Wrocław 2021.12.09

Autor: Michał Przewoźniczek

Techniki Efektywnego Programowania – mini-projekt

Uwaga: mini-projekt ma służyć użyciu umiejętności zdobytych w ramach kursu, w tym w ramach wykonania wcześniejszych list zadań. Dlatego, w ramach mini-projektu można używać konstrukcji językowych oferowanych przez standard C++11 i wyższe. Ograniczenia, które nadal obowiązują to m.in.:

1. **Zakaz nieuzasadnionego rzucania wyjątków (tak jak dla wcześniejszych list).**
2. **Zakaz używania inteligentnych wskaźników, chyba że napisało się je samodzielnie (można użyć/rozbudować klasę zaimplementowaną w ramach listy nr 5).**

W ramach mini-projektu należy zaprojektować, oprogramować i zaprezentować działanie (w oddzielnym pliku, np. w *main*) klasy *CGAOptimizer, CGAIndividual*, oraz *CMax3SatProblem*.

Wymagania dla klas:

1. *CMax3SatProblem*
   - Klasa reprezentuje instancję problemu MAX-3-Sat. Pojedyncze rozwiązanie składa się z *n* zmiennych logicznych (wartości *true/false*) pogrupowanych w 3 elementowe klauzule. Na przykład jeśli problem jest zbudowanych z 10 zmiennych, to dwie klauzule mogą wyglądać tak: (-41 60 -175) (41 -185 47). Pierwsza klauzula grupuje zmienne 41, 60 i 175, natomiast druga grupuje zmienne 41, 185 i 47.
     Klauzula jest spełniona jeśli co najmniej jedna z trzech zgrupowanych wartości ma wartość *true*. Zauważ, że klauzule zawierają zmienne, które są zaprzeczone. Na przykład w drugiej klauzuli zmienna 185 jest zaprzeczona.

     Rozważmy następujące wartości zmiennych:
     - 41 = true
     - 47 = false
     - 60 = false
     - 175 = true
     - 185 = true

     Dla takich wartości pierwsza klauzula jest niespełniona, natomiast druga jest spełniona. Jeśli zmienimy wartość zmiennej nr 41 z *true* na *false*, to sytuacja ulegnie odwróceniu (spełniona będzie pierwsza klauzula, a druga nie).

     Powyższy przykład pokazuje również, że jeśli klauzul jest dużo, a ta sama zmienna wchodzi w skład wielu z nich, to spełnienie wszystkich klauzul, które może pozornie wydawać się proste, okazuje się bardzo trudne.

   - Klasa ma posiadać metodę Load (typ zwracany, oraz lista zmiennych – odpowiednio dobrana przez Autora), która pozwala na wczytanie instancji problemu Max-3-Sat z plików o formacie zamieszczonym na ePortalu. **Pliki mają być wczytywane w takiej formie w jakieś są podane. Nie wolno ich modyfikować.**
   - Metoda *Compute* (typ zwracany, oraz lista zmiennych – odpowiednio dobrana przez Autora) zwracająca jakość rozwiązania (liczbę spełnionych klauzul). Zwróć uwagę, że jednym z argumentów musi być zakodowane rozwiązanie (ciąg wartości 0/1, true/false, lub inny), które może przyjąć formę tablicy, listy, vectora, obiektu, lub inną. Typ argumentu jest wyborem Autora koda.
   - Zastanów się jak reprezentować, przechowywać i wyliczać stan poszczególnych klauzul

2. *CGAOptimizer*
   - Klasa optymalizatora. Optymalizator ma udostępniać interfejsy **programistyczne** pozwalające skonfigurować optymalizator. Dane, które muszą zostać pobrane to:
     - Rozmiar populacji rozwiązań
     - Prawdopodobieństwo krzyżowania
     - Prawdopodobieństwo mutacji
   - Klasa musi posiadać listę/tablicę/vektor osobników (proponowanych rozwiązań), które będzie przetwarzać w kolejnych iteracjach.
   - Klasa ma posiadać metodę *Initialize* (typ zwracany, oraz lista zmiennych – odpowiednio dobrana przez Autora) pozwalającą na przygotowanie optymalizatora do działania, w tym na losową inicjację wszystkich osobników (rozwiązań). Losowa inicjacja polega na tym, że jeśli rozwiązanie składa się np. ze 100 zmiennych to ich wartości określamy losowo.
   - Klasa ma posiadać metodę *RunIteration* (typ zwracany, oraz lista zmiennych – odpowiednio dobrana przez Autora) pozwalającą na uruchomienie pojedynczej iteracji optymalizatora, w której zostanie utworzona nowa populacja rozwiązań i nastąpi to w następujących krokach:

     nowaPopulacja ← empty
     while (*nowaPopulacja*.size < *Populacja*.size) **do**
         rodzic1 ← wybierzRodzica(*Populacja*);
         rodzic2 ← wybierzRodzica(*Populacja*);
         dziecko1, dziecko2 ← krzyżowanie(*rodzic1, rodzic2*);
         dziecko1 ← mutuj(*dziecko1*);
         dziecko2 ← mutuj(*dziecko2*);
         DodajOsobnika(*nowaPopulacja, dziecko1, dziecko2*);
     end while
     Populacja ← *nowaPopulacja*;

     GDZIE:

wybierzRodzica – metoda wyboru osobnika z populacji. Jedną z najprostszych programistycznie (a wysoce skutecznych) jest metoda turniejowa. Wybieramy losowo z równym prawdopodobieństwem t osobników, gdzie t to rozmiar turnieju (zwykle $t = 2$). Zwycięzcą zostaje ten osobnik rozwiązanie, które jest najlepszej jakości (w tym przypadku będzie to rozwiązanie, które spełnia najwięcej klauzul). Każdy osobnik może brać udział w dowolnej liczbie turniejów i może zostać rodzicem dowolną liczbę razy.

Krzyżowanie – metoda stworzenia potomstwa (dzieci). Najpierw sprawdzamy prawdopodobieństwo, czy w ogóle krzyżujemy. Na przykład, jeśli prawdopodobieństwo krzyżowania wynosi 0.3 i z przedziału <0; 1> wylosowaliśmy wartość 0.38 to krzyżowanie nie zachodzi. Wtedy *dziecko1* jest kopią *rodzica1*, a *dziecko2*, to kopia *rodzica2*. Jeżeli wylosujemy liczbę 0.13 to krzyżowanie zachodzi (bo 0.13 < 0.3). Są różne rodzaje krzyżowania, jedną z możliwości jest krzyżowanie jednostajne (ang. *uniform crossover*), gdzie dla każdej wartości genu (zmiennej) *dziecko1* losujemy czy wziąć ją z *rodzica1*, czy z *rodzica2*. *Dziecko2* jest przeciwieństwem dziecka1, tzn., że jeśli *dziecko1* bierze dany gen z *rodzica2*, to *dziecko2* bierze z *rodzica1* i odwrotnie.

Mutuj – metoda mutacji. Można ją wykonać w taki sposób, że dla każdego genu po kolei w danym osobniku sprawdzamy prawdopodobieństwo mutacji. Jeśli mutacja zachodzi, to zmieniamy wartość genu zmiennej na odwrotną. W trakcie procesu mutacji wiele genów może ulec zmutowaniu.

3. *CGAIndividual*

- Klasa osobnika (rozwiązania problemu)
- Musi posiadać genotyp, czyli rozwiązanie w formie tablicy/wektora/listy wartości *true/false* (konkretny typ i forma przechowywania to decyzja Autora)
- Musi posiadać metodę *Crossover*, wykonującą krzyżowanie osobników
- Musi posiadać metodę *Mutation*, wykonującą mutację danego osobnika
- Musi posiadać metodę *Fitness*, zwracającą przystosowanie/jakość (ang. *fitness*) dla danego osobnika

UWAGI:

1. Zastanów się, jaka powinna być relacja pomiędzy obiektem klasy problemu, a obiektem klasy optymalizatora. Czy któraś klasa powinna być właścicielem drugiej? Weź pod uwagę, że wykonany optymalizator może być w przyszłości użyty do rozwiązywania różnych problemów.
2. **Na zajęciach laboratoryjnych będzie oceniana jakość kodu, a nie jakość optymalizacji.**
3. **Niemniej jednak wykonanie mini-projektu daje możliwość wzięcia udziału w konkursie.**