PROJEKT INDYWIDUALNY

ALGORYTM GENETYCZNY DO GRY MASTERMIND

INFORMATYKA STOSOWANA
WYDZIAŁ ELEKTRYCZNY

HUBERT MAZUR
NUMER INDEKSU: 307487

Spis treści

1 Wstęp

1.1 Czym jest algorytm genetyczny?

Algorytmy genetyczne są narzędziem, które pozwala na efektywne i relatywnie łatwe przeszukiwanie pewnej przestrzeni potencjalnych rozwiązań w celu znalezienia tego najbardziej optymalnego. Założenia działania tego typu algorytmu są proste i przypominają ewolucję: potencjalne rozwiązania są reprezentowane przez osobniki, które krzyżują się ze sobą i mutują tworząc kolejne generacje, a o tym, które osobniki będą miały szansę się skrzyżować z innymi by przekazać swoje cechy potomstwu decyduje pewien parametr. Ten parametr odzwierciedla przystosowanie osobnika do środowiska, im wyższy, tym lepszy osobnik jest z punktu ewolucji i ma większe szanse na przetrwanie lub krzyżowanie z innymi osobnikami. Przy odpowiednio dużych populacjach, dobrze dobranej funkcji określającej parametr przystosowania (dalej nazywany ***fitness***) oraz odpowiednio zdefiniowanych procesach krzyżowania i mutacji, możliwe jest bardzo skuteczne eksplorowanie rozległych przestrzeni rozwiązań i odnajdywanie najlepszych z nich. Im lepsze pokrycie tej przestrzeni rozwiązań posiadamy, tym większa szansa na powodzenie w odnalezieniu globalnie najlepszego rozwiązania. Dużą zaletą tego typu algorytmów jest kontrolowalny czynnik losowy oraz szybkość w znajdywaniu rozwiązań, które są wystarczająco dobre, niekoniecznie najlepsze.

1.2 Gra Mastermind – zasady

Gra Mastermind to gra dwuosobowa. Jeden z graczy wciela się w rolę zgadującego, a drugi wymyśla sekretny kod do odgadnięcia dla pierwszego gracza oraz ocenia jego próby zgadnięcia tego kodu. Kod zdefiniowany jest jako ciąg N kolorów, gdzie do wyboru jest C kolorów. W tej pracy skupię się na podstawowym wariancie tej gry, czyli kod będzie miał długość cztery (N=4) oraz do wyboru będzie sześć kolorów (C=6). Kolory mogą się powtarzać w kodzie.

Tura składa się z dwóch kroków: gracz zgadujący podaje sekwencję kolorów, następnie drugi gracz znający sekretny kod ocenia próbę pierwszego gracza. Na ocenę składają się białe oraz czarne pineski. Za każdą pozycję, na której znalazł się kolor odpowiadający kolorowi w haśle na tej samej pozycji gracz zgadujący dostaje jedną czarną pineskę. Za każdy kolor, który znajduje się w haśle, ale nie został umieszczony na prawidłowej pozycji w sekwencji gracz zgadujący dostaje jedną białą pineskę. Kolory z hasła, które zostały wykorzystane przy ocenie warunku na przyznanie czarne pineski nie mogą być wykorzystane przy ocenie warunku na przyznanie białe pineski. Żadna z pinesek nie jest przyporządkowana do konkretnej pozycji, ale stanowią ocenę próby odgadnięcia hasła jako całość.

Cel gry to zgadnięcie przez gracza zgadującego sekretnego kodu w jak najmniejszej liczbie prób.

1.3 Motywacja do wykorzystania algorytmu genetycznego

Algorytm genetyczny jest doskonałym kandydatem na algorytm, który pomoże rozwiązać problem prezentowany przez grę Mastermind. Elementy losowości i dobre kryteria oceny osobników są jednak kluczowe dla jego sukcesu. W 1977 roku Donald Knuth zaproponował algorytm deterministyczny do tej gry, który gwarantuje zgadnięcie hasła w 5 lub mniej prób w rozpatrywanym przez nas wariancie. Optymistycznie byłoby gdyby algorytm genetyczny był w stanie osiągnąć średni wynik porównywalnie dobry lub lepszy od 5 prób zgadnięcia hasła na grę. Każdy osobnik musi posiadać informację genetyczną (tak zwany ***genotyp***), który koduje potencjalne rozwiązanie, czyli ciąg czterech kolorów. Z poziomu kodu nie jest to trudne: każdemu kolorowi przyporządkowana zostaje liczba z zakresu od 0 do 5 włącznie. Cztery takie liczby trzymane są jako lista elementów typu liczba całkowita.

Wyznaczanie funkcji fitness było jednym z największych wyzwań w tym projekcie. Zdecydowałem się na wykorzystanie metryki odległości między potencjalnymi rozwiązaniami, która porównuje dotychczasowe próby zgadnięcia hasła z osobnikami populacji. O każdej próbie zgadnięcia hasła wiemy

dwie rzeczy: sekwencja kolorów i otrzymana ocena przez gracza znającego hasło. Odległość między takimi dwoma osobnikami definiuję wyznaczając minimalną liczbę kolorów, które należałoby zmienić, aby porównując osobnika do wcześniejszej próby otrzymać taką samą ocenę, jak ta próba otrzymała porównując się z hasłem. Liczba tych zamian kolorów na minusie jest tą odległością. Maksymalna wartość tej odległości to 0, a minimalna to -4. Obliczając sumę odległości dla wybranego osobnika z populacji ze wszystkimi do tej pory próbami odgadnięcia hasła otrzymuję jego fitness. Zastosowanie takiej metryki pozwala na szacowanie oceny rozwiązania bez potrzeby wykonywania ruchu oraz wybór najlepszych osobników do krzyżowania.

2 Cel projektu

Głównym celem projektu było poznanie możliwości i zasad działania algorytmów genetycznych przy próbie rozwiązania problemu gry Mastermind w najbardziej optymalny sposób. Drugorzędnym celem w trakcie pisania kodu stało się lepsze planowanie pod kątem podejścia obiektowego i wykorzystanie popularnych praktyk wykorzystywanych właśnie w tej metodyce. Celem nie było utworzenie interfejsu graficznego, ponieważ temat jest na tyle obszerny, że tworzenie interaktywnej aplikacji wykraczałoby poza założony poziom złożoności projektu. Po zaimplementowaniu podstawowych cech algorytmu genetycznego, kolejnym celem stało się dobranie odpowiednich parametrów symulacji w celach optymalizacyjnych, takich jak początkowy rozmiar populacji, szansa na mutację, selekcja osobników do krzyżowania czy metody krzyżowania. Ostatnim celem była analiza skuteczności projektu i wyciągnięcie wniosków na podstawie jego działania.

3 Wykorzystane narzędzia i technologie

Do implementacji algorytmu wybrałem język C++ ze względu na jego szybkość, możliwości pisania obiektowego kodu oraz jego całkiem dobrą znajomość. Do pracy używałem programu Visual Studio Code, który jest intuicyjnym edytorem tekstu z otwartym kodem źródłowym. Pracując na systemie Windows 10 wykorzystałem wirtualizację WSL w wersji drugiej (dystrybucja Ubuntu), aby móc kompilować i testować kod na maszynie z jądrem Linuxowym i pracować w terminalu oraz używać takich narzędzi jak Makefile. WSL jest doskonałym narzędziem dla deweloperów, którzy cenią sobie swobodę pracy w przyjaznym interfejsie graficznym dostarczanym przez system Windows, jednocześnie dostarczając wszelkie niezbędne narzędzia w terminalu utożsamiane z Linuxem. VSCode posiada doskonałą integrację ze środowiskiem WSL dzięki pakietom rozszerzeń dostarczanym przez Microsoft.

4 Implementacja

4.1 Analiza wymagań

- Program ma mieć możliwość symulowania przebiegu gry Mastermind przy użyciu algorytmu genetycznego.
- Hasło powinno być losowe dla każdej instancji gry.
- Program nie musi posiadać interfejsu graficznego, komunikacja może odbywać się za pośrednictwem terminala.
- Program ma mieć możliwość symulowania wielu gier Mastermind i obliczać statystyki działania algorytmu genetycznego na ich podstawie. Do statystyk zaliczać się mają: średnia liczba prób odgadnięcia hasła, liczba nieudanych prób (przekroczenie limitu prób), procent gier, które przekroczyły limit prób. Liczba symulacji do wykonania ma być argumentem programu.
- Wszystkie parametry symulacji powinny być łatwe w edycji.

4.2 Opis klas

4.2.1 Mastermind

Jest to główna klasa projektu. Posiada ona instancje klasy **GeneticAlgorithm** oraz **CodeKeeper.** Jej najważniejszymi metodami są *startGameLoop* oraz *getStatistics*.

Metody:

- *void startGameLoop()* - jest odpowiedzialna za inicjalizację głównej pętli gry, która trwa dopóki hasło nie zostanie odgadnięte lub zostanie osiągnięty limit prób. Przed wejściem do pętli wykonywana jest pierwsza próba odgadnięcia hasła, która jest odgórnie ustalona na ciąg 0011. Ma to na celu inicjalizację historii prób zgadnięcia hasła. W każdej iteracji pętli, algorytm genetyczny tworzy kolejną populację, a następnie dokonuje próby odgadnięcia hasła.
- *Statistics getStatistics() const* - zwraca statystyki z odbytej gry w postaci obiektu klasy **Statistics**.
- *bool isGameFinished() const* - sprawda warunki zakończenia gry i zwraca prawdę, jeżeli te warunki zostały spełnione (hasło odgadnięte lub osiągnięcie limitu prób).
- *Score guess(const Code& guess)* - przyjmuje za argument referencję do kodu reprezentującego próbę odgadnięcia hasła i przedstawia go **CodeKeeper**'owi w celu jego oceny.

4.2.2 GeneticAlgorithm

Zadaniem tej klasy jest implementacja funkcjonalności algorytmu genetycznego i udostępnianie ich klasie **Mastermind**. Posiada historię prób zgadnięcia hasła, obecną populację oraz licznik populacji.

Metody:

- *void nextStep()* - wykonuje kolejny krok algorytmu genetycznego. Oblicza fitness wszystkich osobników, wyznacza kolejną generację osobników i inkrementuje licznik generacji.
- *Code& getNextGuess()* - zwraca referencję do obiektu klasy **Code**, który reprezentuje kolejną próbę odgadnięcia hasła, umożliwiając jego ocenę przez **CodeKeeper**'a.
- *bool generationLimitReached() const* - zwraca prawdę, gdy numer generacji dojdzie do limitu prób odgadnięcia hasła.
- *int getGenerationNo() const* - zwraca wartość pola generationNo reprezentującego numer obecnej generacji.

4.2.3 CodeKeeper

Ta klasa przechowuje ukryty kod, który należy odgadnąć oraz jest odpowiedzialna za jego losową inicjalizację.

Metody:

- *void checkTheGuess(Code& guess)* - przyjmuje za argument referencję do kodu, który jest porównywany do sekretnego hasła i oceniany zgodnie z zasadami gry.
- *bool wasCodeBroken() const* - zwraca informację, czy kod został złamany.

4.2.4 Population

Klasa ta przechowuje wszystkie osobniki z obecnej generacji oraz **CrossSelector**. Odpowiedzialna jest za operacje na zbiorze osobników i tworzeniu kolejnych generacji.

- *void nextGeneration()* - wyznacza osobniki do krzyżowania, generuje dzieci tych osobników, mutuje je i/lub ich rodziców, tworząc w ten sposób nowe pokolenie osobników.
- *void calculateFitness(const std::vector<Code>& prevGuesses)* - przyjmuje za argument historię prób odgadnięcia hasła i na ich podstawie każdemu osobnikowi wyznacza parametr fitness.
- *Specimen& getBestSpecimen()* - zwraca osobnika z największym fitness.
- *std::vector<Specimen> makeOffsprings(std::vector<Specimen>& specimensToCross, int offspringCount)* - przyjmuje zbiór osobników, które będą się ze sobą krzyżować, oraz liczbę dzieci jakie mają powstać. Następnie lista tych dzieci jest zwracana.
- *void applyMutation(std::vector<Specimen>& specimensToMutate)* - przyjmuje zbiór osobników, a następnie poddaje każdego mutacji z pewnym prawdopodobieństwem zdefiniowanym w parametrach.

4.2.5 CrossSelector

Funkcją tej klasy jest jednorazowe stworzenie maski, która definiuje, które osobniki z posortowanej listy są wybierane do krzyżowania i przejdą do kolejnej generacji. Parametr percentOfPopulationToCross wyznacza procent osobników, które ta maska ma wybierać. Maska konstruowana jest w taki sposób, aby osobniki z dużym współczynnikiem fitness miały większą szansę na znalezienie się na liście osobników do krzyżowania, ale im mniejsza jest wartość tego współczynnika tym mniejsza jest szansa.

Metody:

- *std::vector<Specimen> applyMaskToPopulation(std::vector<Specimen>& specimens) const* - przyjmuje posortowaną listę osobników, nakłada na nią maskę krzyżowania, a następnie zwraca tylko wybrane osobniki zgodnie z tą maską.
- void generateMask() - inicjalizuje maskę krzyżowania zgodnie z wartościami parametrów odpowiedzialnych za wielkość populacji oraz procent osobników do krzyżowania.

4.2.6 Code

Jest to klasa, która reprezentuje informacje o sekwencji kolorów i jej ocenie. Jest to klasa bazowa dla klasy **Specimen**.

Metody:

- *Score getScore() const* – zwraca ocenę kodu.
- *void setScore(const Score& score)* – przyjmuje w argumencie obiekt klasy **Score** i przypisuje go do tego kodu.
- *static Score compareCodes(const Code& code1, const Code& code2)* – przyjmuje dwa argumenty typu **Code** i zwraca obiekt klasy **Score** reprezentujący wynik porównania tych dwóch kodów.

4.2.7 Score

Przechowuje liczbę białych i czarnych pinesek jako wynik.

Metody:

- *bool isFullMatch() const* – sprawdza czy wynik reprezentuje całkowitą zgodność, czyli czy liczba czarnych pinesek jest równa długości kodu.
- *int distance(const Score& other) const* – zwraca odległość między dwoma wynikami na podstawie wcześnej omawianej metryki.

4.2.8 Specimen

Dziedziczy po klasie **Code**. Posiada pole fitness.

Metody:

- *void mutate()* – dokonuje mutacji osobnika.
- *Specimen crossWith(const Specimen& other)* – zwraca dziecko w wyniku krzyżowania z innym osobnikiem.
- *void setFitness(int fitness)* – ustawia wartość parametru fitness.

4.2.9 Statistics

Zawiera informacje o statystykach pojedynczej gry Mastermind. Pola tej klasy są publiczne dla łatwiejszego dostępu do ich wartości.

4.2.10 RNG

Zestaw statycznych funkcji realizujących losowanie liczb oraz sprawdzanie losowości zdarzeń. Klasa pomocnicza.

Metody:

- *static double uniformRandom(double lowerbound = 0.0, double upperbound = 1.0)* – zwraca losową liczbę rzeczywistą z przedziału <lowerbound, upperbound) zgodnie z rozkładem jednostajnym.
- *static int uniformRandom(int lowerbound, int upperbound)* – zwraca losową liczbę całkowitą z przedziału <lowerbound, upperbound> zgodnie z rozkładem jednostajnym.
- *static bool testEvent(double chance)* – przyjmuje w argumencie wartość procentową na wystąpienie wydarzenia i wykonuje sprawdzenie i zwraca, czy wydarzenie miało miejsce czy nie.

4.3 Krzyżowanie

Krzyżowanie polega na wzięciu pierwszych kilku kolorów (zdefiniowane przez parametr **splitIndex**) od jednego z rodziców, a następnie dobranie pozostałych kolorów od drugiego rodzica. Dla rozpatrywanego wariantu wybierane są 2 pierwsze kolory od pierwszego z rodziców oraz dwa ostatnie kolory od drugiego z rodziców.

4.4 Mutacja

Mutacja polega na wyborze jednej z pozycji i zwiększeniu lub zmniejszeniu wartości koloru o jeden w arytmetyce modulo równej liczbie dostępnych kolorów.

4.5 Parametry symulacji

W pliku **Parameters.hpp** znajduje się lista parametrów algorytmu genetycznego. Ważne, aby po każdej zmianie tych parametrów skompilować program ponownie z uwzględnieniem zmian w tym pliku nagłówkowym. Domyślny limit prób odgadnięcia hasła to 13. Jest to bardzo wysoki limit w porównaniu ze średnimi wynikami algorytmu. Domyślny stały rozmiar populacji w każdej epoce to 2000. Połowa z tych osobników bierze udział w krzyżowaniu oraz może zostać poddana mutacji, tak jak ich dzieci. Szansa na wywołanie mutacji u osobnika to domyślnie 10%.

5 Prezentacja działania

Testy przeprowadzone zostały na 10000 symulacjach na populacji z 2000 osobników. Limit prób odgadnięcia hasła to 13. Oto wyniki:

Dane o symulacjach, w których hasło zostało odgadnięte	
Liczba symulacji	9761
Średnia liczba prób na symulację	5.59636
Dane o symulacjach, w których hasło nie zostało odgadnięte	
Liczba symulacji	219
Procent	2.19 %

Surowe wyjście:

TERMINATED:
count: 239
percent: 2.39%

SUCCESS:
count: 9761
average: 5.61367

6 Ocena wyników

Algorytm osiągnął zaskakująco dobrą średnią prób odgadnięcia hasła. Nie jest to wynik lepszy od średniej dla algorytmu deterministycznego Knutha, ale wciąż uważam, że jak na pierwszy przeze mnie napisany algorytm genetyczny wyniki są bardzo zadowalające. Niski procent symulacji, w których nie odgadnięto hasła, wskazuje na fakt, iż algorytm jest skuteczny pod kątem odnajdywania rozwiązania, nawet jeżeli przychodzi mu to większym nakładem prób. Możliwe, że limit prób jest za duży. Jego zmniejszenie zwiększyłoby liczbę symulacji, w których nie udało się odnaleźć hasła, ale zmniejszyłoby to średnią liczbę prób na symulację.

Osiągnięcie takich wyników wiązało się z ciągłymi zmianami parametrów symulacji i obserwacje ich wpływu na wyniki gry. Najważniejszym parametrem są wielkość populacji oraz procent populacji wybieranej do krzyżowania. Mutacja również była bardzo ważnym czynnikiem, ale nie miała tak dużego wpływu na wyniki, jak pozostałe parametry.

Przy maksymalnej optymalizacji, program symuluje na jednym wątku 10000 w około 18 sekund, czyli 1,8 sekundy na 1000 symulacji.

7 Możliwości rozwoju

- Wprowadzenie wielowątkowości na poziomie symulowania wielu gier jednocześnie dla przyspieszenia testów.
- Dodanie interfejsu graficznego umożliwiającego edycję parametrów symulacji bez potrzeby rekompilacji.

- Dodanie innych implementacji mechanizmu krzyżowania oraz mutacji.
- Generowanie plików tekstowych z bardziej szczegółowymi statystykami o każdej grze, np. z listą prób odgadnięcia hasła.
- Możliwość podawania ziarna dla generatorów liczb losowych, w celu uzyskania deterministycznego zachowania algorytmu.

8 Wrażenia z realizacji projektu

Jestem wielkim fanem gier planszowych, dlatego kiedy zobaczyłem temat dotyczący gry Mastermind, który wykorzystuje algorytmikę i kreatywne metody rozwiązywania problemów wiedziałem, że chcę się podjąć jego realizacji. Pierwszy raz miałem przyjemność pracować z algorytmami genetycznymi podczas realizacji tego projektu. Nauczyłem się wielu istotnych rzeczy w kwestii ich działania, między innymi: w jaki sposób wybrać konkretną liczbę osobników z populacji zgodnie z pewnym rozkładem prawdopodobieństwa, jak istotny jest dobry mechanizm mutacji, w jaki sposób analizować pracę algorytmów genetycznych.

Ponieważ był to temat na pracę inżynierską, nie na projekt indywidulany, nie byłem w stanie zrealizować wszystkich elementów, które chciałbym zobaczyć w bardziej rozwiniętej wersji tego programu, przede wszystkim interfejs graficzny. Nie oznacza to jednak, że nie wrócę do pracy nad nim. Może nawet zdecyduję się spróbować swoich sił pisząc pracę inżynierską opisują właśnie tego typu algorytmy.

Biorąc pod uwagę brak wcześniejszej wiedzy w zakresie algorytmów genetycznych oraz brak praktyki w ich pisaniu, wyniki według dużych testów są zaskakująco satysfakcjonujące. Zbliżenie się średnią do 5 prób odgadnięcia hasła w pierwszych etapach pracy wydawało się być bardzo odległe, nierealne wręcz. Mimo to, analizując problem dogłębniej, uzyskałem niezbędną wiedzę, aby stworzony przeze mnie algorytm podołał zadaniu.