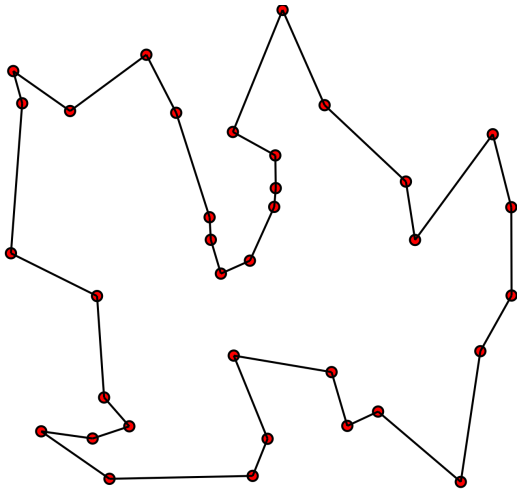
Algoritmi Avansați
Laborator 2

Gabriel Majeri

Exerciții

1. (Problema comisului-voiajor [1], în spațiul euclidian). *Travelling salesman problem* este o problemă clasică din domeniul optimizării. Deși enunțul este simplu de înțeles (vrem să găsim un drum de cost minim care să treacă prin toate orașele de pe o hartă), problema este computațional dificilă (în clasa *NP-hard* [2]).



Sursa imaginii: Wikipedia

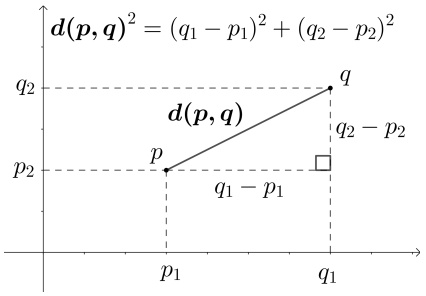
În acest exercițiu vom implementa câțiva algoritmi care să rezolve problema în cazul special în care orașele sunt puncte în plan.

1. Scrieți un subprogram care **să citească „orașele" dintr-un fișier**, pentru fiecare oraș fiind date coordonatele x și y (ca numere reale).

   Hint: puteți găsi pe internet exemple pentru cum să citiți date din fișier în Python [3], respectiv în C++ [4].

2. Scrieți un subprogram care să calculeze **distanța euclidiană** între două orașe:

$$d(\underbrace{\text{ORAȘ}_1}_{=(x_1,y_1)}, \underbrace{\text{ORAȘ}_2}_{=(x_2,y_2)}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



Sursa imaginii: Wikipedia

3. Scrieți un subprogram care să transforme lista de orașe într-un **graf complet**, unde **nodurile** sunt **orașele**, iar **ponderile** muchiilor sunt **distanțele euclidiene**, calculate cu subprogramul implementat anterior.

   Hint: puteți reprezenta graful cu o matrice de adiacență sau folosind liste de adiacență.

4. Implementați o soluție *brute force* pentru problema comisului-voiajor. Soluția se poate baza pe *backtracking*, care recursiv s-ar implementa în felul următor [5]:

   (a) Alegem orice oraș ca nodul de plecare.

   (b) Inițializăm o listă (globală) în care să reținem **soluția curentă**, o variabilă în care să reținem **costul soluției curente** și o variabilă în care reținem **costul minim** găsit (inițial infinit).

   (c) În mod recursiv:

   - Dacă nu am construit încă o soluție:
     i. Luăm pe rând toate orașele care nu sunt deja în soluție.
     ii. Adăugăm la soluție orașul și adăugăm costul muchiei la costul total.
     iii. Apelăm recursiv subprogramul de backtracking.
     iv. Eliminăm orașul din listă și îl încercăm pe următorul.

- Dacă ajungem la o listă care conține toate orașele, comparăm costul ei cu costul minim, și îl actualizăm în mod corespunzător.

(d) Afișăm costul minim găsit.

Algoritmul poate fi testat pe un număr mic de orașe (ex. 5-6 orașe), deoarece complexitatea lui de timp este $\mathcal{O}(n!)$.

5. (Nearest neighbour algorithm [6]) O soluție aproximativă se bazează pe metoda *celui mai apropriat vecin*: pornim de la un oraș oarecare, și mereu alegem dintre orașele nevizitate pe **cel cu distanța minimă** față de cel curent.

   Făcând alegerea „optimă" la fiecare pas, algoritmul descris mai sus este de tip *greedy*. Din păcate, deși rezolvă rapid problema, nici nu se poate considera că este un algoritm α-aproximativ — pentru orice α, se poate găsi un contraexemplu în care obține soluția cea mai proastă, deci nu are un factor de aproximare constant [7, 8].

   Implementați acest algoritm și vedeți ce soluție obține pe niște exemple (în comparație cu soluția optimă produsă de algoritmul cu backtracking).

6. (Double-tree algorithm [9]). Un algoritm aproximativ care ne oferă garanția că soluția obținută are, în cel mai rău caz, un cost de 2 ori mai mare decât soluția de cost minim, este „algoritmul arborelui dublu".

   În linii mari, algoritmul are următorii pași:

   (a) Construiește un arbore de acoperire de cost minim (*minimum spanning tree*) pentru graful cu orașele [10]. Puteți folosi orice algoritm cunoscut: Prim [11], Kruskall [12], ștergere inversă [13] etc.

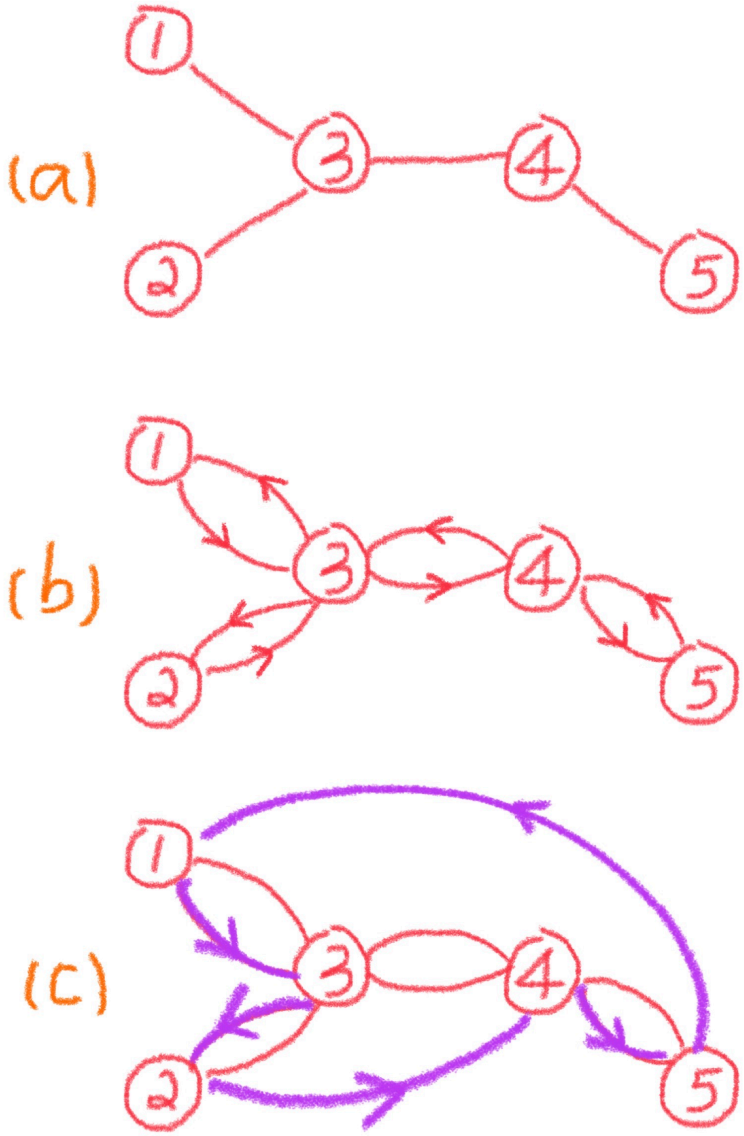
   (b) „Dublează" arborele de acoperire, transformându-l într-un graf orientat (nu e nevoie să faceți efectiv nimic, parcurgerea descrisă în [14] funcționează și pe arborele obișnuit).

   (c) Determină un tur eulerian pentru arbore, folosind o parcurgere în adâncime [14].

   (d) Elimină nodurile care se repetă în turul eulerian (i.e. păstrează doar prima apariție a fiecărui nod), realizând astfel niște „scurtături".

   (e) Lista de noduri rămasă constituie soluția aproximativă la problema comisului-voiajor.

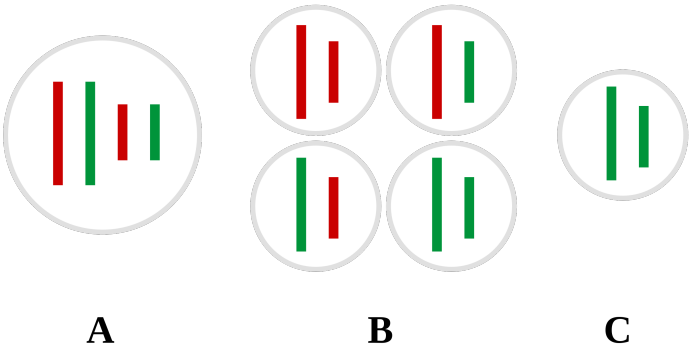
Reprezentare grafică a pașilor algoritmului *double-tree*



Sursa imaginii: Bo's Blog

Implementați algoritmul descris mai sus. Testați-l pe câteva exemple și convingeți-vă că soluțiile obținute au costul cel mult 2× costul minim găsit prin backtracking.

2. (Introducere în programarea genetică [15]). Programarea genetică este o metodă de a rezolva **probleme de optimizare** (de obicei cele computațional dificile/NP-hard, cum ar fi knapsack discret, travelling salesman etc.) folosind concepte inspirate din biologie (teoria evoluției). Fiecare soluție a unei probleme poate fi văzută ca un *individ* dintr-o populație și, împerechind și încrucișând soluțiile cele mai bune, putem „crește" soluții din ce în ce mai bune pentru problema noastră.



Sursa imaginii: Wikipedia

Ca să păstrăm lucrurile cât mai simple, în cazul nostru vom încerca să determinăm o valoare de intrare care să maximizeze valoarea de ieșire a unei funcții (presupuse necunoscută) [16]. Fiecare soluție va fi un *număr real*, reprezentat sub forma unui *șir de biți*. Lucrând pe biți, este mai ușor să „combinăm" între ele soluțiile, decât dacă am lucra direct cu niște numere reale. Fiecare bit din acest șir poartă denumirea de „genă".

Rezolvarea unei probleme folosind un algoritm genetic nu este dificilă, ci mai mult de muncă este la **implementarea funcționalităților de bază** și **modelarea problemei** folosind conceptele din programarea genetică.

Acest exercițiu este unul pregătitor, iar codul pe care îl dezvoltați poate fi refolosit într-un laborator viitor pentru a rezolva probleme de programare genetică.

1. Definiți o clasă care să reprezinte un *cromozom*. Constructorul va primi ca parametrii:
   - capetele intervalului $[a, b]$ în care se află numerele reale reprezentate;

- n, numărul de biți pe care îi are cromozomul.

și va aloca o listă/un vector de n valori boolene, reprezentând *șirul de biți* din cromozom, inițial toate `false`.

De asemenea, va salva într-o variabilă de instanță incrementul/precizia de reprezentare, care se poate calcula ca:

$$p = \frac{b - a}{2^n}$$

Hint: dacă nu mai țineți minte cum să definiți o clasă, puteți găsi pe internet tutoriale pentru Python [17], respectiv C++ [18].

2. Implementați o metodă pe clasa definită mai sus, care să transforme șirul de biți stocat în cromozom într-un număr real felul următor:
   (a) Transformă șirul de biți într-un număr întreg (`00…00` este 0, `00…01` este 1, `00…10` este 2, `00…11` este 3 etc.).
   (b) Înmulțește numărul întreg obținut cu incrementul, obținând astfel un număr real în intervalul $[0, b - a)$.
   (c) Adună capătul stâng al intervalului (adică a) la această valoare, obținând astfel un număr real în intervalul $[a, b)$.
3. Testați clasa voastră setând diferite valori pentru șirul de biți (puteți să îl modificați direct) și observând ce numere reale se obțin.

Referințe

[1] Wikipedia contributors, *Travelling salesman problem*, URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[2] GeeksforGeeks, *Proof that travelling salesman problem is NP-hard*, URL: https://www.geeksforgeeks.org/proof-that-traveling-salesman-problem-is-np-hard/.

[3] Jessica Wilkins, *Python Open File – How to Read a Text File Line by Line*, URL: https://www.freecodecamp.org/news/python-open-file-how-to-read-a-text-file-line-by-line/.

[4] Udacity Team, *How To Read From a File in C++*, URL: https://www.udacity.com/blog/2021/05/how-to-read-from-a-file-in-cpp.html.

[5] GeeksforGeeks, *Travelling Salesman Problem implementation using BackTracking*, URL: https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/.

[6] Wikipedia contributors, *Nearest neighbour algorithm*, URL: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.

[7] Jørgen Bang-Jensen, Gregory Gutin și Anders Yeo, „When the greedy algorithm fails", în *Discrete Optimization* 1.2 (2004), pp. 121–127, ISSN: 1572-5286, URL: https://www.sciencedirect.com/science/article/pii/S1572528604000222.

[8] Stefan Hougardy și Mirko Wilde, „On the nearest neighbor rule for the metric traveling salesman problem", în *Discrete Applied Mathematics* 195 (2015), 12th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW 2013), pp. 101–103, ISSN: 0166-218X, URL: https://www.sciencedirect.com/science/article/pii/S0166218X14001486.

[9] Bo Chang, „Traveling Salesman Problem and Approximation Algorithms", în *Bo's Blog* (10 Feb. 2019), URL: https://bochang.me/blog/posts/tsp/.

[10] Wikipedia contributors, *Minimum spanning tree*, URL: https://en.wikipedia.org/wiki/Minimum_spanning_tree.

[11] Wikipedia contributors, *Prim's algorithm*, URL: https://en.wikipedia.org/wiki/Prim's_algorithm.

[12] Wikipedia contributors, *Kruskal's algorithm*, URL: https://en.wikipedia.org/wiki/Kruskal's_algorithm.

[13] Wikipedia contributors, *Reverse-delete algorithm*, URL: https://en.wikipedia.org/wiki/Reverse-delete_algorithm.

[14] GeeksforGeeks, *Euler Tour of Tree*, URL: https://www.geeksforgeeks.org/euler-tour-tree/.

[15] Wikipedia contributors, *Genetic programming*, URL: https://en.wikipedia.org/wiki/Genetic_programming.

[16] Ahmed Gad, *Introduction to Optimization with Genetic Algorithm*, online, URL: https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b.

[17] David Amos, *Object-Oriented Programming (OOP) in Python 3*, URL: https://realpython.com/python3-object-oriented-programming/.

[18] GeeksforGeeks, *C++ Classes and Objects*, URL: https://www.geeksforgeeks.org/c-classes-and-objects/.