Wrocław 2022.12.08

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 zadania należy napisać własną metodę Algorytmu Genetycznego (AG). AG służy do wyszukiwania jak najlepszych rozwiązań dla problemów optymalizacyjnych. Niniejsze zadanie jest zadaniem programistycznym, nie ma na celu pokazania całego spektrum możliwych zastosowań AG. Może jednak stanowić intelektualną rozrywkę dla osób, którym spodoba się ta tematyka.

Niniejsze zadanie, dla celów dydaktycznych związanych z kursem TEP, ograniczy się do rozwiązywania binarnego problemu plecakowego (0/1 Knapsack Problem), jednak implementacja dobrej jakości będzie mogła być bezpośrednio użyta również w konkursie programowania w C++ prowadzonym w ramach przedmiotu.

UWAGA – w ramach oceny z laboratorium będzie oceniana wyłącznie jakość dostarczonego kodu. Jakość działania AG (jakość optymalizacji) nie będzie brana pod uwagę.

Binarny problem plecakowy

Wyobraźmy sobie, że mamy dane n przedmiotów i plecak. Plecak ma pojemność W. Każdy przedmiot zajmuje ω_i miejsca w plecaku, gdzie i to numer przedmiotu. Chcemy zabrać do plecaka jak najwięcej przedmiotów, ale nie możemy przekroczyć jego pojemności. A więc:

$$\sum_{i=1}^{n} \omega_i x_i \leq W, \text{ gdzie} \quad (1)$$

x_i przyjmuje wartość 1 jeżeli chcemy spakować przedmiot do plecaka, lub 0 w przeciwnym przypadku. Tak więc, wzór (1) to ograniczenie, które mówi nam, że rozmiar wszystkich spakowanych do plecaka przedmiotów nie może przekroczyć objętości plecaka.

Każdy przedmiot ma również swoją wartość v_i. Można policzyć wartość wszystkich spakowanych przedmiotów:

$$V(x) = \sum_{i=1}^{n} v_i x_i \quad (2)$$

Zadanie w binarnym problemie plecakowym jest następujące. Spośród dostępnych przedmiotów, należy zapakować do plecaka takie, których sumaryczna wartość będzie jak największa (wzór (2)), nie wolno jednak przy tym złamać ograniczenia, którym jest rozmiar samego plecaka (wzór (1)).

Przykład:

Mamy $n=4$ przedmioty, których wartość i masa została przedstawiona w **Tabela** 1. Pojemność plecaka wynosi $W = 5$.

Tabela 1 –wartości i rozmiar przedmiotów

i	Wartość (v_i)	Rozmiar (ω_i)
1	5	4
2	1	1
3	4	3
4	3	2

Jedne z możliwych rozwiązań to:

- Pakujemy przedmioty 2 i 4. Wtedy ich wartość wynosi $1 + 3 = 4$. Natomiast rozmiar wynosi: $1 + 2 = 3$. Takie rozwiązanie jest dopuszczalne, ponieważ $3 \leq W$, czyli $3 \leq 5$.
- Pakujemy przedmioty 1, 2 i 3. Wtedy wartość przedmiotów wynosi aż $5 + 1 + 4 = 10$. Jednak nie jest to rozwiązanie dopuszczalne, ponieważ rozmiar przedmiotów wynosi $4 + 1 + 3 = 8$. W związku z tym, że $8 > 5$ rozmiar plecaka został przekroczony, czyli takie rozwiązanie nie jest dopuszczalne.
- Dla tak małej instancji problemu nietrudno zauważyć, że rozwiązanie optymalne to spakowanie przedmiotów 3 i 4. Co ciekawe, spakowanie najbardziej wartościowego przedmiotu nr 1 nie prowadzi do znalezienia optymalnego rozwiązania.

Rozwiązania dla problemu można przedstawić za pomocą ciągu 0 i 1 o długości n. A więc dla powyższego przykładu pierwsze rozwiązanie zostanie zakodowane jako 0101. Drugie (niedopuszczalne) jako 1110. Natomiast rozwiązanie optymalne to 0011. A więc każde 0 lub 1 będzie odpowiadać x_i we wzorach (1) i (2).

Więcej informacji na temat binarnego problemu plecakowego można znaleźć w Internecie. Między innymi na stronach:

https://en.wikipedia.org/wiki/Knapsack_problem
http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/
https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

Z powyższego wywodu wynika, że rozwiązanie dowolnego binarnego przedmiotu plecakowego, który dotyczy *n* przedmiotów można zapisać w postaci ciągu *n* 0 i 1.

Podstawy działania algorytmu genetycznego

Do poszukiwania dobrych jakościowo rozwiązań takich problemów jak binarny problem plecakowy służy między innymi metoda nazywana Algorytmem Genetycznym. Główna idea jej działania polega na stworzeniu populacji losowo wybranych rozwiązań. Następnie, zgodnie z ideą ewolucji wybieramy rozwiązania jakościowo lepsze i krzyżujemy je z innymi dobrymi jakościowo rozwiązaniami. Liczymy na to, że skoro rodzice byli dobrej jakości, to być może wśród ich potomstwa trafi się takie, które będzie jeszcze lepsze.

W przypadku opisanego powyżej problemu, pojedyncze rozwiązanie można zakodować jako tablicę *n* liczb typu *int* o wartościach 0 lub 1.

Poszczególne kroki algorytmu genetycznego są przedstawione poniżej, każdy z nich jest opisany dokładniej w dalszej części dokumentu.

Zmienne wejściowe: *PopSize* (rozmiar populacji), *CrossProb* (prawdopodobieństwo krzyżowania), *MutProb* (prawdopodobieństwo mutacji)

1. Wygeneruj populację *PopSize* losowych rozwiązań
2. Oceń wszystkie rozwiązania wyliczając wartość dla każdego wartość przedmiotów. Uwzględnij fakt, czy rozwiązanie jest dopuszczalne.
3. Wykonaj krzyżowanie
4. Wykonaj mutację
5. Jeśli osiągnąłeś warunek zatrzymania metody (np. upłynął czas dostępny na obliczenia) to zakończ, jeśli nie, to wróć do punktu 2.

Populacja i jej generowanie

W algorytmie genetycznym (AG) pojedyncze rozwiązanie jest nazywane *osobnikiem*. Żeby AG mogło rozpocząć swoje działanie należy najpierw wygenerować populację (pewną grupę osobników). Liczbę osobników określa użytkownik, w niniejszym zadaniu parametr ten nazywa się *PopSize*. Początkowa populacja jest generowana losowo. Należy użyć generatora liczb pseudolosowych. W C++98 do generowania liczb pseudolosowych służą komendy *srand(time(NULL))* (ustawienie tzw. *seeda* wywoływane jeden raz, na starcie programu) oraz rand() (patrz: http://www.cplusplus.com/reference/cstdlib/rand/). **W tym zadaniu można również użyć mechanizmu z C++11 (i jest to zalecane, ale nie nakazywane) dokumentacja: https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution.**

Przykład

Wykorzystując AG chcemy szukać dobrych rozwiązań problemu plecakowego dla *n=4* przedmiotów (problem jak w przykładzie powyżej), a zadany *PopSize = 4*, to populacja początkowa może wyglądać następująco:

Osobnik 1: 1011
Osobnik 2: 0010
Osobnik 3: 1110
Osobnik 4: 0101

Ocena osobników
W AG wartość oceny osobnika nazywa się *przystosowaniem* (ang. *fitness*). W omawianym przykładzie można uznać, że *fitness = wartość przedmiotów*, jeśli rozwiązanie jest dopuszczalne, oraz *fitness = 0*, jeśli rozwiązanie jest niedopuszczalne.

A więc dla osobników z powyższego przykładu przystosowanie będzie następujące:
Osobnik 1: 1011 (przystosowanie=0; wartość przedmiotów to 9, ale ich rozmiar jest za duży)
Osobnik 2: 0010 (przystosowanie=3; przedmiot mieści się w plecaku)
Osobnik 3: 1110 (przystosowanie=0)
Osobnik 4: 0101 (przystosowanie=4)

Można również definiować bardziej wyrafinowane funkcje przystosowania. Dobrze zaprojektowane mogą pomóc w optymalizacji.

Krzyżowanie
W normalnym procesie ewolucji osobniki żyjące w naturze krzyżują się i mają wspólne potomstwo. Podobnie jest w AG. Żeby wytworzyć potomstwo, potrzebnych jest dwoje rodziców. Jak ich wybrać? Z istniejącej populacji wybieramy losowo dwa osobniki i konkurują one bezpośrednio między sobą, wybieramy tego, który jest lepszy. Krzyżowanie jest powtarzane tak długo, aż utworzonych zostanie *PopSize* nowych osobników.

Przykład
Mamy populację przedstawioną i ocenioną tak, jak w poprzednim przykładzie. Chcemy utworzyć nowe pokolenie. Wybieramy losowo (z równym prawdopodobieństwem 25% dla każdego osobnika, bo osobników jest 4) dwa osobniki. Załóżmy, że wybraliśmy osobnika nr 2 i osobnika nr 3. Przystosowanie osobnika nr 2 jest wyższe, więc wybieramy go jako pierwszego rodzica. Następnie znów losujemy dwa osobniki i wybieramy jednego z nich jako rodzica. Załóżmy, że wylosowaliśmy osobnika nr 2 (znowu, ale jest to dopuszczalne) oraz osobnika nr 4. Jako drugiego rodzica wybieramy więc osobnika nr 4. Jako rodziców mamy więc wybrane 2 osobniki: 2 (z pierwszej losowej pary) i 4 (z drugiej losowej pary).

Następnie sprawdzane jest prawdopodobieństwo krzyżowania (*CrossProb*), o którym mowa na poprzedniej stronie. Sprawdzamy (losowo!), czy krzyżowanie ma nastąpić, czy nie. Na przykład, jeśli *CrossProb=0.6* (czyli 60%), a z zakresu [0-1] wylosowaliśmy 0.445, to krzyżowanie nastąpi, bo 0.445 < 0.6 (co jeśli nie, jest opisane przy drugiej parze rodziców).

Mamy dwoje rodziców o zakodowanych ciągiem 0 i 1, zwanym dalej *genotypem*.

Rodzic 1: 0010
Rodzic 2: 0101

Genotyp rodzica może zostać przecięty na dwie części w jednym z 3 miejsc (*n-1*). Na przykład Rodzic 1, może być przecięty na dwie części na jeden z następujących sposobów:
Punkt 1: 0 010
Punkt 2: 00 10
Punkt 3: 001 0

Dla obu rodziców losujemy jeden z 3 dostępnych punktów krzyżowania. Załóżmy, że wylosowaliśmy punkt nr 3. Wtedy każdego z rodziców dzielimy na dwie następujące części:

Rodzic 1: 001 0
Rodzic 2: 010 1

Z tej dwójki rodziców powstanie dwójka dzieci. Dzieci tworzymy w następujący sposób.
Dziecko nr 1 = część 1 Rodzcia 1 + część 2 Rodzica 2.
Dziecko nr 2 = część 2 Rodzcia 1 + część 1 Rodzica 2.

A więc, w powyższym przykładzie:
Dziecko nr 1: **0011**
Dziecko nr 2: **0100**

Dzieci nr 1 i 2 są wstawiane do kolejnej populacji. Na razie są w niej tylko te 2 osobniki, a wymagany *PopSize=4*. W związku z powyższym losujemy kolejną parę rodziców, zgodnie z procedurą opisaną powyżej. Załóżmy, że jako rodziców wybraliśmy osobnika 1 i 4. Proszę zauważyć, że osobnik nr 4 został wybrany jako rodzic już drugi raz. Nie ma tutaj żadnych ograniczeń. Załóżmy, że tym razem sprawdzając, czy ma nastąpić krzyżowanie wylosowaliśmy 0.778. Ponieważ 0.778 > *CrossProb* (które jest równe 0.6), to krzyżowanie nie następuje. W takiej sytuacji w kolejnej populacji znajdą się kopie rodziców. A więc następna populacja została utworzona z osobników o genotypach:

0011 (dziecko z krzyżowania osobników 2 i 4)
0100 (dziecko z krzyżowania osobników 2 i 4)
1011 (kopia osobnika 1)
0101 (kopia osobnika 4)

Mutacja
W przypadku mutacji, dla realizacji niniejszego zadania należy przyjąć, że przebiega ona następująco. Dla każdego osobnika i dla każdego jego genu oddzielnie, sprawdzamy, czy mutacja zachodzi. Losujemy liczbę z zakresu [0;1] i porównujemy ją z prawdopodobieństwem mutacji *MutProb*. Jeśli wylosowana liczba jest mniejsza, to mutujemy gen, w przeciwnym przypadku pozostawiamy osobnika w niezmienionym stanie.

Na przykład:
Dla osobnika o genotypie **0011** wykonujemy mutację. Załóżmy, że *MutProb=0.1*.

Sprawdzamy mutację dla genu nr 1. Losujemy liczbę z zakresu [0;1] i porównujemy ją z *MutProb*. Wylosowaliśmy 0.221, a więc gen pozostaje niezmieniony.

Sprawdzamy mutację dla genu nr 2, wylosowaliśmy 0.02, a więc mutacja zachodzi. W takiej sytuacji zmieniamy wartość genu na przeciwną, czyli aktualny genotyp wygląda teraz tak: **0111.** Na drugiej pozycji było 0, ale zostało zmienione na 1, bo drugi gen uległ mutacji.

Następnie sprawdzamy mutację dla genu nr 3. Wylosowlaiśmy 0.154, a więc mutacja nie zachodzi i genotyp osobnika nadal ma postać **0111** (na czerwono zaznaczony jest zmutowany wcześniej gen nr 2).

Wreszcie sprawdzamy mutację dla genu nr 4. Wylosowalismy 0.003, a więc gen jest mutowany. W związku z powyższym genotyp po mutacji ma postać: **0110**.

Zauważ, że w efekcie mutacji osobnika może tak się zdarzyć, że żaden gen nie zostanie zmieniony (zmutowany), może być tak, że zmianie ulegnie 1 gen, a może być ich też kilka. W skrajnym przypadku mogą zostać zmutowane wszystkie geny.

Wymagania do programu:

W ramach wykonania programu należy oprogramować, co najmniej następujące klasy:

- **CKnapsackProblem**
  - Klasa ma pozwalać na obsługę konkretnych instancji binarnego problemu plecakowego
  - Należy umożliwić skonfigurowanie konkretnego obiektu tej klasy (wprowadzenie informacji o liczbie przedmiotów, tabeli rozmiarów, oraz wartości)
  - Należy pamiętać, że dane podane z wyższej warstwy systemu mogą być błędne (np. liczba przedmiotów $n=-234$). W takiej sytuacji należy w odpowiedni sposób informować o błędach. (Dodatkowa wskazówka: warto rozważyć jak oprogramować konstruktor. Przykład z wykładu z wczytywaniem z pliku może być pomocny)
  - Należy dostarczyć metodę, pozwalającą na wyliczenie wartości rozwiązania zapisanego na konkretnym genotypie (tablicy, wektorze, liście zer i jedynek)
  - Zastanów się jaki typ danych powinien przechowywać informację o pojemności plecaka, wartości przedmiotu i ilości zajmowanego przez przedmiot miejsca, tak aby było to możliwie ogólne.
  - Należy zapewnić metodę do wczytywania instancji problemu z pliku

- **CGeneticAlgorithm**
  - Klasa do obsługi uruchomienia konkretnej instancji AG
  - Należy uwzględnić parametry wykonania AG (rozmiar populacji, prawdopodobieństwo krzyżowania i mutacji)
  - Jako kryterium zatrzymania można przyjąć liczbę iteracji metody, czas obliczeń, albo liczbę wywołań metody wyliczającej jakość danego rozwiązania
  - Po zakończeniu przebiegu AG obiekt ma dawać możliwość pobrania najlepszego rozwiązania znalezionego w trakcie przebiegu metody

- **CIndividual**
  - Klasa osobnika
  - Osobnik musi posiadać genotyp, kodujący rozwiązanie
  - Wymagana jest metoda wyliczająca przystosowanie danego osobnika
  - Wymagana jest metoda mutująca danego osobnika
  - Wymagana jest metoda pozwalająca skrzyżować danego osobnika z innym i zwracająca utworzone w ten sposób dzieci

Wyniesienie powyższych funkcjonalności poza klasę będzie traktowane jak przejaw programowania strukturalnego i karane jako nieodpowiednie przypisanie funkcjonalności obiektom.

Zastanów się jak ustalić relacje pomiędzy obiektem optymalizatora, a obiektem problemu. Które obiekty będą ze sobą w relacji całość-część?

Program nie musi posiadać interfejsu użytkownika. Wystarczy wykonanie przykładowego przebiegu z poziomu funkcji main.