Algorytmy metaheurystyczne

Lista 1

Piotr Syga

15 marca 2020

Termin: 2020-03-30, 09:59:59

1 Cel listy

Celem listy jest praktyczne przećwiczenie metaheurystyk opratych na lokalnym przeszukiwaniu, w szczególności Tabu Search. Dobór parametrów (np. długość listy Tabu, definicję sąsiedztwa, wybór sąsiedniego rozwiązania i liczba sprawdzanych sąsiadów czy wybór pierwszego testowanego rozwiązania) 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/l1/` (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`. Program **nie może** wykorzystywać obliczeń równoległych, w szczególności zabronione jest testowanie kilku rozwiązań jednocześnie.

3 Zadania

1. Napisz program, który za pomocą wybranej modyfikacji przeszukiwania lokalnego znajdzie minimum funkcji.

   (a) `HappyCat`: $h(\bar{x}) = \left[\left(||\bar{x}||^2 - 4\right)^2\right]^{\frac{1}{8}} + \frac{1}{4}\left(\frac{1}{2}||\bar{x}||^2 + \sum_{i=1}^{4} x_i\right) + \frac{1}{2}$, dla $\bar{x} = (x_1, x_2, x_3, x_4)$.

   (b) `Griewank`: $g(\bar{x}) = 1 + \sum_{i=1}^{4} \frac{x_i^2}{4000} - \prod_{i=1}^{4} \cos(\frac{x_i}{\sqrt{i}})$, dla $\bar{x} = (x_1, x_2, x_3, x_4)$.

   In: Para liczb całkowitych `t b` oddzielonych spacją.
   `t` – maksymalna liczba sekund, która może wykonywać się program w tym uruchomieniu, `b` – jeśli parametr ma wartość 0, to powinien minimalizować funkcję h, w p.p. funkcję g.

   Out: 5 liczb typu `double` oddzielonych spacją, z czego cztery pierwsze to $\bar{x}$, natomiast piąta to wartość odpowiedniej funkcji w punkcie $\bar{x}$.

2. Napisz program, który dla przedstawionej instancji problemu TSP znajdzie, z wykorzystaniem Tabu Search, cykl o możliwie najmniejszym koszcie, rozpoczynając z pierwszego miasta.

   In: Dane wejściowe składać się będą z $n+1$ linii. W pierwszej linii będą umieszczone, oddzielone spacją liczby całkowite `t` i n, gdzie `t` jest limitem czasu, jak w zadaniu 1, natomiast n liczbą miast do odwiedzenia. W kolejnych n liniach będą znajdowały się odległości pomiędzy miastami oddzielone co najmniej jedną spacją. Uwaga, odległość z miasta i do miasta i zawsze wynosi 0, natomiast nie należy zakładać, że mamy zadany problem Metric TSP, ani że odległości między wybranymi dwoma miastami są takie same w obu kierunkach. Przykładowe dane wejściowe można znaleźć tu i tutaj.

   Out: Na standardowym wyjściu znaleźć się powinien jedynie koszt pokonania cyklu. Ostatnią linią na standardowym wyjściu błędów powinien być $(n+1)$-elementowy ciąg liczb całkowitych oddzielonych spacjami, oznaczający kolejno odwiedzane miasta, np. `1 5 4 3 2 6 1`.

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 Tabu Search powinien generować kolejne sekwencje kroków, 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.

In: Dane wejściowe składać się będą z $n+1$ linii. W pierwszej linii będą umieszczone, oddzielone spacją liczby całkowite `t`, n i m, gdzie `t` jest limitem czasu, jak w zadaniu 1, natomiast n i m oznaczają wymiary kraty (odpowiednio liczba wierszy i kolumn w labiryncie, który będzie chciał opuścić agent). W kolejnych 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 będą znajdować się jedynie na obrzeżach, nie będzie ścian wewnątrz labiryntu).
- **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: jest dokładnie jeden symbol 8 oraz znajduje się 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 i tutaj.

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: Zadanie 3. będzie rozbudowywane na kolejnych listach (dla różnych podejść metaheurystycznych), rozbudowana wersja zadania będzie również celem projektu.