Algorytmy metaheurystyczne

Lista 3

Piotr Syga

3 maja 2020

Termin: 2020-05-23, 09:59:59

1 Cel listy

Celem listy jest praktyczne przećwiczenie metaheurystyk populacyjnych. Dobór parametrów (np. kodowanie, liczność populacji, liczba iteracji, operatory stochastyczne) należy do autora programu. W czasie rozwiązywania listy autor powinien rozpoznać jaki wpływ na działanie programu (czas działania, wymagania pamięciowe, osiągany rezultat, podatność na utknięcia w lokalnym minimum, ...) mają poszczególne parametry.

2 Wymagania formalne

Rozwiązanie listy powinno zostać umieszczone w ścieżce `amh/l3/` (case sensitive) głównego katalogu studenta na repozytorium svn—`https://156.17.7.16/`. Rozwiązanie każdego zadania powinno znajdować się w odpowiednim folderze `zi/`, dla zadania `i`. Poza plikami źródłowymi rozwiązania, w tym samym folderze należy umieścić `makefile`, który po wpisaniu polecenia `make` utworzy w bieżącym folderze plik wykonywalny `main`. Wszystkie dane wejściowe podawane są na standardowym wejściu, na standardowym wyjściu powinno znajdować się jedynie rozwiązanie w ustalonym formacie. Program będzie uruchamiany poprzez przekierowanie danych wejściowych i wyjściowych z/do plików: `./main <In >Out` (lub `./main <In >Out 2>Err`). Program **nie może** wykorzystywać obliczeń równoległych, w szczególności zabronione jest testowanie kilku rozwiązań jednocześnie.

3 Zadania

1. Napisz program, który za pomocą wybranego algorytmu populacyjnego (sugerowane podejście: rój cząstek lub algorytm genetyczny z reprezentowaniem genów przez liczbę rzeczywistą) znajdzie minimum funkcji X.S. Yang zadanej wzorem

$$f(\bar{x}) = \sum_{i=1}^{5} \varepsilon_i |x_i|^i \ ,$$

dla ε_i będących niezależnymi zmiennymi losowymi z rozkładu jednostajnego na [0, 1].

In: 11 liczb (6 całkowitych i 5 typu double) `t x1 x2 x3 x4 x5` ε_1 ε_2 ε_3 ε_4 ε_5 oddzielonych spacją.
`t` – maksymalna liczba sekund, którą może wykonywać się program w tym uruchomieniu, $\bar{x}$ = (`x1`, `x2`, `x3`, `x4`, `x5`) – jest rozwiązaniem początkowym (podane liczby zawsze są całkowite, program natomiast może je interpretować jako dowolny typ liczbowy). ε_i są współczynnikami funkcji reprezentowanymi przez liczby typu double.
Uwaga: Dla każdego i $|x_i| \leqslant 5$.

Out: 6 liczb typu `double` oddzielonych spacją, z czego cztery pierwsze to $\bar{x}$, natomiast ostatnia to wartość funkcji w punkcie $\bar{x}$.

2. Napisz program, który za pomocą algorytmu genetycznego będzie generował słowa, które można ułożyć z otrzymanego multizbioru liter, należące do załączonego słownika, maksymalizując ich punktację. Zakładamy, że w folderze bieżącym pliku `main` znajduje się słownik dopuszczalnych słów `dict.txt` (przykład tutaj, testy mogą być prowadzone dla mniejszego słownika, nie będą prowadzone dla słownika liczniejszego od przykładowego, ograniczamy się do ASCII), który służy do weryfikacji dopuszczalności uzyskanego rozwiązania. Słowo uważane jest za niedopuszczalne również wtedy, gdy nie można go ułożyć z dostępnych liter (liczba kopii jest brana pod uwagę). Jeśli rozwiązanie jest dopuszczalne, za jego punktację przyjmujemy sumę punktów wszystkich liter, które się na nie składają.

In: Wejście składać się będzie z $n + s + 1$ linii. W pierwszej linii znajdować się będą trzy liczby całkowite oddzielone spacjami: `t` n s. Liczba `t` jest ograniczeniem czasowym jak w zadaniu pierwszym, liczba n określa rozmiar multizbioru liter, s oznacza liczbę zadanych na wejściu rozwiązań dopuszczalnych. W kolejnych n liniach znajdować się będą oddzielone spacją pary c_i p_i, gdzie c_i jest i-tym elementem multizbioru (i-tą literą), natomiast p_i jej wartością punktową. Można przyjąć, że wszystkie kopie tej samej litery mają identyczną wartość punktową. W s ostatnich liniach znajdować się

będą kolejno słowa (bez rozdzielających litery spacji) będące dopuszczalnymi rozwiązaniami dla danego wywołania.
Przykładowe dane wejściowe można znaleźć tu i tutaj.

Out: Na standardowym wyjściu powinna zostać zwrócona liczba całkowita będąca sumą punktów uzyskanego rozwiązania. Na standardowym wyjściu błędów powinno zostać wypisane rozwiązanie końcowe (w przypadku większej ilości informacji, słowo końcowe powinno być jedynym ciągiem znaków w ostatniej linii stderr).

Uwaga: Słownik zawsze będzie uporządkowany w kolejności alfabetycznej. W ramach preprocessingu można reorganizować słownik (np. drzewo prefiksowe) natomiast `t` jest ograniczeniem na całkowity czas działania programu.
Uwaga 2: Zakładamy, że kapitalizacja liter nie ma znaczenia.

3. Napisz program, który będzie symulował poruszanie się agenta po kracie (wycinek $\mathbb{Z}^2$). Celem jest dotarcie agenta do wyznaczonego punktu, przy założeniach, że w każdym kroku może poruszyć się o 1 w lewo, o 1 w prawo, o 1 w górę albo o 1 w dół. Program powinien za pomocą algorytmu genetycznego generować kolejne rozwiązania, tak by dotarcie do celu zajęło jak najmniej rund. Jeśli agent dotrze do celu przed wykonaniem wszystkich kroków, liczymy liczbę wykonanych kroków, jeśli po wykonaniu całej wygenerowanej sekwencji kroków, agent nie dotarł do celu - kontynuuje z miejsca, w którym wylądował a długość wykonanej sekwencji wlicza się do bieżącego rozwiązania albo zaczyna nową iterację.

   In: Dane wejściowe składać się będą z $n + s + 1$ linii. W pierwszej linii będą umieszczone, oddzielone spacją liczby całkowite `t`, n, m, s i p , gdzie `t` jest limitem czasu, jak w zadaniu 1, n i m oznaczają wymiary kraty (odpowiednio liczba wierszy i kolumn w labiryncie, który będzie chciał opuścić agent). Liczba całkowita s oznacza liczbę danych na wejściu rozwiązań początkowych, natomiast $p \geqslant s$ określa górne ograniczenie na liczebność populacji. W kolejnych s liniach będą znajdować się ciągi znaków `U`, `D`, `R`, `L` prowadzące z punktu startowego do jednego z wyjść (ciągi mogą być różnej długości, mogą prowadzić do różnych wyjść). W ostatnich n liniach będą znajdowały się cyfry tworzące labirynt. Między cyframi **nie będzie** spacji. Możliwe cyfry:

   - **0** – standardowe, puste pole, po którym agent może się poruszać
   - **1** – ściana, która nie może zostać pokonana (Uwaga: ściany mogą znajdować się również wewnątrz labiryntu, **nie ma** wymagania by przynajmniej jednym sąsiadem `1` była `1`; ściany mogą całkowicie blokować dostęp do części labiryntu, pod warunkiem, że istnieje ścieżka z pozycji startowej agenta do przynajmniej jednego wyjścia.).

5 – symbol agenta, oznaczający jego pozycję początkową (Uwaga: nie ma konieczności wizualizacji kolejnych kroków).

8 – symbol wyjścia, oznaczający pozycję celu, na który agent powinien dotrzeć (Uwaga: w danej instancji może być więcej niż jeden symbol `8`, **nie wszystkie** `8` muszą się znajdować on na obrzeżu).

Uwaga: Agent nie zna swojej pozycji początkowej, ani pozycji celu. Można założyć, że agent potrafi rozpoznać rodzaj pola (cyfrę) swoich czterech sąsiadów oraz, że zna n oraz m.
Przykładowe dane wejściowe można znaleźć tu, tutaj i tutaj. Przykłady z list wcześniejszych mają inny format wejścia zatem przed użyciem na L3 muszą zostać zmodyfikowane.

Out: Na standardowym wyjściu znaleźć się powinna jedynie łączna liczba kroków k od pozycji startowej do celu, wykonana w wybranym rozwiązaniu. Ostatnią linią na standardowym wyjściu błędów powinien być k-elementowy ciąg znaków `U`, `D`, `R`, `L` oznaczający kolejne wybrane kierunki w rozwiązaniu, gdzie litery kodują odpowiednio krok w górę, krok w dół, krok w prawo i krok w lewo.

Uwaga: Rozbudowana wersja zadania 3 będzie celem projektu.