierdzenia odnoszące się do procesów będą równie prawdziwie dla wątków, chyba że wyraźnie zaznaczono inaczej.

3. Przyjmuje się, że aplikacje rozproszone i procesy uruchomione są w pewnym systemie operacyjnym, który wyposażony jest w **dyspozytora**. Dyspozytor jest częścią systemu operacyjnego, który decyduje o tym, który proces ma być jako następny dopuszczony do procesora. Zakładane jest, że dyspozytor jest wywłaszczający. Istnieją architektury, gdzie to poszczególne procesy decydują o zrzeczeniu się wykonywania, nie są jednak zbyt popularne, ponieważ wystarczy istnienie jednego "chciwego" procesu do destabilizacji całego systemu operacyjnego. Różne systemy operacyjne wykorzystują różne dyspozytory w celu szeregowania procesów. Algorytm może zależeć od architektury komputera oraz tego, czy do dyspozycji jest jeden, czy więcej procesorów. Wszystkie dyspozytory gwarantują **uczciwość** przydzielania czasu procesorów

- tzn. każdy aktywny proces otrzyma, wcześniej czy później, możliwość wykonania przynajmniej jednej instrukcji. Jednocześnie każdy dyspozytor ma jakiś mechanizm przyznawania priorytetu kluczowym procesom, ponieważ niektóre procesy krytyczne dla funkcjonowania systemu operacyjnego nie mogą zwlekać z wykonaniem. Z perspektywy dalszej analizy, założenie o istnieniu dyspozytora daje gwarancje, że każdy proces zostanie kiedyś dopuszczony do wykonania.

4. Przyjmuje się, że system operacyjny daje też mechanizm, który pozwala aktywnemu procesowi na wysłanie do dyspozytora prośby o zawieszenie jego wykonania oraz prośby o wznowienie pracy innego procesu. Opisany mechanizm realizowany jest przy pomocy **semaforów**. Semafor jest typem danych posiadającym dwie operacje: operację V (nazywaną również *up*) oraz operacje P (analogicznie nazywaną *down*). Klasycznie, implementowany jest przy pomocy zmiennej całkowitej o dodatnich wartościach. Operacja V inkrementuje zmienną, zaś operacja P oczekuje w sposób nieangażujący procesor (tzn. oddając kontrolę dyspozytorowi) aż wartość zmiennej będzie dodatnia, a następnie ją dekrementuje. Założeniem semafora jest niepodzielność operacji V i dekrementacji w operacji P.

Algorytm 5 Pseudokod klasycznego semafora [31]

```
procedure V(S)
    S = S + 1
end procedure

procedure P(S)
    repeat
        Wait()
    until S > 0
    S = S − 1
end procedure
```

Popularnym rodzajem semafora jest semafor binarny, często nazywany **mutexem**. Realizowany jest przy pomocy zmiennej binarnej - przyjmującej jedynie wartość 1 lub 0. Implementacja poszczególnych operacji zmienia się w następujący sposób:

- W operacji V, zamiast inkrementacji zmiennej jej wartość ustawiana jest na 1.
- W operacji P, zamiast dekrementacji, zmienna jest ustawiana na 0. (Oczywiście wykonanie P na semaforze o wartości 0 sprawi, że proces zostanie wstrzymany, zaś jego wykonywanie zostanie wznowione tylko gdy inny proces wykona operację V.)

Założenie o istnieniu semaforów pozwala na budowanie złożonych mechanizmów synchronizacji i komunikacji pomiędzy procesami.

4.1.1 Podstawowe problemy współbieżności

Programy współbieżne wykorzystywane są powszechnie w praktyce obliczeniowej od ponad pięćdziesięciu lat. W obecnych czasach prawdopodobnie najbardziej rozpowszechnionym

przykładem programów współbieżnych są aplikacje serwerów, które muszą być w stanie obsłużyć wiele żądań równocześnie. Nawiązując do tematyki tej pracy, współbieżność stosuje się często do równoległej oceny osobników w algorytmach ewolucyjnych. Jednak dodatkowa złożoność płynąca z współbieżność niesie za sobą szereg problemów związanych z synchronizacją i prawidłowym dostępem do współdzielonych zasobów. Poniżej przedstawiono trzy często występujące problemy związane z programami współbieżnymi.

1. **Zakleszczenie** (z ang. *deadlock*) jest sytuacją, gdy grupa procesów związana jest wzajemnym oczekiwaniem na wykonanie pewnej akcji lub spełnienia pewnego warunku, które nigdy nie zostanie zrealizowane.

   By zilustrować problem zakleszczenia możemy wykorzystać przykład czterech samochodów znajdujących się na skrzyżowaniu równorzędnym. Kodeks drogowy nakazuje aby kierowca znajdujący się na takim skrzyżowaniu zawsze ustąpił pierwszeństwa pojazdowi po prawej. Oczywiście, w opisanym scenariuszu każdy kierowca ma pojazd z prawej strony, stąd żaden nie może wjechać na skrzyżowanie. Mówimy że dochodzi wtedy do zakleszczenia.

   Typowym przykładem zakleszczenia w programowaniu współbieżnym jest **wzajemne trzymanie i oczekiwanie na zasób**. Załóżmy że proces A otrzymał dostęp do gniazda G_1 oraz oczekuje na dostęp do gniazda G_2. Załóżmy również, że proces B znajduje się w sytuacji dokładnie odwrotnej - tzn. otrzymał gniazdo G_2 i oczekuje na gniazdo G_1. W opisanym przypadku oba procesy będą oczekiwały na gniazdo, które posiada ten drugi, prowadząc do zatrzymania pracy obu. Dodatkową konsekwencją zakleszczenia jest marnotrawienia cennego czasu procesora. Dyspozytor, kierując się zasadą uczciwości, wciąż będzie dopuszczał do wykonywania zakleszczone procesy. Spędzą one jednak przydzielony kwant czasu na oczekiwaniu na zasób, którego nigdy nie otrzymają.

2. **Zagłodzenie** (z ang. *starvation*) ma miejsce wtedy, gdy proces nie jest dopuszczony do wykonania z powodu niskiego priorytetu.

   Dla zilustrowania wykorzystamy ponownie przykład z ruchu drogowego. Załóżmy, że pewien pojazd próbuje skręcić na skrzyżowaniu z pierwszeństwem, znajdując się na drodze podporządkowanej. Pojazd taki jest zobowiązany do ustępowania pojazdom znajdującym się na drodze głównej. Przy odpowiednim natężeniu ruchu pojazd taki może nigdy nie zostać dopuszczony do drogi głównej.

   Analogicznie może wydarzyć się z procesami i najczęściej jest to wynikiem **nieuczciwego** (ang. *unfair*) **algorytmu szeregowania**. Algorytm ten ustala kolejność w jakiej procesy otrzymują zasoby, na przykład nadając im odpowiedni priorytet. W wypadku nieuczciwego algorytmu szeregowania może zaistnieć proces, któremu zawsze będzie przydzielany najniższy priorytet i będzie on zmuszony ustępować kolejnym procesom. Jedną z metod rozwiązania problemu zagłodzenia jest uzależnienie priorytetu procesu od czasu oczekiwania na zasób. W porównaniu do zakleszczenia, zagłodzenie jest problemem subtelniejszym i cięższym do wykrycia.

3. **Niespójność pamięci** (z ang. *memory corruption*) jest zjawiskiem, które może wystąpić podczas równoległej modyfikacji współdzielonej pamięci. Na potrzeby zobrazowania problemu załóżmy, iż dwa niezależne procesy wykonują operacje inkrementacji

na współdzielonej zmiennej całkowitoliczbowej, nie korzystając z żadnych mechanizmów synchronizacji. Inkrementacja składa się z trzech atomowych instrukcji:

- odczytu wartości zmiennej (i np. jej zapisu w rejestrze tymczasowym);
- podniesienia tej wartości o jeden;
- zapisu nowej wartości do zmiennej w pamięci.

W tabeli 4.1 zaprezentowano możliwy przebieg wykonywania dwóch instrukcji inkrementacji prowadzący do niespójności pamięci.

Proces A	Proces B	Wartość zmiennej
Odczytaj wartość	-	0
-	Odczytaj wartość	0
-	Zwiększ wartość	0
-	Zapisz wartość	**1**
Zwiększ wartość	-	**0**
Zapisz wartość	-	1

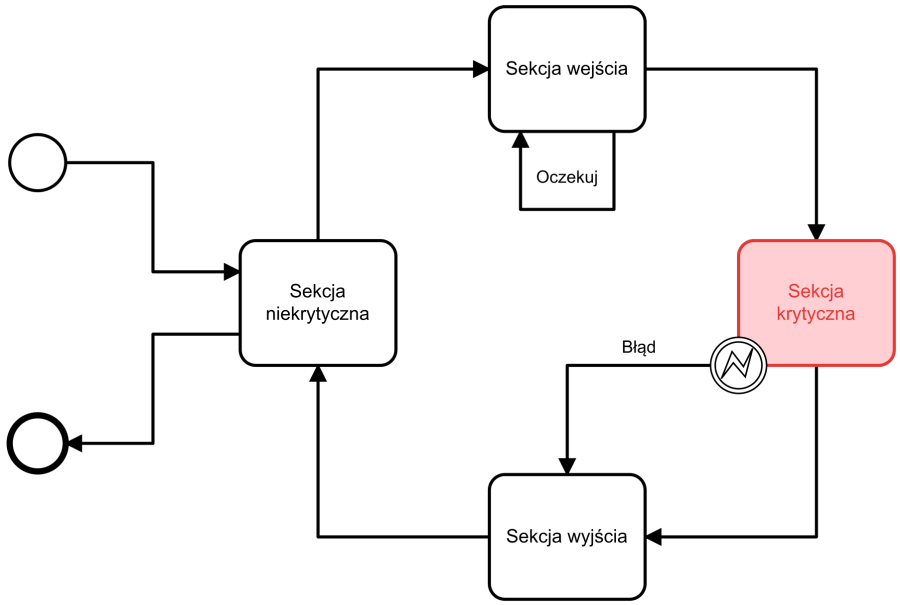
Tablica 4.1: Przebieg dwóch równoległych inkrementacji prowadzący do niespójności pamięci

Pomiędzy instrukcją zapisu procesu B a instrukcją zapisu wykonywaną przez proces A, wartość zmiennej jest niespójna. Z perspektywy procesu B wynosi ona jeden a z perspektywy procesu A zero. Rezultatem tej niespójności będzie zwiększenie wartości zmiennej wyłącznie o jeden - przy wykonaniu dwóch operacji inkrementacji. Jest to zachowanie niezamierzone i niepożądane. Problem niespójności pamięci jest niezwykle trudny do wykrycia, gdyż występuje on w sposób **niedeterministyczny**, w zależności od przebiegu, i klasyczne testy mogą go nie wykryć.

4.1.2 Problem wzajemnego wykluczenia

Niniejsza praca koncentruje się na próbach naprawy programów, które rozwiązują pewne podstawowe zagadnienie programowania współbieżnego - problem wzajemnego wykluczenia (ang. *mutual exclusion*). Realizując schemat wzajemnego wykluczenia zapewniamy że wybrany fragment kodu, nazywany **sekcją krytyczną**, zawsze wykonywany jest przez co najwyżej jeden proces na raz. Inaczej mówiąc, poprawnie zaprojektowany schemat wzajemnego wykluczenia gwarantuje że sekcja krytyczne nie będzie wykonywana współbieżnie.

Najczęściej w sekcji krytycznej umieszczane są czynności związane z dostępem do wrażliwych współdzielonych zasobów. Wzajemne wykluczenie jest niezwykle użyteczne, upraszczając rozwiązanie części problemów związanych z współbieżnością. Na przykład, rozwiązując problem niespójności pamięci wystarczy umieścić operacje na pamięci współdzielonej w sekcji krytycznej. Oczywiście, wykrycie wszystkich miejsc, gdzie takie operacje zachodzą jest wciąż wyzwaniem i pozostaje problemem samym w sobie. Ogólną strukturę procesu korzystającego ze schematu wzajemnego wykluczenia można podzielić na 4 sekcje.



Rysunek 4.1: Diagram stanów (w notacji BPMN) procesu korzystającego z schematu wzajemnego wykluczenia

1. **Sekcja niekrytyczna** jest sekcją w której proces wykonuje czynności niewymagające wzajemnego wykluczenia. Proces w tej fazie nie używa współdzielonych zasobów i nie wchodzi w interakcje z innymi procesami.

2. **Sekcja wejścia** jest sekcją w której proces oczekuje na wejście do sekcji krytycznej. W celu podjęcia decyzji o wejściu do sekcji krytycznej lub dalszym oczekiwaniu w sekcji wejścia, konieczna jest komunikacja pomiędzy procesami. Odbywa się on przy pomocy współdzielonych zasobów. Należy upewnić się by ich wykorzystanie nie było narażone na problemy z niespójnością pamięci lub zadbać by wystąpienie takich problemów nie wpłynęło na poprawność programu.

3. **Sekcja krytyczna** jest sekcją w której ma miejsce dostęp do wrażliwej części, współdzielonej pamięci lub wykonywana jest inna operacja narażona na błędy związane z współbieżnością. Zakładamy że sekcja krytyczna może być odwiedzana w danym momencie przez tylko jeden proces.

4. **Sekcja wyjścia** jest sekcją w której proces sygnalizuje, że sekcja krytyczne jest wolna i że kolejny oczekujący proces (jeśli taki istnieje) może zostać do niej dopuszczony. Podobnie jak w sekcji wejścia, należy upewnić się, że komunikacja pomiędzy procesami odbywa się w bezpieczny sposób. Po wysłaniu odpowiedniego komunikatu proces powraca do sekcji niekrytycznej. Może w niej pozostać do końca swojego działania lub ponownie przejść do sekcji wejścia.

Poprawnie zaimplementowany schemat wzajemnego wykluczenia gwarantuje dwie własności. Po pierwsze, gwarantuje nienaruszalność sekcji krytycznej - sekcja krytyczna nigdy nie będzie wykonywana współbieżnie. Po drugie, zapewnia brak zakleszczeń. Oznacza to, że jeśli dany proces zasygnalizuje chęć wejścia do sekcji krytycznej (znajdzie się w sekcji

wejścia) to prędzej czy później zostanie on do niej wpuszczony. Dodatkowo, po zakończeniu wykonywania sekcji krytycznej, proces zawsze musi przejść do sekcji wyjścia, również wtedy, gdy opuścił sekcje krytyczną w wyniku napotkania błędów lub wystąpienia awarii (zob. Rys. 4.1).

4.1.3 Analiza programów współbieżnych

Każdy proces, niezależnie czy wykonuje program sekwencyjny czy współbieżny, realizuje w pewnej ustalonej kolejności ciąg atomowych instrukcji. Wzajemne uszeregowanie w czasie instrukcji procesów uczestniczących w wykonaniu programu nazywane jest **przeplotem**.

Dla sekwencji N instrukcji wykonywanych przez T procesów liczbę możliwych przeplotów M można określić poniższym wzorem [18, s. 332].

$$M = \frac{(NT)!}{N!^T} \tag{4.1}$$

Liczba możliwych przeplotów dla wcześniej przedstawionego przykładu inkrementacji zmiennej wynosi 20. Dla sekwencji złożonej z 8 instrukcji liczba ta wynosi 12870. Łatwo zauważyć że gwałtownie rosnąca liczba przebiegów szybko sprawi że analiza programów współbieżnych stanie się niezwykle trudna. Aby stwierdzić poprawność programu współbieżnego konieczne jest jednak wykazanie, że dla każdego możliwego przeplotu wykona on się poprawnie.

Ze względu na wielkość przestrzeni przeplotów, nie jest możliwe pełne ich przetestowanie dla realistycznych przykładów. Dodatkowo, z perspektywy użytkownika systemu operacyjnego przeploty generowane są w sposób zupełnie niedeterministyczny, zależny m.in. od architektury komputera, algorytmu szeregowania wykorzystywanego przez dyspozytora, czy częstotliwości taktowania zegara. Z tego powodu najczęściej spotykaną formą testowania programów współbieżnych jest ich wielokrotne uruchomienie, przy użyciu (jeśli to możliwe) jak największej ilości równolegle działających procesów, w nadziei że natrafi się na przeplot powodujący błędne wykonanie. Jednak nawet jeśli omawiane testy wykażą istnienie błędu, to jego zidentyfikowanie i naprawa wciąż pozostaje czasochłonnym i wymagającym przedsięwzięciem. Istnieją narzędzia do formalnej analizy programów współbieżnych, takie jak Spin [11] czy TLA+ [15], jednak wymagają one dość dużej wiedzy specjalistycznej.

W niniejszej pracy sprawdzone zostanie, jak algorytmy genetyczne zaimplementowane w ARJi radzą sobie z wykrywaniem błędów w programach współbieżnych na przykładzie (uszkodzonego) historycznie wczesnego rozwiązania Petersona problemu wzajemnego wykluczenia.

4.2 Algorytm Petersona

Algorytm Petersona jest algorytmem rozwiązującym problem wzajemnego wykluczenia dla dwóch procesów, sformułowanym przez Gary'ego L. Petersona w 1981 roku [21]. Późniejsze rozwiązania (jak algorytm filtrujący) uogólniają algorytm Petersona do dowolnej liczby współdziałających procesów, jednak w tej sekcji przedstawiono próby napraw wyłącznie wersji oryginalnej. Dla poprawienia czytelności w dalszej części dwa omawiane procesy będą nazywane odpowiednio procesem lewym i procesem prawym.

Algorytm 6 Współdzielone zmienne

$lewy = False$
$prawy = False$
$priorytet = 0$

Algorytm 7 Lewy proces

$lewy = True$
$priorytet = 1$
while $prawy$ **and** $priorytet == 1$ **do**
    ▷ Czekaj
end while
▷ Początek sekcji krytycznej
▷ ...
▷ Koniec sekcji krytycznej
$lewy = False$

Algorytm 8 Prawy proces

$prawy = True$
$priorytet = 0$
while $lewy$ **and** $priorytet == 0$ **do**
    ▷ Czekaj
end while
▷ Początek sekcji krytycznej
▷ ...
▷ Koniec sekcji krytycznej
$prawy = False$

Rysunek 4.2: Algorytm Petersona

Rys. 4.2 prezentuje klasyczną wersję algorytmu Petersona. Pseudokod przedstawiony w Alg. 6 prezentuje inicjalizację zmiennych współdzielonych, zaś Alg. 7 i 8 prezentują pseudokod lewego i prawego procesu.

Opis działania Poprawne działanie algorytmu zapewnione jest dzięki odpowiedniej komunikacji pomiędzy procesami, odbywającej się w sekcji wejścia i wyjścia. W celu komunikacji procesy używają trzech zmiennych współdzielonych.

- Oba procesy mają przypisane po jednej zmiennej boolowskiej, której wartość sygnalizuje chęć wejścia do sekcji krytycznej. Zmienne te to, odpowiednio, *lewy* i *prawy*. Wyrażają one źainteresowanie"procesu wejściem do sekcji krytycznej.
- Trzecia zmienna (nazywana dalej *priorytet*) przechowuje informacje o tym, który proces w danym momencie ma pierwszeństwo. Jest to zmienna binarna, gdzie uznaje się że wartość 0 oznacza pierwszeństwo lewego procesu, a wartość 1 prawego.

Protokół komunikacji pomiędzy procesami przebiega następująco. Po zakończeniu sekcji niekrytycznej proces przechodzi do sekcji wejścia. Wpierw przypisuje skojarzonej boolowskiej zmiennej wartość *True*, dając znać że jest zainteresowany wejściem do sekcji krytycznej. Następnie ustawia zmienną *priorytet*, tak aby przekazać pierwszeństwo drugiemu procesowi. Przykładowo, lewy proces przypisuje priorytetowi wartość 1, co sprawi że pierwszeństwo otrzyma proces prawy. Po ustaleniu pierwszeństwa proces oczekuje w pętli na dopuszczenie do sekcji krytycznej. Proces oczekuje dopóki **nie otrzyma pierwszeństwa** lub **drugi proces nie wykaże braku zainteresowania** wejściem do sekcji krytycznej. Gdy proces zostanie dopuszczony, wykonuje sekcje krytyczną, a następnie wkracza do sekcji wyjścia. W sekcji wyjścia deklaruje obecny brak zainteresowanie wejściem do sekcji krytycznej poprzez ustawienie odpowiedniej zmiennej na wartość *False*, po czym przechodzi do sekcji niekrytycznej.

Założenia i własności algorytmu Aby algorytm Petersona działał poprawnie, konieczne jest spełnienie założenia o **natychmiastowości** i **atomowości** operacji wykonywanych na współdzielonych zmiennych. Warto odnotować, że nie jest konieczne zapewnienie tych właściwości dla koniunkcji wykonywanej w warunku pętli. Jeśli wymienione założenia są spełnione, algorytm Petersona gwarantuje własności opisane pod koniec sekcji 4.1.2 - tzn. nienaruszalność sekcji krytycznej oraz brak zakleszczeń. Dodatkowo zapewnia też brak zagłodzeń. Poniżej zawarto zarys dowodów dla poszczególnych własności algorytmu, opierając się na pracy [32].

- **Nienaruszalność sekcji krytycznej** Teza: lewy i prawy proces nigdy nie znajdą się równocześnie w sekcji krytycznej. Przeprowadzimy dowód przez zaprzeczenie. Załóżmy, że w pewnej chwili oba procesy znajdują się w sekcji krytycznej, co oznacza, że oba zasygnalizowały chęć wkroczenia do niej, więc $lewy = prawy = True$. Jako, że oba warunki pętli nie mogą zostać spełnione jednocześnie, ponieważ priorytet zawsze będzie dawał pierwszeństwo jednemu z procesów, któryś proces musiał wkroczyć do sekcji krytycznej pierwszy. Bez utraty ogólności załóżmy, że pierwszy wkroczył proces prawy, w chwili t_1, a następnie proces lewy w chwili t_2, gdzie $t_1 < t_2$. Warunek pętli prawego procesu może być sfalsyfikowany w chwili t_1 na dwa sposoby:

  - $lewy = False$, co oznacza, że lewy proces znajduje się w sekcji wyjścia lub przed wykonaniem sekcji niekrytycznej. Skoro jednak proces prawy nie zmieni wartości $True$ zmiennej $prawy$ aż do chwili t_2 włącznie, lewy proces nie będzie mógł wejść w czasie t_2 do sekcji krytycznej. Otrzymaliśmy więc sprzeczność.
  - $priorytet = 1$, co po przeprowadzeniu podobnego rozumowania również prowadzi do sprzeczności.

  Pełen formalny dowód tej własności algorytmu można znaleźć w pracy [21].

- **Brak zakleszczeń oraz zagłodzeń** Aby ukazać brak zakleszczeń rozważa się dwa przypadki. W pierwszym przypadku tylko jeden wątek jest zainteresowany wejściem do sekcji krytycznej. Bez utraty ogólności załóżmy że jest to lewy proces. Patrząc na warunek pętli widać że proces ten zostanie wpuszczony do sekcji krytycznej, ponieważ zmienna $prawy$ ma wartość $False$. Zatem nie dojdzie do zakleszczania. W przypadku gdy oba procesy oczekują na wkroczenie do sekcji krytycznej, wartość priorytetu musi sfalsyfikować warunek jednej z pętli sekcji wejścia. W obu opisanych przypadkach nie ma możliwości zakleszczenia.

  Rozważając zagłodzenie, należy rozpatrzyć scenariusz w którym proces który opuścił sekcje krytyczną natychmiast podejmuje próbę ponownego wejścia. Jednak przy założeniu uczciwego przydziału czasu procesora, drugi proces w końcu zasygnalizuje swoje zainteresowanie wejściem do sekcji krytycznej. W połączeniu z przekazywaniem priorytetu na rzecz drugiego procesu uniemożliwia to zagłodzenie drugiego procesu.

4.2.1 Uszkodzony algorytm Petersona

W celu sprawdzenia czy ARJA jest w stanie naprawiać programy współbieżne, przygotowano uszkodzoną wersję algorytmu Petersona. Wprowadzony błąd polega na zamienieniu kolejności wykonywania dwóch instrukcji lewego procesu – instrukcji przekazania

priorytetu oraz instrukcji sygnalizującej zainteresowanie procesu wkroczeniem do sekcji krytycznej. Kod algorytmu z uszkodzonym procesem przedstawiono na Rys. 4.3.

Warto zauważyć, że wprowadzony błąd jest rozwiązywalny przy wykorzystaniu istniejących wyrażeń – wystarczy bowiem ponownie zamienić linijki miejscami by ponownie otrzymać poprawny kod (por. Rys. 4.2). Tym samym spełnione jest **założenie nadmierności** i ARJA nie jest z góry skazana na porażkę.

Algorytm 9 Współdzielone zmienne

$lewy = False$
$prawy = False$
$priorytet = 0$

Algorytm 10 Lewy proces

$priorytet = 1$
$lewy = True$
while $prawy$ **and** $priorytet == 1$ **do**
    ▷ Czekaj
end while
▷ Początek sekcji krytycznej
▷ ...
▷ Koniec sekcji krytycznej
$lewy = False$

Algorytm 11 Prawy proces

$prawy = True$
$priorytet = 0$
while $lewy$ **and** $priorytet == 0$ **do**
    ▷ Czekaj
end while
▷ Początek sekcji krytycznej
▷ ...
▷ Koniec sekcji krytycznej
$prawy = False$

Rysunek 4.3: Uszkodzony algorytm Petersona

- **Nienaruszalność sekcji krytycznej** Wykażemy, że uszkodzony algorytm Petersona nie zachowuje własności wzajemnego wykluczenia. W tym celu wskażemy przeplot, który doprowadzi do pojawienia się w sekcji krytycznej obu procesów w tym samym czasie. Kolumny tabeli 4.2 prezentują, odpowiednio, wykonywaną linijkę lewego i prawego procesu, oraz wektor stanu programu, tzn. wartości zmiennych programu po wykonaniu danej linijki. Przyjęto że zmienne w wektorze stanu kodowane są w kolejności [$lewy$, $prawy$, $priorytet$].

  Warto zaobserwować, że w żadnym momencie nie są wykonywane dwie instrukcje równolegle, więc błąd programu wynika ze złej kolejności instrukcji, nie niespójności pamięci.

- **Brak zakleszczeń oraz zagłodzeń** Wprowadzony błąd nie doprowadzi do pojawienia się zakleszczeń lub zagłodzeń, co można wykazać przy pomocy dokładnie takiej samej analizy jak w przypadku poprawnego algorytmu.

Przeplot		
Lewy proces	Prawy proces	Wektor stanu
$priorytet = 1$	-	[False, False, 1]
-	$prawy = True$	[False, True, 1]
-	$priorytet = 0$	[False, True, 0]
-	**while** $lewy$ **and** $priorytet == 0$	[False, True, 0]
-	**sekcja krytyczna**	[False, True, 0]
$lewy = True$	**sekcja krytyczna**	[True, True, 0]
while $prawy$ **and** $priorytet == 1$	**sekcja krytyczna**	[True, True, 0]
sekcja krytyczna	**sekcja krytyczna**	[True, True, 0]
-	$prawy = False$	[True, False, 0]
$lewy = False$	-	[False, False, 0]

Tablica 4.2: Przeplot uszkodzonej wersji algorytmu Petersona prowadzący do naruszenia sekcji krytycznej

4.2.2 Przygotowanie do naprawy algorytmu Petersona

W celu przeprowadzenia automatycznej naprawy algorytmu Petersona przygotowano implementację uszkodzonej wersji algorytmu, napisaną w języku Java. Oprócz tego opracowano zbiór testów, który posłuży jako wyrocznia oraz ustalono parametry eksperymentu.

Implementacja uszkodzonego algorytmu Listingi 4.1, 4.2 i 4.3 prezentują implementację uszkodzonej wersji algorytmu Petersona.

W implementacji zakładamy atomowość operacji wykonywanych na współdzielonych zmiennych. W tym celu w klasie *SharedVariables* użyto wbudowanych klas z przedrostkiem *Atomic*, które dostarczają zestawu metod przy pomocy których można modyfikować wartość zmiennych w sposób atomowy.

Klasa *Process* (List. 4.2), z której dziedziczą lewy i prawy proces, jest klasą abstrakcyjną dostarczającą konstruktor oraz abstrakcyjną metodę *execute*.

Klasa *Counter* (List. 4.3) jest implementacją licznika. Inkrementacja jego wartości, wykonywana przy pomocy metody *increment* nie jest atomowa i tym samym jest narażona na problemy niespójności pamięci. Fakt ten będzie wykorzystywany w dalszej części w celu wykrycia naruszenia sekcji krytycznej.

```java
public class SharedVariables {
    public final AtomicInteger priority = new AtomicInteger(1);
    public final AtomicBoolean leftInterested = new AtomicBoolean(false);
    public final AtomicBoolean rightInterested = new AtomicBoolean(false);
}
```

```java
public class LeftProcess extends Process {

    public LeftProcess(SharedVariables sharedVariables, Counter counter) {
        super(sharedVariables, counter);
    }

    @Override
    public void execute() {
        sharedVariables.priority.set(2); /* <-- Faulty Line */
        sharedVariables.leftInterested.set(true);
        while (accessDenied()) { /* Wait */ }
        counter.increment();
        sharedVariables.leftInterested.set(false);
    }

    private boolean accessDenied() {
        return sharedVariables.rightInterested.get() &&
                sharedVariables.priority.get() == 2;
    }
}
```

```java
public class RightProcess extends Process {

    public RightProcess(SharedVariables sharedVariables, Counter counter) {
        super(sharedVariables, counter);
    }

    @Override
    public void execute() {
        sharedVariables.rightInterested.set(true);
        sharedVariables.priority.set(1);
        while (accessDenied()) { /* Wait */ }
        counter.increment();
        sharedVariables.rightInterested.set(false);
    }

    private boolean accessDenied() {
        return sharedVariables.leftInterested.get() &&
                sharedVariables.priority.get() == 1;
    }
}
```

Listing 4.1: Implementacja uszkodzonego algorytmu Petersona w języku Java

```
1  public abstract class Process {
2
3      protected final SharedVariables sharedVariables;
4      protected final Counter counter;
5
6      public Process(SharedVariables sharedVariables, Counter counter) {
7          this.sharedVariables = sharedVariables;
8          this.counter = counter;
9      }
10
11     public abstract void execute();
12
13 }
```

Listing 4.2: Process - klasa abstrakcyjna

```
1  public class Counter {
2
3      private int counter;
4
5      public void increment() {
6          counter++;
7      }
8
9      public int getCount() {
10         return counter;
11     }
12
13 }
```

Listing 4.3: Counter - (nieatomowy) licznik

Wyrocznia Wyrocznia składa się z **8 testów jednostkowych**. Testy zrealizowane zostały przy pomocy biblioteki *JUnit* [1], zgodnie z wymaganiami narzuconymi przez narzędzie ARJA (zob. sekcja 3.2.6).

W celu zweryfikowania poprawności testu używa się licznika *Counter*, który jest współdzielony przez oba procesy. Licznik inkrementowany jest w sekcji krytycznej podczas każdego wykonania procesu (tzn. użycia metody *execute*).

- **Testy poprawne:** Test uznaje się za poprawny, gdy liczba wykonań obu procesów równa się końcowej wartości licznika. Łatwo zauważyć, że jeśli zapewniona jest własność wzajemnego wykluczenia, to po wykonaniu n par procesów (w dowolnym przeplocie) wartość licznika powinna wynosić $2n$.

- **Testy błędne:** Jeśli po wykonaniu n par procesów wartość licznika jest różna (a ściślej, mniejsza) od $2n$, to mamy do czynienia z niespójnością pamięci (zob. Tab. 4.1). W tym przypadku oznacza to iż naruszona została własność wzajemnego wykluczenia.

[1] JUnit: https://junit.org/

Trzeba zaznaczyć, że nie każde naruszenie sekcji krytycznej musi zakończyć się niespójnością pamięci. Fakt ten, oraz brak gwarancji że zawsze wygenerowany zostanie wadliwy przeplot programu, sprawiają że zaimplementowane testy są **niedeterministyczne**. Aby zapewnić wyższą częstość wykrywania błędów część testów wykonywane jest wielokrotnie. W celu zaimplementowania tej funkcjonalności użyto biblioteki *MultithreadedTC* [2] przeznaczonej do testowania programów współbieżnych.

Parametry eksperymentu Przeprowadzono **5 niezależnych prób naprawy**, z których każda miała narzucony limit czasowy **50 minut**. Próba mogła zakończyć się wcześniej jeśli wygenerowane zostało **5 łatek** pomyślnie przechodzących wszystkie testy z wyroczni.

Parametry ARJA	
Parametr	*Wartość*
Rozmiar populacji	40
Limit generacji	50
Minimalna podejrzliwość	0.1
Limit punktów modyfikacji	40
Limit czasowy testów	120000 ms
Prawdopodobieństwo mutacji	$1/n$
Prawdopodobieństwo krzyżowania	1.0

Tablica 4.3: Parametry uruchomienia ARJA

Tabela 4.3 prezentuje parametry uruchomienia ARJi dla prób napraw, gdzie n jest liczbą wybranych punktów modyfikacji (zob. sekcja 3.2.1). Wszystkie wartości parametrów, oprócz limitu czasowego testów, są wartościami domyślnymi i zostały użyte w testach przeprowadzanych w pracy [35]. Ze względu na charakter wykonywanych testów - duża ilość powtórzeń - limit czasowy został znacząco zwiększony w porównaniu do domyślnej wartości 60000 ms. Wszystkie próby wykonywane były na maszynie z procesorem 3.2 GHz Intel(R) Core(TM) i5-4460, posiadającej 16 GB pamięci RAM. Eksperyment przeprowadzony był na maszynie wirtualnej, obsługującej dystrybucję systemu operacyjnego Linux - Ubuntu 18.04.5 LTS.

4.2.3 Próby naprawy algorytmu Petersona

Ta sekcja poświęcona jest zaprezentowaniu wyników z pięciu prób napraw uszkodzonego algorytmu Petersona. Każda próba została przeprowadzona przy użyciu parametrów oraz ograniczeń opisanych w poprzedniej sekcji. Każda wygenerowana łatka została poddana ocenie w celu ustalenia, czy jest poprawna czy przetrenowana. Przy ocenie poprawności łatki brane było pod uwagę nie tylko zachowanie własności wzajemnego wykluczenia ale również to, czy naprawiona wersja programu nie jest narażona na zakleszczenia lub zagło-

[2] MultithreadedTC: https://www.cs.umd.edu/projects/PL/multithreadedtc

dzenia. Tabela 4.4 prezentuje wyniki eksperymentu. Dla każdej próby podany został czas wykonania oraz liczba zwróconych łatek - zarówno poprawnych, jak i przetrenowanych.

Wyniki eksperymentu		
Próba	*Poprawne/Przetrenowane*	*Czas*
1	0/0	50m (timeout)
2	0/1	50m (timeout)
3	5/0	49m 28s
4	5/0	36m 43s
5	1/1	50m (timeout)

Tablica 4.4: Wyniki z 5 prób napraw algorytmu Petersona

Do każdej wygenerowanej łatki dołączany jest krótki raport, w którym zawarte są informacje o wprowadzonych modyfikacjach. Dla każdej użytej operacji podawany jest jej typ, punkt modyfikacji oraz wyrażenie przy pomocy którego punkt modyfikacji był edytowany. Dodatkowo dostarczane są też informacje o liczbie ewaluacji (wyliczeń przystosowania) które łatka przeszła oraz oszacowanym czasie potrzebnym do jej znalezienia. Listing 4.4 przedstawia przykładowy raport, zaś Tabela 4.5 prezentuje uśrednione wyniki z otrzymanych raportów.

```
1 InsertBefore LeftProcess.java 11

Faulty:
sharedVariables.priority.set(2);

Seed:
sharedVariables.leftInterested.set(true);

************************************************

Evaluations: 22
EstimatedTime: 529137
```

Listing 4.4: Przykładowy raport (identyfikator 19) z trzeciej próby

ARJA znalazła łatki, które poprawnie przechodzą wszystkie testy, w czterech z pięciu podjętych prób napraw. Jedynie dwie próby zakończyły się wygenerowaniem wystarczającej liczbę łatek by zakończyć naprawę przed osiągnięciem limitu czasowego. Znalezione łatki były stosunkowo niedługie - średnio poniżej dwóch operacji, co utwierdza zasadność korzystania z wielokryterialnej funkcji przystosowania. Średni czas generowania łatki nie napawa optymizmem: najlepszy średni czas to 15 minut i 47 sekund, w którym wygenerowana została przetrenowana łatka z drugiej próby. Należy zaznaczyć, że sprawdzono iż w każdej z przeprowadzonych prób rozwiązanie znajduje się w przestrzeni łatek - tzn. odpowiedni punkt modyfikacji zawierał w swoim zbiorze odsianych wyrażeń linijkę naprawiającą

Uśrednione wyniki raportów			
Próba	*Średnia długość*	*Średni czas*	*Średnia liczba ewaluacji*
1	-	-	-
2	1,00	15m 47s	50,00
3	1,80	36m 7s	155,40
4	1,60	21m 24s	152,20
5	1,50	36m 31s	161,50

Tablica 4.5: Uśrednione wyniki raportów z 5 prób napraw algorytmu Petersona

wprowadzony błąd. Wynika z tego, że niepowodzenie próby pierwszej jest efektem niedostatecznie efektywnego przeszukiwania przestrzeni łatek. W sumie, podczas wszystkich prób wygenerowano 21 łatek poprawnych oraz 2 przetrenowane. Poniżej zostały omówione wybrane z nich.

Poprawne łatki W trzech próbach ARJA znalazła łatki poprawne. W każdej z tych prób ARJA zdołała odkryć rozwiązanie naprawiające wadliwą wersję programu przy pomocy pojedynczej operacji. Przykładowa łatka z trzeciej próby, pokazana na listingu 4.5, polega na wstawieniu wyrażenia *sharedVariables.leftInterested.set(true);* przed linijką przekazującą priorytet prawemu procesowi. Taka modyfikacja programu naprawia wprowadzony błąd i uniemożliwia zaistnienie przeplotu z 4.2. Ponowne ustawienie zmiennej $leftInterested$ na wartość *true* w żaden sposób nie wpływa na poprawność programu, stąd rozwiązanie to zachowuje oczekiwane własności oryginalnego algorytmu Petersona.

```
1  public class LeftProcess extends Process {
2
3       ...
4
5      @Override
6      public void execute() {
7          sharedVariables.leftInterested.set(true);
8          sharedVariables.priority.set(2);
9          sharedVariables.leftInterested.set(true);
10         while (accessDenied()) { /* Wait */ }
11         counter.increment();
12         sharedVariables.leftInterested.set(false);
13     }
14
15      ...
16 }
```

Listing 4.5: Poprawna łatka (identyfikator 19) z trzeciej próby (dla czytelności zamieszczono tylko zmodyfikowaną część)

Kolejnym przykładem łatki, która naprawia program przy użyciu tylko jednej operacji, jest jedyna poprawna łatka z piątej próby. Dokonuje ona wstawienia wyrażenia *sharedVariables.priority.set(2);* przed linijką z instrukcją pętli *while*, dzięki czemu ostatnią operacją wykonywaną przez zmieniony proces jest przekazanie priorytetu, co uniemożliwia naruszenie sekcji krytycznej. Warto zauważyć, że w tej łatce priorytet będzie przekazywany dwa razy: na początku metody *execute* oraz po raz kolejny przed pętlą *while*. Nie powoduje to jednak zagłodzenia lewego procesu, bowiem w momencie dotarcia do pętli lewy proces zawsze wejdzie do sekcji krytycznej przed procesem prawym (ponieważ prawy proces musi wpierw przekazać priorytet).

Podczas **5 prób** wygenerowano w sumie **11 poprawnych łatek**. Próba 3 i 4 zakończyły się przed osiągnięciem limitu czasowego, ponieważ znaleziono 5 łatek (zgodnie z założeniami dotyczącymi warunku zatrzymania eksperymentu). Po przyjrzeniu się poszczególnym łatkom wygenerowanym w próbie 3 i 4 można łatwo dostrzec pewną prawidłowość: mianowicie, po znalezieniu pierwszej poprawnej łatki ARJA ma tendencję do generowanie jej "mutacji", co zazwyczaj polega na wykonaniu operacji, które nie mają wpływu na semantykę programu. Przykładowo, jedna z mutacji wstawa wyrażenie *sharedVariables.leftInterested.set(true);* na początku metody *accessDenied*. Ponieważ proces wykonujący tę metodę jest zawsze zainteresowany wejściem do sekcji krytycznej, modyfikacja ta nie wpływa na poprawność programu.

Jak przedstawiono w tabeli 4.5, dla prób 3 i 4 średnia długość łatki wynosi odpowiednio 1,80 oraz 1,60 operacji. Długość ta wynika z omówionej tendencji do dodawania operacji niezmieniających semantyki programu. Podobnego zachowania nie zaobserwowano w próbie piątej, wynika to jednak najprawdopodobniej z faktu, że pierwsza i jedyna poprawna łatka została wygenerowana stosunkowo późno, bo w 25 minucie. Dla porównania, pierwsza poprawna łatka w próbach 3 i 4 wygenerowana została odpowiednio po 8 i 13 minutach.

Przetrenowane łatki Dwie znalezione łatki okazały się przetrenowane. Co ciekawe, przetrenowanie obu łatek wynika z dokładnie tej samej przyczyny. Mianowicie, jak zilustrowano na listingu 4.6, w obu łatkach wykonywana jest operacja wstawienia wyrażenia *sharedVariables.priority.set(2);* na początku metody *accessDenied*.

```
1  public class LeftProcess extends Process {
2
3       ...
4
5      private boolean accessDenied() {
6          sharedVariables.priority.set(2);
7          return sharedVariables.rightInterested.get() &&
8                  sharedVariables.priority.get() == 2;
9      }
10 }
```

Listing 4.6: Przetrenowana łatka (identyfikator 41) z drugiej próby (dla czytelność zamieszono tylko zmodyfikowaną część)

Wprowadzona modyfikacja zapewni utrzymanie własności wzajemnego wykluczenia, w szczególności uniemożliwiając zaistnienie przeplotu z tabeli 4.2. Jednak dokładniejsza analiza wykaże, że omawiana łatka nie spełnia własności braku zagłodzeń. Wynika to z tego, że z każdym wywołaniem metody *accessDenied* przez lewy proces pierwszeństwo będzie przekazywane procesowi prawemu. Skutkiem tego, jeśli prawy proces nieustannie po opuszczeniu sekcji krytycznej będzie chciał do niej powrócić, to lewy proces nigdy nie zostanie dopuszczony do wykonania. W rezultacie lewy proces zostanie zagłodzony.

Ponieważ spełnienie obu własności opisanych w sekcji 4.2 jest konieczne do uznania programu za poprawny, opisane łatki zostały zaklasyfikowane jako przetrenowane.

4.2.4 Przygotowanie do naprawy sekwencyjnej wersji algorytmu Petersona

Wyniki poprzedniego eksperymentu pokazują, że ARJA jest w stanie naprawić programy współbieżne przy użyciu wyroczni złożonej z niedeterministycznych testów. Niestety, fakt że aż w dwóch próbach ARJA nie była w stanie znaleźć poprawnej łatki przed upływem limitu czasowego, nie napawa optymizmem. Średni czas konieczny do wygenerowanie poprawnej łatki również pozostawia wiele do życzenia. Powyższe powody przemówiły za przeprowadzeniem kolejnego eksperymentu w którym zaimplementowano zarówno moduł dyspozytora, pozwalającego na deterministyczne testowanie wybranych przeplotów, jak i sterowaną nim wersję algorytmu Petersona.

Implementacja uszkodzonego algorytmu Petersona: ręczne sterowanie W celu uzyskania pełnej kontroli nad kolejnością wykonania przeplotu instrukcji (a ściślej, linii kodu) procesów algorytmu Petersona przygotowano jego wersję sekwencyjną, sterowaną poprzez ręczne podanie skończonego przeplotu. Listing 4.8 prezentuje implementację lewego (uszkodzonego) procesu, zaś listing 4.7 prawego, poprawnego. Idea implementacji jest następująca.

- Do każdego z procesów dodano zmienną *processCounter*, będącą odpowiednikiem licznika programu i wskazującą, która instrukcja będzie wykonana jako następna.
- Wykonanie bieżącej instrukcji następuje poprzez wywołanie metody *step*; np. *leftProcess.step()*, gdzie *leftProcess* jest instancją *LeftProcess*.
- Po wykonaniu instrukcji zmienna *processCounter* jest zmieniana tak, by wskazywać kolejna instrukcję do wykonania.

Wyrocznia Dzięki zaprezentowanej implementacji łatwo można przygotować deterministyczne testy. W celu zasymulowania dowolnego przeplotu w programie należy wykonać w odpowiedniej kolejności metody *step* obu procesów, symulując w ten sposób działanie dyspozytora.

Wykrywania błędu w programie dokonujemy poprzez badanie wektora stanu programu, który zawiera wartości zmiennych współdzielonych oraz wartości zmiennych *processCounter* lewego i prawego procesu. Jako że badania prowadzone w tej części pracy mają charakter czysto akademicki, do wyroczni dodano *testy naprowadzające* na poprawne rozwiązanie. W szczególności, w wyroczni znajduje się test, który po jednokrotnym uruchomieniu metody *step* lewego procesu (przy braku uruchomień procesu prawego) sprawdza iż zmienna *sharedVariables.priority* ma wartość 1.

```
public class RightProcess extends Process {

    public RightProcess(SharedVariables sharedVariables) { super(sharedVariables); }

    public void step() {
        switch (processCounter) {
            case 1:
                sharedVariables.rightInterested = true;
                ++processCounter;
                break;
            case 2:
                sharedVariables.priority = 1;
                ++processCounter;
                break;
            case 3:
                if (sharedVariables.priority == 2 || !sharedVariables.leftInterested)
                    ++processCounter;
                break;
            case 4:
                ++processCounter;
                /* Critical Section */
                break;
            case 5:
                sharedVariables.rightInterested = false;
                processCounter = 1;
                break;
        }
    }
}
```

Listing 4.7: Implementacja prawego (nieuszkodzonego) procesu w ręcznie sterowanej wersji algorytmu Petersona

```
public class LeftProcess extends Process {

    public LeftProcess(SharedVariables sharedVariables) { super(sharedVariables); }

    public void step() {
        switch (processCounter) {
            case 1:
                sharedVariables.priority = 2; /* <-- Faulty Line */
                ++processCounter;
                break;
            case 2:
                sharedVariables.leftInterested = true;
                ++processCounter;
                break;
            case 3:
                if (sharedVariables.priority == 1 || !sharedVariables.rightInterested)
                    ++processCounter;
                break;
            case 4:
                ++processCounter;
                /* Critical Section */
                break;
            case 5:
                sharedVariables.leftInterested = false;
                processCounter = 1;
                break;
        }
    }
}
```

Listing 4.8: Implementacja lewego (uszkodzonego) procesu w ręcznie sterowanej wersji algorytmu Petersona

```
1  public void testOne() {
2      reset();
3
4      leftProcess.step();
5      rightProcess.step();
6      rightProcess.step();
7      rightProcess.step();
8      rightProcess.step();
9      leftProcess.step();
10     leftProcess.step();
11     leftProcess.step();
12
13     Assert.assertTrue(assertState(3, 3, 2, true, true));
14 }
```

Listing 4.9: Przykładowy deterministyczny test przeplotu programu

Przykładowy test przedstawiony jest na listingu 4.9, gdzie wykonywany jest przeplot:

1. Lewy proces: `sharedVariables.priority = 2;`
2. Prawy proces: `sharedVariables.rightInterested = true;`
3. Prawy proces: `sharedVariables.priority = 1;`
4. Prawy proces: `if (sharedVariables.priority == 2 || !sharedVariables.leftInterested) ++processCounter;`
5. Prawy proces: sekcja krytyczna.
6. Lewy proces: `sharedVariables.leftInterested = true;`
7. Lewy proces: `if (sharedVariables.priority == 1 || !sharedVariables.rightInterested) ++processCounter;`
8. Lewy proces: sekcja krytyczna.

Podobnie jak poprzednio, wyrocznia składa się z **8 testów**. Wykonanie **pięciu** z nich kończy się niepowodzeniem.

4.2.5 Próby napraw sekwencyjnej wersji algorytmu Petersona

Wersję sekwencyjną uszkodzonego programu poddano pięciu niezależnym próbom naprawy, których podsumowanie zaprezentowane jest w Tabeli 4.6. Ponieważ w wersji sekwencyjnej testy wykonywane są w sposób deterministyczny i zajmują dużo mniej czasu, limit czasowy dla wykonania próby został zmniejszony do **25 minut**. Zwiększono również wielkość populacji oraz limit liczby generacji do 100, tak aby przeszukiwanie nie kończyło się zbyt wcześnie - osiągnięciem limitu generacji. Pozostałe parametry eksperymentu pozostały bez zmian. Podobnie jak w poprzednim eksperymencie, każda łatka została poddana ocenie w celu ustalenia czy jest poprawna czy przetrenowana.

Jak zilustrowano w Tabeli 4.6, jedna próba naprawy zakończyła się niepowodzeniem, tzn. obliczenia przekroczyły zadany limit czasowy, zaś pozostałe próby wygenerowały po 5 łatek, osiągając w ten sposób warunek zatrzymania. W szczególności, podczas trzeciej

Wyniki eksperymentu		
Próba	*Poprawne/Przetrenowane*	*Czas*
1	4/1	18m 34s
2	4/1	16m 15s
3	5/0	10m 47s
4	5/0	12m 16s
5	0/0	25m (timeout)

Tablica 4.6: Wyniki z 5 prób napraw sekwencyjnej wersji algorytmu Petersona

próby ARJA wygenerowała 5 łatek w zaledwie 10 minut i 47 sekund. Tabela 4.7 prezentuje uśrednione wyniki z raportów znalezionych łatek.

Uśrednione wyniki raportów			
Próba	*Średnia długość*	*Średni czas*	*Średnia liczba ewaluacji*
1	3,40	14m 39s	4344,60
2	2,80	14m 27s	4323,40
3	2,80	8m 58s	2659,00
4	2,60	9m 54s	2889,40
5	-	-	-

Tablica 4.7: Uśrednione wyniki raportów z 5 prób napraw sekwencyjnej wersji algorytmu Petersona

Porównanie tabel 4.5 i 4.7 wskazuje iż naprawa wersji sekwencyjnej algorytmu Petersona wymagała więcej operacji niż naprawa wersji współbieżnej. Każda z wygenerowanych łatek koduje przynajmniej dwie operacje. Po dokładniejszej analizie okazało się, że ARJA działa w następujący sposób:

- Naturalnym sposobem naprawy, wymagającym tylko jednej operacji, wydaje się być wstawienie wyrażenia *sharedVariables.leftInterested = true;* przed wadliwą linijkę (Lin. 8, Listing 4.8). Rozwiązanie takie pozwala na uzyskanie własności wzajemnego wykluczenia.
- W wyroczni znajduje się jednak test naprowadzający sprawdzający wektor stanu po pojedynczym wykonaniu metody *step* lewego procesu, w którym oczekiwane jest że wartość zmiennej *priority* jest równa 1 (wartość początkowa), zaś przy omawianym rozwiązaniu wynosi ona 2.
- W związku z błędnym wynikiem poprzedniego testu, ARJA dokonuje pozbycia się wyrażenia *sharedVariables.priority = 2;*

W porównaniu do poprzedniego eksperymentu obok długości łatek wzrosła również średnia liczba ewaluacji, co wynika z faktu, że deterministyczne testy przebiegają dużo szybciej i tym samym ARJA ma czas przeprowadzić więcej obliczeń. Pomimo tego, średni czas potrzebny na wygenerowanie łatki znacząco się zmniejszył.

Próby napraw sekwencyjnej wersji algorytmu Petersona wygenerowały **18 poprawnych łatek** oraz **2 przetrenowane**, z których wybrane omawiamy w dalszej części.

Poprawne łatki Każda udana próba wygenerowała łatki poprawne, przy czym zaobserwowano, że utrzymana została tendencja do generowania "mutacji" pierwszego znalezionego rozwiązania. Ściślej, łatki generowane po zwróceniu pierwszego rozwiazania były jego zmodyfikowanymi wersjami, rozszerzonymi o operacje niezmieniające poprawności programu. Wśród wyników pojawiają się jednak i sytuacje, gdy program zwraca dwa różne i poprawne rozwiązania.

```
1  public class LeftProcess extends Process
2
3      public void step() {
4          switch (processCounter) {
5              case 1:
6              sharedVariables.leftInterested = true;
7                  ++processCounter;
8                  break;
9              case 2:
10                 sharedVariables.leftInterested = true;
11                 ++processCounter;
12             {
13                 sharedVariables.priority = 2;
14                 break;
15             }
16              ...
17         }
18     }
19 }
```

Listing 4.10: Poprawna łatka (identyfikator 2152) z trzeciej próby (dla czytelności zamieszczono tylko zmodyfikowaną część)

W każdej udanej próbie pierwsza wygenerowana łatka jest łatką najkrótszą.
Pierwsza poprawna łatka w próbie pierwszej wykonuje trzy operacje:

1. Zaczyna od wstawienia wyrażenia *sharedVariables.leftInterested = true;* przed instrukcją *switch*.
2. Następnie usuwa wadliwą linijkę.
3. Finalnie wstawia linijkę *sharedVariables.priority = 2;* na początku drugiego bloku *case* instrukcji *switch*.

W próbie 2, 3 oraz 4 pierwotnie wygenerowane łatki były praktycznie identyczne i polegały na zastąpieniu wadliwej linijki wyrażeniem *sharedVariables.leftInterested = true;* a następnie wstawieniem wyrażenia *sharedVariables.priority = 2;* w drugim bloku *case*

instrukcji *switch*. Ściślej, w drugiej i trzeciej próbie, instrukcja ta została wstawiona przed wyrażenie *break;*, a w próbie czwartej przed wyrażenie *++processCounter;* Listing 4.10 prezentuje zmodyfikowany fragment kodu pierwszej łatki z trzeciej próby.

Przetrenowane łatki W trakcie prób napraw wygenerowano dwie łatki, które uznano za przetrenowane. Tak samo jak w eksperymencie z współbieżną wersją algorytmu Petersona, obie łatki są przetrenowane z podobnych powodów, poniżej więc omówiono przypadek tylko jednej z nich.

```
1  public class LeftProcess extends Process {
2
3      public void step() {
4          switch (processCounter) {
5              case 1:
6                  sharedVariables.leftInterested = true;
7                  ++processCounter;
8                  break;
9              case 2:
10                 sharedVariables.leftInterested = true;
11                 sharedVariables.priority = 2;
12             {
13                 if (sharedVariables.priority == 2 || !sharedVariables.
                       leftInterested)
14                     ++processCounter;
15                 break;
16             }
17              ...
18             case 4:
19                 ++processCounter;
20             {
21                 sharedVariables.rightInterested = true;
22                 break;
23             }
24              ...
25         }
26     }
27 }
```

Listing 4.11: Przetrenowana łatka (identyfikator 3847) z drugiej próby (dla czytelność zamieszczono tylko zmodyfikowaną część)

Powyższa łatka jest jedną z "mutacji" pierwotnej łatki z próby drugiej (identycznej z łatką z listingu 4.10). Łatka wykonuje cztery operacje:

1. Zaczyna od zastąpienia wadliwej linijki wyrażeniem *sharedVariables.leftInterested = true;*.
2. Następnie zastępuje wyrażenie *++processCounter;* z drugiego bloku *case* linijką *sharedVariables.priority = 2;*.
3. Dalej wstawia cały blok instrukcji warunkowej z prawego procesu - tzn. wyrażenie *if (sharedVariables.priority == 2 || !sharedVariables.leftInterested)*

++processCounter; przed instrukcje *break;* w drugim bloku *case*.

4. Na końcu wstawia wyrażenie *sharedVariables.rightInterested = true;* przed instrukcje *break;* w czwartym bloku *case*.

W teorii zaprezentowana łatka jest poprawna, jednak jedynie dla przyjętego w eksperymencie sposobu symulacji dyspozytora. Wykorzystana implementacja umożliwia jedynie testowanie przeplotów wykonań bloków *case*, jednak nie dowolnych przeplotów wykonań linijek. Jeśli w obrębie jednego bloku *case* umieszczono więcej niż jedną (istotną) linijkę kodu, to nie ma sposobu na przetestowanie wszystkich możliwych przeplotów zawierających te linijki. Przykładowo, dla omawianej łatki nie da się napisać testu sprawdzającego poniższy przeplot.

1. Lewy proces: `sharedVariables.leftInterested = true;`
2. Lewy proces: `sharedVariables.leftInterested = true;`
3. Lewy proces: `sharedVariables.priority == 2`
4. Prawy proces: `sharedVariables.rightInterested = true;`
5. Prawy proces: `sharedVariables.priority = 1;`
6. Prawy proces: `if (sharedVariables.priority == 2 || !sharedVariables.leftInterested) ++processCounter;`
7. Lewy proces: `if (sharedVariables.priority == 2 || !sharedVariables.leftInterested) ++processCounter;`

Wynika to z faktu że instrukcja 2, 3, i 7 znajdują się w jednym bloku *case*. Tym samym z perspektywy używanego dyspozytora ich wykonanie jest atomowe.

Zaprezentowany przeplot prowadzi do **zakleszczenia** procesów. W momencie wykonania instrukcji 6 i 7, oba procesy są zainteresowane wejściem do sekcji krytycznej, a wartość zmiennej *priority* wynosi 1. Doprowadzi to do sfalsyfikowania obu warunków pętli i uniemożliwi procesom przejście do kolejnych bloków *case*. Ponieważ poprawność omawianej łatki wynika jedynie z ograniczeń przyjętego sposobu symulacji dyspozytora, uznano ją za przetrenowaną.

Rozdział 5

Konkluzje

W tym rozdziale przedstawiamy wnioski i przemyślenia dotyczące wyników eksperymentu. Wskazujemy trudności z jakimi zetknięto się w trakcie jego przeprowadzania oraz dalsze kierunki badań związanych z ewolucyjną naprawą programów współbieżnych.

5.1 Napotkane trudności

W trakcie pisania pracy znaleziono dwa błędy w kodzie narzędzia ARJA. Oba błędy wpływają na wiarygodność wyników naprawy współbieżnej wersji algorytmu Petersona. W rezultacie tego część eksperymentu musiała zostać powtórzona po ich skorygowaniu. Znalezione usterki zostały zgłoszone oraz zweryfikowane przez twórców narzędzia[1],[2].

Przyczyną obu błędów jest źle skonstruowany warunek logiczny w dwóch fragmentach kodu uruchamiającego zewnętrzne procesy. W trakcie swojego działania ARJA deleguje część zadań do innych "podprocesów", które odpowiedzialne są między innymi za filtrację testów oraz generowanie i odsiew wyrażeń. Wspomniany wadliwy warunek logiczny sprawia iż ścieżki do zależności bibliotek z których korzysta naprawiany program, nie są przekazywane do uruchamianych procesów. W rezultacie czego procesy te kończą się błędem. Dalej krótko opisujemy wpływ usterek odkrytych w programie ARJA na wyniki badań.

1. **Wadliwa generacja wyrażeń.** Pierwszy błąd znajduje się we fragmencie kodu uruchamiającego proces odpowiedzialny za generowanie i odsiew wyrażeń (zob. str. **??**). Bez odpowiednich zależności proces kończy się błędem i nie zwraca żadnych wyrażeń. Jedną z trudności w wykryciu omawianej usterki było to, że jego efekty nie były widoczne dla każdego naprawianego programu. Przykładowo, dla sekwencyjnej wersji algorytmu Petersona wyrażenia generowane były poprawnie, co wynika z tego, że ta wersja korzysta tylko z zależności biblioteki testowej *JUnit*. ARJA zakłada, że wszystkie testy zaimplementowane są przy pomocy tej właśnie biblioteki, stąd w jej ścieżce klasowej znajdują się już kopie wymaganych zależności. Tym samym, mimo że zależności podane przez użytkownika nie były przekazywane, to proces mógł wykorzystać ich posiadane kopie aby odnaleźć definicje odpowiednich klas. Inaczej sprawa wygląda przy naprawie wersji współbieżnej, która oprócz biblioteki *JUnit* wykorzystuje również bibliotekę *MultithreadedTC*. W trakcie prób naprawy tej wersji

[1]Wadliwa generacja wyrażeń: `https://github.com/yyxhdy/arja/issues/12`
[2]Wadliwa filtracja testów: `https://github.com/yyxhdy/arja/issues/13`

efekty błędu były widoczne: bez poprawnej generacji wyrażeń żadna łatka nie może być znaleziona i tym samym każda próba naprawy kończy się osiągnięciem limitu czasowego.

2. **Wadliwa filtracja testów.** Drugi znaleziony błąd związany jest z wadliwą filtracją testów (procedura filtracji testów opisana została w sekcji 3.2.5). Problem z przekazywaniem zależności sprawia, że proces filtrujący testy kończy się błędem i wszystkie poprawne testy, nawet te pokrywające podejrzane linijki, są odfiltrowywane. Ten błąd również nie jest widoczny w przypadku programów korzystających wyłącznie z biblioteki *JUnit*. Konsekwencje opisywanej usterki nie były w naszych eksperymentach tak daleko idące jak konsekwencje poprzedniego. ARJA była w stanie generować łatki, jednak w wyniku odfiltrowania części testów dużo łatwiej było o wygenerowanie łatek przetrenowanych. W szczególności, przy naprawie współbieżnej wersji algorytmu Petersona odfiltrowane zostały dwa testy sprawdzające, czy pojedyncze uruchomienie prawego oraz lewego procesu zwracają odpowiednią wartość licznika. Bez tych testów łatka najczęściej generowana w eksperymentach polega na dodaniu kolejnego użycia metody *increment* w sekcji krytycznej jednego z procesów oraz usunięcie jej wywołania u drugiego. W rezultacie wartość licznika modyfikowana jest tylko przez jeden proces, który za każdym razem inkrementuje ją dwa razy. Program naprawiony przy pomocy takiej łatki przechodzi poprawnie wszystkie nieodfiltrowane testy.

5.2 Wnioski i obserwacje

Przeprowadzony eksperyment wykazał, że automatyczna naprawa programów współbieżnych jest możliwa. ARJA była w stanie wygenerować liczne poprawne łatki dla uszkodzonego algorytmu Petersona - zarówno dla wersji współbieżnej jak i sekwencyjnej. Poniżej zaprezentowano wybrane przemyślenia, obserwacje i wnioski dotyczące przebiegu oraz wyników obu eksperymentów.

Przetrenowane łatki Jedną z ciekawszych poczynionych obserwacji jest mała liczba przetrenowanych łatek, jaka towarzyszy naprawie algorytmu Petersona. Dla wersji sekwencyjnej wynika to najprawdopodobniej z dość precyzyjnych testów naprowadzających, sprawdzających wektor stanu programu. Oczywiście, jest to sytuacja czysto akademicka - znano a priori zarówno lokalizację błędów i "podpowiedziano" ARJi metodę ich naprawy, stąd wyniki otrzymane dla tej wersji nie powinny prowadzić do dalekosiężnych wniosków. Jednak testy skonstruowane na potrzeby wersji współbieżnej dużo bardziej przypominały typowe testy, stosowane komercyjnie dla programów wielowątkowych. Powód dla jakiego również w tej wersji zaobserwowano mało przetrenowanych łatek wydaje się trudniejszy do stwierdzenia. Można jednak postawić hipotezę, że ponieważ w trakcie testów dla wersji współbieżnej każdy test powtarzany jest wielokrotnie, to za każdym razem testuje się wiele różnych losowo generowanych przeplotów programu. Konieczność poprawnego wykonania testów dla dużej liczby przeplotów może chronić przed nadmierną specjalizacją, tzn. przetrenowaniem. Aby ustalić, czy rzeczywiście tak jest konieczne są jednak dalsze badania. To że dla wersji współbieżnej wygenerowano niewiele łatek przetrenowanych daje nadzieję, że choć próby napraw programów współbieżnych są bardzo czasochłonne, to jednak poświęcony czas może przekładać się na jakość generowanych wyników.

Ocena poprawności Istotnym wyzwaniem w trakcie przeprowadzania eksperymentu była ocena poprawności programu. W zamyśle, narzędzia do automatycznej naprawy oprogramowania przeznaczone są do korekcji prostych usterek, np. wynikających z nieuwagi programisty. Mówimy tutaj o błędach takich, jak wstawienie niewłaściwego operatora arytmetycznego czy błędnym wyrażeniu logicznym w warunkach pętli lub instrukcji warunkowej. W przypadku programów współbieżnych sprawa wygląda jednak inaczej: analiza ich poprawności jest niezwykle wymagającym zadaniem, co starano się pokazać w sekcji 4.1.3. Każda najmniejsza zmiana kodu programu współbieżnego może mieć daleko idące konsekwencje. W sekcji 4.1.3 wspomniano o narzędziach do formalnej analizy poprawności programów współbieżnych, takich jak np. TLA+ lub Spin, które mogłyby pomóc przy ocenie łatek. Jednak translacja pomiędzy kodem programu i kodem oczekiwanym przez w/w narzędzia nie jest prostym zadaniem, w obecnej chwili TLA+ jest wykorzystywany więc głównie jako metoda wspomagania modelowanie systemów.

Narzędzie ARJA Na końcu należy ponownie przyjrzeć się samej ARJi. Jedną z istotniejszych decyzji jakie podjęli twórcy narzędzie było użycie wielokryterialnej funkcji przystosowania. Wyniki przeprowadzonych eksperymentów wskazują, że była to dobra decyzja: położenie nacisku zarówno na długość jak i prawność łatki sprawiają, że nie są one zaśmiecone niepotrzebnymi operacjami. Dla praktycznie każdej próby naprawy, łatka która została znaleziona jako pierwsza okazywała się najkrótszą. Kolejną istotną zaletą ARJi jest obecność mechanizmu odsiewu wyrażeń oraz metod ograniczających przestrzeń łatek (14 reguł). Dzięki nim w trakcie prób napraw zaobserwowano stosunkowo mało łatek prowadzących do błędów kompilacji. Niestety, na stan obecny ARJA nie wydaje się być narzędziem nadającym się do użytku komercyjnego, przynajmniej jeśli chodzi o naprawę programów współbieżnych. Proces generowania łatek jest czasochłonny i nie zawsze skuteczny, przy czym szczególnie dużo pracy musi zostać włożone w projektowanie testów wyroczni oraz późniejszą analizę poprawności łatek. Niemniej, ARJA jest ciekawym projektem badawczym i niezaprzeczalnie krokiem naprzód w porównaniu do swoich poprzedników.

5.3 Dalsze badania

W tej sekcji omawiamy dalsze możliwe kierunki badań związanych z ewolucyjną naprawą programów współbieżnych. Przedstawione propozycje odnoszą się do rozpoczętego eksperymentu dotyczącego **prób naprawy uszkodzonego algorytmu piekarnianego Lamporta [14]**.

Parę słów o algorytmie piekarnianym Algorytm piekarniany [14], stworzony przez Lesliego Lamporta w 1974 roku, jest rozwiązaniem problemu wzajemnego wykluczenia dla dowolnej liczby procesów. Algorytm nazywa się piekarnianym, ponieważ w działaniu przypomina proces składania zamówień w dawnych piekarniach:

1. W momencie wejścia do piekarni klient otrzymuje bilet z numerem. Otrzymany numer zawsze jest o jeden większy od numeru poprzednio wydanego biletu.

2. Po otrzymaniu biletu klient oczekuje, aż zostanie wywołany przez pracownika piekarni. Pracownik wywołuje zawsze klienta z najmniejszym, spośród nieobsłużonych, numerem.

3. Po wywołaniu, klient podchodzi do lady i składa zamówienie.

Analogicznie, w algorytmie piekarnianym Lamporta, każdemu procesowi wkraczającemu do sekcji wejścia przypisywana jest wartość liczbowa symbolizująca jego numer. Następnie proces oczekuje, aż jego numer będzie najmniejszy spośród zainteresowanych procesów, po czym wkracza do sekcji krytycznej. Oczywiście, pełne działanie algorytmu jest bardziej złożone, między innymi algorytm obsługuje scenariusz gdy, w wyniku niespójności pamięci, kiku procesom przydzielony jest ten sam numer.

Ocena wygenerowanych łatek Podobnie, jak w poprzednich eksperymentach, do implementacji algorytmu piekarnianego wprowadzono błąd umożliwiający naruszenie własności wzajemnego wykluczenia. Testy stworzone na potrzeby naprawy tego algorytmu przypominają w działaniu te których używano dla współbieżnej wersji algorytmu Petersona, użyto również takich samych parametrów jak opisano w sekcji 4.2.2. Każda z pięciu prób naprawy zakończyła się przed osiągnięciem limitu czasowego. Dalsze kontynuacja badań wymaga oceny poprawności wygenerowanych łatek. Algorytm piekarniany jest jednak bardziej złożony niż rozwiązanie Petersona, więc poprawna ocena 25 łatek jest bardzo trudnym zadaniem i wykracza poza zakres tej pracy. W szczególności, analiza łatek najprawdopodobniej wymaga użycia jednego z wspomnianych narzędzi do analizy formalnej. Odpowiednia ocena otrzymanych wyników może być istotnym krokiem w kierunku dalszej weryfikacji hipotez postawionych w sekcji 5.2. W szczególności, może ona wskazać, czy mała liczba przetrenowanych łatek jest cechą towarzyszącą próbom napraw programów współbieżnych przy pomocy ARJi.

Ulepszenie testów Kolejnym potencjalnym kierunkiem badań może być ulepszenie zaimplementowanych testów. Po powierzchownej analizie, dla wybranych łatek (wyglądających na przetrenowane) ponownie uruchomiono testy, tym razem zwiększając znacząco liczbę ich powtórzeń. Istotna część testów zakończyła się niepowodzeniem, potwierdzając przetrenowanie łatek i sugerując że obecne testy są niewystarczające. Wydaje się, że przy rozważanej wyroczni ARJA ma tendencje do generowania łatek, które jedynie zmniejszają prawdopodobieństwo wystąpienia wadliwego przeplotu, np. kilkakrotnie powielając pętlę w której proces oczekuje na wejście do sekcji krytycznej. Ulepszone testy powinny przeciwdziałać opisanej sytuacji, uniemożliwiając nadmierną generacje przetrenowanych łatek.

Bibliografia

[1] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, 2006.

[2] J. Arabas. *Wykłady z algorytmów ewolucyjnych*. PWN, 2016.

[3] K. Arulkumaran, A. Cully, and J. Togelius. Alphastar: An evolutionary computation perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 314–315, New York, NY, USA, 2019. Association for Computing Machinery.

[4] M. Ben-Ari. *Podstawy Programowania Współbieżnego i Rozproszonego*. WNT, 2016.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[7] P. Diaz-Gomez and D. Hougen. Initial population for genetic algorithms: A metric approach. pages 43–49, 01 2007.

[8] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. Empirical review of java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. pages 302–313. ACM, 2019.

[9] M. Engel. Programowanie współbieżne i rozproszone. `http://smurf.mimuw.edu.pl/book/export/html/1250`, 2011. Accessed: 2021–01-11.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[11] G. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[12] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[13] J. R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford, CA, USA, 1990.

[14] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[15] L. Lamport. Specifying concurrent systems with tla+. *Calculational System Design*, pages 183–247, April 1999.

[16] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[17] G. Le Lann. An analysis of the ariane 5 flight 501 failure - a system engineering perspective. In *Proceedings of the 1997 International Conference on Engineering of Computer-Based Systems*, ECBS'97, page 339–346, USA, 1997. IEEE Computer Society.

[18] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008.

[19] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 492–495, New York, NY, USA, 2014. Association for Computing Machinery.

[20] M. Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.

[21] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115 – 116, 1981.

[22] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 254–265, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[24] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery.

[25] H.-P. Schwefel. *Numerical Optimization of Computer Models*. Wiley, Chichester, 1981.

[26] Y. Tian, H. Wang, X. Zhang, and Y. Jin. Effectiveness and efficiency of non-dominated sorting for evolutionary multi- and many-objective optimization. *Complex Intell. Syst.*, 3, 2017.

[27] W. Vent. Rechenberg, ingo, evolutionsstrategie — optimierung technischer systeme nach prinzipien der biologischen evolution. 170 s. mit 36 abb. frommann-holzboog-verlag. stuttgart 1973. broschiert. *Feddes Repertorium*, 86(5):337–337, 1975.

[28] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[29] Wikipedia. American fuzzy lop (fuzzer) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=American%20fuzzy%20lop%20(fuzzer)&oldid=966550193`, 2020. [Online; accessed 09-July-2020].

[30] Wikipedia. Drzewo składniowe — Wikipedia, the free encyclopedia. `http://pl.wikipedia.org/w/index.php?title=Drzewo%20sk%C5%82adniowe&oldid=57128011`, 2020. [Online; accessed 09-May-2020].

[31] Wikipedia. Semafor (informatyka) — Wikipedia, the free encyclopedia. `http://pl.wikipedia.org/w/index.php?title=Semafor%20(informatyka)&oldid=59479903`, 2021. [Online; accessed 11-January-2021].

[32] G. Wolffe. Operating system concepts. `https://cis.gvsu.edu/~wolffe/courses/cs452/docs/CS_AlgProof.html`. Accessed: 2021–01-11.

[33] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55, Jan. 2017.

[34] H. Ye, M. Martinez, T. Durieux, and M. Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. In *Proceedings of SANER Workshops*, 2019.

[35] Y. Yuan and W. Banzhaf. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.