Algoritmi Avansați
Seminar 1

Gabriel Majeri

1 Introducere

Ce este o problemă (computațională)?

O **problemă computațională** [1] este o problemă la care putem determina răspunsul folosind un algoritm[1].

- **Exemplu de problemă computațională:** Care sunt factorii primi ai numărului natural n?
- **Exemplu de problemă care nu se poate rezolva folosind algoritmi:** Ce pereche de pantaloni să port astăzi?[2]

Ce tipuri de probleme computaționale există?

- **Probleme de decizie** [4]**:** Răspunsul este de tipul adevărat/fals (posibil/imposibil, există/nu există etc.)
- **Probleme de optimizare** [5]**:** Răspunsul este o *soluție* (un număr, un șir de numere, un text etc.) care este "cea mai bună" dintr-un anumit punct de vedere (consumă cele mai puține resurse, produce cel mai mare profit, acoperă cele mai multe noduri etc.) față de toate celelalte soluții posibile.

[1] Un *algoritm* [2] este o serie de pași bine-definiți care îți permit să obții un rezultat plecând de la niște date de intrare. În comparație, "să ghicești răspunsul corect" nu e tocmai un algoritm. Dar poate fi o euristică [3].

[2] Dacă ai un număr finit de opțiuni (pantaloni din care să alegi) și asociezi fiecărei perechi câte un *scor* (număr) care să indice dezirabilitatea acesteia, atunci această problemă se poate reduce la o problemă computațională, aceea de a găsi perechea de pantaloni cu scorul maxim.

2 Exerciții

Cunoștințe generale

1. Dați exemplu de **2 probleme** pe care le întâlniți în viața de zi cu zi care se pot interpreta ca probleme computaționale (e.g. cum alegi traseul pe care să vii la facultate).

Soluție: Există o mulțime de probleme care se pot rezolva algoritmic:

- Când te conectezi pe un website, la autentificare ți se poate cere adresa de e-mail și o parolă. Din motive de securitate, nu e bine ca o aplicație să păstreze parolele necriptate în baza de date [6]. Cel mai sigur este ca parola să fie trecută printr-un algoritm ca `bcrypt` [7], care îți permite să determini ulterior dacă utilizatorul a introdus parola corectă, dar fără să poți recupera parola originală.
- Vrei să faci niște cumpărături dar trebuie să treci pe la mai multe magazine, minimizând timpul pe care îl pierzi în deplasare. Alegerea optimă a ordinii în care să treci prin magazine se aseamănă cu problema comisului-voiajor [8].

□

2. Dați exemplu de **2 probleme de decizie** și **2 probleme de optimizare** pe care le-ați întâlnit în practică sau de care ați auzit.

Soluție: Probleme de decizie:

- Problema conectivității unui graf. De exemplu, am achiziționat și configurat câteva routere și vrem să știm dacă acum toate nodurile din rețeaua noastră pot comunica între ele.
- Determinarea dacă două grafuri sunt izomorfe. Chimiștii construiesc baze de date în care se află diferiți compuși chimici, iar ulterior vor să găsească și să compare substanțele, ignorând rotațiile sau simetriile structurii lor [9].

Probleme de optimizare:

- TEX, sistemul de tehnoredactare folosit la realizarea acestui material, își propune să aranjeze cuvintele pe linii și în paragrafe în așa fel încât să maximizeze aspectul estetic al paginii, utilizând cât mai puțin spațiu. Mai multe detalii se pot găsi în lucrarea lui Knuth și Plass din 1981 [10].

- Algoritmii de învățare automată, folosiți în inteligența artificială, sunt de fapt algoritmi care încearcă să minimizeze o funcție de cost.

Se pot găsi alte exemple pe Wikipedia [4, 5]. □

P versus NP

3. În cele ce urmează, să presupunem că avem **o listă** (un vector) **de** n **numere întregi**.

1. Propuneți un algoritm care să determine **cel mai mare element** (în **valoare absolută**) din listă. Ce complexitate de timp/memorie are algoritmul propus?

   Soluție: O soluție ar putea fi să citim toată lista în memorie și apoi să o parcurgem, reținând mereu valoarea maximă în modul:

   ```
   numbers = [int(x) for x in input().split()]

   max_n = numbers[0]
   for n in numbers[1:]:
       if abs(n) > abs(max_n):
           max_n = n

   print(max_n)
   ```

   Această metodă are complexitatea de timp $\mathcal{O}(n)$ (deoarece parcurgem toate numerele o singură dată), iar complexitatea de spațiu este $\mathcal{O}(n)$ (stocăm toate numerele într-o listă).

   O altă soluție, mai ineficientă ca timp, ar fi să ne folosim de funcția de sortare din Python, care are o complexitate de $\mathcal{O}(n \log n)$, și să comparăm capetele vectorului ordonat:

   ```
   numbers = [int(x) for x in input().split()]

   numbers.sort()

   if abs(numbers[0]) > abs(numbers[-1]):
       print(numbers[0])
   else:
       print(numbers[-1])
   ```

Dacă avem o cale de a citi șirul de numere element cu element (de exemplu, este stocat într-un fișier sau vine de pe rețea) putem reduce consumul de memorie:

```
max_n = 0

for line in open('numere.txt', 'r'):
    n = int(line)
    if abs(n) > abs(max_n):
        max_n = n

print(max_n)
```

În acest caz, memoria consumată este $\mathcal{O}(1)$. Totuși, fișierul consumă $\mathcal{O}(n)$ spațiu pe disc. □

2. Pentru un număr întreg dat S, propuneți un algoritm care să determine **toate perechile de numere** din lista inițială **care adunate dau** S. Puteți găsi un algoritm cu complexitate de timp, în cel mai rău caz, mai bună decât $\mathcal{O}(n^2)$?

Soluție: O soluție naivă este să parcurgem lista de numere cu două `for`-uri imbricate:

```
S = int(input())
numbers = [int(x) for x in input().split()]

for i in range(len(numbers) - 1):
    for j in range(i + 1, len(numbers)):
        if numbers[i] + numbers[j] == S:
            print(i, j)
```

Bucla interioară parcurge un subșir de $n-1$ numere, apoi de $n-2$ numere, ..., 2, 1. Deci numărul total de pași vine

$$1+2+3+\cdots+(n-2)+(n-1)=\frac{(n-1)(n-2)}{2}\in\mathcal{O}(n^2).$$

Memoria folosită se încadrează în $\mathcal{O}(n)$.

O soluție mai eficientă ar fi să parcurgem o dată vectorul și să salvăm într-un dicționar valoarea care, adunată cu elementul de pe poziția i, ne-ar da S:

```
S = int(input())
numbers = [int(x) for x in input().split()]

diff = { (S - n) : i for (i, n) in enumerate(numbers) }

for j, m in enumerate(numbers):
    if m in diff and j != diff[m]:
        print(i, j)
```

O problemă cu această soluție este că, dacă avem mai multe numere cu aceeași valoare în lista inițială, nu mai afișăm toate combinațiile posibile care generează suma S. Dar dacă ignorăm acest caz, soluția are complexitate de timp $\mathcal{O}(n)$. □

3. (3SUM [11]) Propuneți un algoritm care să determine dacă în lista inițială există **trei numere** care **adunate să aibă suma 0**. Credeți că puteți un algoritm determinist care să rezolve problema în mai puțin de n^2 pași? Dar dacă ați aborda problema în mod nedeterminist?

Soluție: Soluția deterministă care rezolvă problema în $\mathcal{O}(n^2)$ pași este:

```
numbers = [int(x) for x in input().split()]

diff = { (0 - n) : i for (i, n) in enumerate(numbers) }

for j in range(len(numbers) - 1):
    m = numbers[j]
    for k in range(j + 1, len(numbers)):
        p = numbers[k]
        rem = 0 - m - p
        if rem in diff and j != diff[rem] and k != diff[rem]:
            print(i, j, k)
            exit(0)

print('Nu există')
```

Dacă există sau nu vreun algoritm mai eficient de n^2 pentru a rezolva această problemă este o întrebare încă deschisă în informatică [11].

Pentru rezolvarea nedeterministă, să observăm că, dacă ni se spune că elementele de pe pozițiile i, j și k sunt o soluție, putem verifica asta

prin

$$numbers[i] + numbers[j] + numbers[k] == 0$$

care este timp $\mathcal{O}(1)$ în raport cu lungimea vectorului. Un algoritm nedeterminist ar ști cum să „aleagă" din prima indicii potriviți, iar verificarea s-ar face în timp polinomial (chiar constant), deci este o problemă NP (*nondeterministic polynomial time*). □

Metoda greedy

Mai multe informații despre principiile care stau la baza metodei greedy se pot găsi la [12].

4. Suntem administratorii unui spital și vrem să ne asigurăm că avem tot timpul cel puțin un medic prezent la camera de gardă într-un anumit interval de timp. Presupunem că avem la dispoziție n medici, care ar fi dispuși să stea de gardă fiecare între anumite ore. Vrem să asignăm câți mai puțini medici, ca ceilalți să se poată ocupa de alte urgențe.

Problema formalizată: Find dat un interval $[a, b]$ și o mulțime de intervale $[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]$, vrem să alegem un număr *minim* dintre acestea astfel încât, reunite, să includă intervalul inițial $[a, b]$.

1. Aceasta este o problemă de **decizie** sau de **optimizare**? Dar dacă ne-ar interesa doar să vedem dacă putem acoperi intervalul dat cu intervalele disponibile?

   Soluție: Așa cum a fost dată, problema este una de optimizare; ne interesează să găsim o soluție care să acopere intervalul dat alegând *cât mai puține* intervale.

   Dacă ne interesează doar dacă se poate acoperi intervalul țintă cu intervale, atunci problema este una de decizie. □

2. Propuneți un algoritm care **să determine dacă** măcar putem acoperi intervalul dat cu intervalele disponibile.

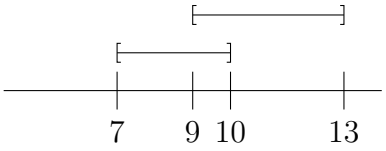
   Soluție: Începem prin a defini un mod de reprezentare al datelor de intrare ale problemei. Fiecare interval poate fi interpretat ca o pereche de numere reale $(start, end)$. Intervalul inițial îl vom nota cu (a, b), iar celelalte intervale vor fi stocate într-o listă de perechi:

```python
# Citim intervalul țintă
a, b = [float(x) for x in input().split()]

# „n" este numărul de intervale pe care le vom citi
n = int(input())

intervals = []
for i in range(n):
    start, end = [float(x) for x in input().split()]
    intervals.append((start, end))
```

Acum, să ne gândim ce fel de operații putem face cu niște intervale. Intervalele sunt de fapt **mulțimi** de numere reale, deci putem aplica operațiile uzuale de reuniune, intersecție, diferență, verificăm incluziuni etc. Nu vom avea nevoie de toate acestea ca să rezolvăm problema, dar să ne uităm peste cum le-am putea implementa:

∩ Pentru a calcula intersecția intervalelor $[a, b]$ și $[c, d]$, avem de luat în considerare două cazuri:

- Intervalele sunt disjuncte (adică $b < c$ sau $d < a$), deci intersecția este mulțimea vidă $\emptyset$.
- Intervalele se suprapun (măcar parțial) iar intersecția este nevidă. Dacă nu suntem în cazul de mai sus, intersecția intervalelor este un nou interval cu capetele $((\max(a, c), \min(b, d))$.



∪ Și pentru a calcula reuniunea intervalelor $[a, b]$ și $[b, c]$ avem de luat în calcul tot două cazuri:

- Dacă intervalele sunt disjuncte (putem verifica asta ca mai sus), atunci reuniunea lor nu mai este un interval; nu ne prea ajută acest caz, pentru că nu vrem să complicăm modul în care reprezentăm intervalele.
- Dacă intervalele nu sunt disjuncte, atunci reuniunea lor este un nou interval cu capetele $(\min(a, c), \max(b, d))$.

⊆ Pentru a verifica incluziunea intervalului (a, b) în intervalul c, d, putem pur și simplu să vedem dacă $c \leq a$ și $b \leq d$.

Aceste operații ne duc cu gândul la un mod de a rezolva problema. Dacă există o submulțime a intervalelor care, reunite, acoperă intervalul inițial, atunci sigur luând toate intervalele putem acoperi întreg intervalul țintă. Deci, o idee ar fi să construim $U = [a_1, b_1] \cup [a_2, b_2] \cup \cdots \cup [a_n, b_n]$ și să vedem dacă $[a, b] \subseteq U$.

Mai mult de atât, nici nu e nevoie să luăm *toate* intervalele la rând. Putem să ne uităm doar la cele care se intersectează cu $[a, b]$, sau cu alte cuvinte, să calculăm intersecțiile $[a, b] \cap [a_1, b_1], [a, b] \cap [a_2, b_2], \ldots$ și să le reunim pe acestea.

Ca să ne fie mai ușor să calculăm reuniunile, am vrea să nu tratăm și cazul în care unele intervale sunt disjuncte. De fapt, dacă rămânem cu o „groapă" între intervalele noastre, înseamnă că nu putem acoperi întreg intervalul țintă. În acest scop, vom **sorta** intervalele crescător după capătul din stânga.

```
# Sortarea va compara perechile de numere folosind
# ordinea lexicografică, deci va ordona mai întâi
# după capătul din stânga
intervals.sort()

i = 0

# Ignorăm intervalele care nu ne ajută pentru că
# oricum sunt la stânga intervalului țintă
while i < len(intervals) and intervals[i][1] < start:
    i += 1

# Toate intervalele sunt la stânga celui țintă
if i == len(intervals):
    print('NU')
    exit()

left, right = intervals[i]
for interval in intervals[i + 1:]:
    # Dacă nu putem reuni următorul interval,
    # înseamnă că există o „groapă" pe care
    # nu o putem acoperi
    if interval[0] > right:
        break
```

```
        # Încerc să reunesc noul interval
        right = max(right, interval[1])

        # Dacă deja am acoperit intervalul [a, b]...
        if right >= b:
            # ...nu mai continuăm parcurgerea
            break

# Verificăm dacă am reușit
# să cuprindem intervalul țintă
if left <= a and b <= right:
    print('DA')
else:
    print('NU')
```

□

3. Propuneți un algoritm care să rezolve problema inițială. **Demonstrați** că algoritmul propus obține soluția optimă (adică că nu poate exista o altă soluție, diferită de cea obținută prin metoda voastră, care să folosească mai puține intervale).

   Soluție: Pornind de la algoritmul implementat anterior, tot ce trebuie să modificăm pentru a obține în toate cazurile o soluție optimă este să reunim la fiecare pas intervalul „cel mai din dreapta" care poate fi folosit să extindă soluția curentă. O implementare în Python a acestei metode se poate găsi aici.

   Pentru a demonstra că în acest caz rezultatul este optim, putem să ne folosim de metoda reducerii la absurd. Dacă presupunem că soluția obținută de noi, notată SOL_{Greedy}, nu este optimă, înseamnă că există o altă mulțime de intervale SOL_{Opt} care acoperă intervalul inițial, cu $|SOL_{Opt}| < |U_{Greedy}|$.

   Putem presupune că cele două mulțimi coincid până la intervalul cu indicele k, iar apoi diferă:

$$SOL_{Greedy} = I_1, I_2, \dots, I_k, I_{k+1}, \dots, I_m$$
$$SOL_{Opt} = I_1, I_2, \dots, I_k, I_{k'+1}, \dots, I_{m'}$$

   unde $m' < m$.

Datorită modului în care am sortat și ales intervalele, știm că I_{k+1} este cu siguranță intervalul care are capătul din dreapta cel mai mare dintre intervalele compatibile cu soluția curentă. În particular, I_{k+1} se termină mai la dreapta față de (sau în același punct cu) $I_{k'+1}$.

Dacă ar fi să îl înlocuim pe $I_{k'+1}$ cu I_{k+1} în SOL_{Opt}, atunci soluția ar rămâne la fel (ca număr de intervale), sau cel mult s-ar putea să se reducă numărul de intervale necesare ca să fie o soluție validă (dar asta ar contrazice optimalitatea ei!). Tot aplicând aceste înlocuiri, am obține că soluția optimă are de fapt același număr de intervale ca soluția greedy, o contradicție cu $|SOL_{Opt}| < |U_{Greedy}|$. □

Referințe

[1] Wikipedia contributors, *Computational problem*, URL: https://en.wikipedia.org/wiki/Computational_problem.

[2] Wikipedia contributors, *Algorithm*, URL: https://en.wikipedia.org/wiki/Algorithm.

[3] Wikipedia contributors, *Heuristic*, URL: https://en.wikipedia.org/wiki/Heuristic_(computer_science).

[4] Wikipedia contributors, *Decision problem*, URL: https://en.wikipedia.org/wiki/Decision_problem.

[5] Wikipedia contributors, *Optimization problem*, URL: https://en.wikipedia.org/wiki/Optimization_problem.

[6] Information Security Stack Exchange Contributors, *Why shouldn't I store passwords in plaintext?*, 15 Apr. 2016, URL: https://security.stackexchange.com/a/120544.

[7] Wikipedia contributors, *bcrypt*, URL: https://en.wikipedia.org/wiki/Bcrypt.

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

[9] Christophe-André Mario Irniger, *Graph matching: filtering databases of graphs using machine learning techniques*, Berlin: AKA, 2005, ISBN: 1-58603-557-6.

[10] Donald E. Knuth și Michael F. Plass, „Breaking paragraphs into lines", în *Software: Practice and Experience* 11.11 (1981), pp. 1119–1184, URL: http://www.eprg.org/G53DOC/pdfs/knuth-plass-breaking.pdf.

[11] Wikipedia contributors, *3SUM*, URL: https://en.wikipedia.org/wiki/3SUM.

[12] Karleigh Moore, Jimin Khim și Eli Ross, *Greedy Algorithm*, URL: https://brilliant.org/wiki/greedy-algorithm/.

Algoritmi Avansați
Seminar 2

Gabriel Majeri

1 Introducere

În seminarul trecut am discutat principalele tipuri de probleme computaționale și am văzut că, pentru unele probleme, putem calcula în mod eficient soluția optimă. În acest seminar vom investiga câteva probleme dificile din punct de vedere computațional, pe care nu le putem rezolva în mod eficient și pentru care putem cel mult să obținem rapid o soluție **aproximativă** [1].

2 Exerciții

1. (Vertex cover [2]) Problema acoperirii unui graf (neorientat) ne cere să găsim o **submulțime de noduri din graf** astfel încât orice muchie din graf să fie învecinată cu (cel puțin) un nod din această mulțime. Problema de optimizare asociată implică găsimea mulțimii de cardinal **minim**.

În cele ce urmează, vom presupune că avem un graf cu n și m muchii, reprezentat prin liste de adiacență.

1. Fiind dată o mulțime de noduri dintr-un graf, cum ați **verifica** că mulțimea dată este o acoperire? Ce complexitate de timp are soluția propusă?

   Soluție: Dacă graful este reprezentat sub forma unor liste de adiacență, putem parcurge mulțimea de noduri date ca „soluție”, iar pentru fiecare nod îi ștergem vecinii din lista de adiacență (și pe el din listele vecinilor lui). La final, verificăm dacă toate listele de adiacență sunt vide.

   Complexitatea de timp este cel mult $\mathcal{O}(n^2)$, deoarece soluția poate avea cel mult n noduri, iar fiecare nod are cel mult $n-1$ vecini (am presupus

că „listele” de adiacență sunt de fapt stocate ca *set*-uri, deci ștergerile se fac în timp $\mathcal{O}(1)$). □

2. Propuneți o soluție de tip brute-force care să rezolve problema (i.e. să verifice toate submulțimile posibile de noduri). Ce complexitate de timp are soluția?

   Soluție: Putem genera toate submulțimile posibile de noduri din graf (exceptând-o pe cea vidă), urmând apoi să le verificăm cu algoritmul descris mai sus. Această soluție are o complexitate de timp $\mathcal{O}(2^n)$ (timp *exponențial*). □

3. Să presupunem că alegeți o muchie aleatoare din graf, cu capetele u și v. Având în vedere că într-o soluție (optimă) a problemei *vertex cover* trebuie „atinse” toate muchiile, ce puteți spune despre nodurile u și v în raport cu mulțimea de noduri care constituie soluția (optimă)?

   Soluție: Cel puțin unul dintre u sau v va fi în soluția finală. Altfel, nu ar fi acoperită muchia uv. □

4. Ne gândim la următorul algoritm pentru a construi o soluție la problema acoperirii:

   (a) Inițial, acoperirea noastră este mulțimea vidă $\emptyset$.

   (b) Cât timp mai sunt muchii neacoperite în graf:

      i. Luăm (aleator) orice muchie din graf.

      ii. Includem capetele muchiei în mulțimea care va reprezenta acoperirea noastră.

      iii. Ștergem toate muchiile incidente la cele două noduri menționate.

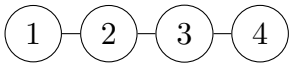
   (c) Mulțimea de noduri construită este soluția la problema noastră.

   Acest algoritm construiește o soluție *aproximativă* la problema de optim pe care vrem să o rezolvăm; în unele cazuri, s-ar putea să fie posibil să construim o acoperire folosind mai puține noduri.

   Determinați factorul de aproximare al acestui algoritm, dați un exemplu când algoritmul obține soluția optimă și unul în care obține soluția cea mai proastă în raport cu soluția optimă.

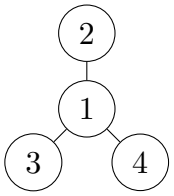
Soluție: Algoritmul prezentat este 2-aproximativ.

Un exemplu în care algoritmul obține soluția optimă este un graf liniar cu 4 noduri:



Dacă algoritmul alege muchia dintre 2 și 3, o să se oprească imediat, găsind soluția optimă formată din două noduri.

Un exemplu în care algoritmul obține o soluție de două ori mai proastă decât cea optimă este următorul:



Orice muchie ar alege, algoritmul construiește o soluție formată din două noduri, deși ar putea acoperi tot graful alegând unul singur (pe cel central). □

2. (Maximum cut [3]) Fiind dat un graf neorientat (V, E), problema tăieturii maxime ne cere să găsim o **submulțime de noduri din graf**, astfel încât numărul de muchii care o leagă pe aceasta de complementul ei să fie maxim. Cu alte cuvinte, vrem două mulțimi de noduri A și B, cu $A \cup B = V$ și $A = V \setminus B$.

Această probleme are aplicații în fizica statistică și în proiectarea de circuite electronice [4].

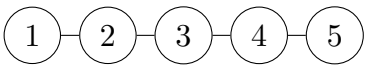
Un algoritm aproximativ care rezolvă problema, descris în [5], este următorul:

1. Alegem aleator două noduri diferite din graf, v_1 și v_2.
2. Inițializăm $A := \{ v_1 \}$, $B := \{ v_2 \}$.
3. Cât timp mai sunt noduri care nu se află nici în A, nici în B:
   (a) Luăm aleator un astfel de nod.
   (b) Dacă are mai multe muchii care duc către A decât muchii care duc către B, îl punem în B; altfel, în A.
4. Soluția finală sunt mulțimile A și B.

Determinați ce **complexitate de timp** are algoritmul propus, ce **factor de aproximare** oferă și dați un **exemplu** în care soluția aproximativă găsită de algoritm este cea mai îndepărtată de soluția optimă.

Soluție: Complexitatea de timp a algoritmului dat este $\mathcal{O}(n^2)$ (dacă reținem listele de adiacență ca niște *set*-uri, putem calcula foarte eficient intersecția dintre vecinii unui nod și cele două mulțimi A și B).

Algoritmul descris este $\frac{1}{2}$-aproximativ. Un exemplu în care soluția găsită este de două ori mai proastă decât cea optimă poate fi:



Dacă algoritmul alege la început să-l facă pe nodul 1 să aparțină lui A și pe 3 lui B, iar apoi continuă aleator cu 5 (pe care îl pune în A), atunci și 2 și 4 vor ajunge tot în A, iar tăietura va cuprinde doar două muchii: 2–3 și 3–4. În schimb, soluția optimă ar fi ca nodurile să alterneze de la stânga la dreapta, obținând astfel patru muchii în tăietură. □

Referințe

[1] Wikipedia contributors, *Approximation algorithm*, URL: https://en.wikipedia.org/wiki/Approximation_algorithm.

[2] Wikipedia contributors, *Vertex cover*, URL: https://en.wikipedia.org/wiki/Vertex_cover.

[3] Wikipedia contributors, *Maximum cut*, URL: https://en.wikipedia.org/wiki/Maximum_cut.

[4] Francisco Barahona et al., „An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design", în *Operations Research* 36.3 (1988), pp. 493–513, URL: http://www.jstor.org/stable/170992.

[5] Vijay V. Vazirani, *Approximation algorithms*, Springer, 2001, ISBN: 978-3642084690.

Algoritmi Avansați
Seminar 3

Gabriel Majeri

1 Introducere

În acest seminar vom discuta câteva tipuri de algoritmi care nu se încadrează în tiparele obișnuite de rezolvare ale problemelor, dar care pot fi extrem de utili și versatili în unele situații.

2 Algoritmi probabiliști

O metodă de a rezolva mai eficient unele probleme dificile din punct de vedere computațional este să ne folosim de *șansă*. Astfel obținem algoritmi „probabiliști" / „randomizați" [1].

Un scurt istoric

În cadrul Proiectului Manhattan, cercetătorii Stanislaw Ulam și John von Neumann aveau nevoie să rezolve niște probleme foarte complicate legate de difuzia neutronilor. Așa că au venit cu ideea să modeleze pe calculator o simulare a reacțiilor care au loc în nucleul unei arme de fisiune nucleară, ca să înțeleagă ce se întâmplă fără să mai fie nevoie de un experiment fizic. Astfel a fost creată și folosită pentru prima dată *metoda Monte Carlo* [2, 3]: modelarea pe calculator a unei probleme teoretice pentru a o rezolva numeric.

Generarea numerelor aleatoare

Primul pas în implementarea unui algoritm probabilist este să avem acces la o sursă de numere aleatoare [4].

Calculatoarele digitale obișnuite sunt mașinării *deterministe*, care nu pot produce numere cu adevărat aleatoare. Un calculator poate folosi un **generator de numere pseudo-aleatoare** (PRNG [5]) pentru a genera un șir de

numere care aparent nu au nicio legătură unul cu celălalt, plecând de la un număr dat (denumit *seed*); dar dacă numărul inițial este dezvăluit, se pot determina toate numerele care urmează. Din acest motiv, dacă aveți nevoie să generați valori aleatoare într-un context sensibil (e.g. în criptografie), recomandarea este să utilizați un **generator de numerele pseudo-aleatoare sigur din punct de vedere criptografic** (CSPRNG [6]).

2.1 Algoritmi Monte Carlo

Un algoritm de tip Monte Carlo se execută pentru un anumit număr de pași și ne oferă o soluție care este doar *probabil* corectă; cu cât lăsăm algoritmul să ruleze mai mult, cu atât ne apropiem de soluția optimă sau o aproximăm mai bine [7].

Un exemplu de astfel de algoritm este un algoritm bazat pe programarea genetică; cu cât lăsăm să ruleze mai mult programul, cu atât soluțiile obținute sunt din ce în ce mai bune.

2.2 Algoritmi Las Vegas

Un algoritm de tip Las Vegas s-ar putea să nu obțină o soluție validă după o execuție, și să fie nevoie să-l rulăm de mai multe ori. Dar în momentul în care a obținut o soluție, aceasta este sigur cea corectă [8].

Un exemplu de algoritm de acest tip este cel care rezolvă problema așezării a 8 regine pe tabla de șah alegând aleator unde să pună o regină pe următorul rând; există riscul ca o soluție pe care a început să o construiască să nu fie corectă și să fie obligat să o ia de la capăt, dar în momentul în care a ajuns la final, soluția este sigur bună.

Vedere de ansamblu

Tabelul de mai jos compară cele două familii principale de algoritmi probabiliști, arătând cum una este „duala” celeilalte:

	Timp de execuție	Corectitudine rezultat
Monte Carlo	Determinist	Probabilistă
Las Vegas	Probabilist	Deterministă

Exerciții

1. Dacă aveți la dispoziție o funcție `random()` care generează un număr real în intervalul $[0, 1]$ (cu o distribuție uniformă), cum puteți obține un număr real în intervalul $[a, b]$ (cu o distribuție uniformă pe $[a, b]$)?

Soluție: Fie x un număr din intervalul $[0, 1]$, pe care l-am obținut prin funcția `random()`. Înmulțindu-l cu $b - a$, obținem un nou număr $(b - a)x$, care se află în intervalul $[0, b - a]$. Adunând valoarea lui a la acesta, obținem $a + (b - a)x$, care este în intervalul $[a, b]$.

Această formulă este utilă în limbajele care nu oferă în biblioteca standard o funcție pentru a genera un număr aleator dintr-un interval arbitrar $[a, b]$, ci doar din $[0, 1]$, cum se întâmplă în cazul limbajului JavaScript. □

2. Dacă aveți la dispoziție o funcție `biased_random()` care generează aleator 0 sau 1, dar cu o probabilitate inegală p ($\neq 50\%$), puteți să definiți o funcție care să vă genereze 0 sau 1 cu probabilitate 50%–50%?

Soluție: Da, putem construi o astfel de funcție.

Observația de la care plecăm este că, dacă apelăm de două ori funcția `biased_random()`, singurele valori pe care le putem obține 00, 01, 10 sau 11. Calculând probabilitatea pentru fiecare dintre aceste situații, obținem

X	00	01	10	11
$\mathbb{P}(X)$	$(1-p)^2$	$p(1-p)$	$(1-p)p$	p^2

unde am notat cu p probabilitatea ca un apel al funcției `biased_random()` să returneze valoarea 1. Există o simetrie în tabel: $p(1-p) = (1-p)p$; deci $\mathbb{P}(01) = \mathbb{P}(10)$.

```
def unbiased_random():
    while True:
        x = biased_random()
        y = biased_random()

        # Suntem în cazul 00 sau 11, care nu ne ajută
        if x == y:
            continue
        else:
            if x == 0:
                # Cazul 01
                return 0
            else:
                # Cazul 10
                return 1
```

Remarcăm că această implementare este un *algoritm probabilist de tip Las Vegas*; rezultatul obținut respetă întotdeauna cerința (este 0 sau 1 cu

probabilitate 50%–50%), dar nu putem spune de dinainte de câte ori trebuie să se execute blocul `while` înainte de a obține un rezultat valid. □

3. În acest exercițiu ne propunem să construim un algoritm probabilist care să ne ajute să determinăm valoarea lui π.

1. Care este aria cercului unitate?
2. Care este aria pătratului definit de punctele $(-1,-1)$, $(1,-1)$, $(1,1)$ și $(-1,1)$?
3. Dacă aleg aleator un punct în pătratul descris mai sus, care este probabilitatea ca el să fie în interiorul cercului unitate?
4. Dacă am un punct $(x,y) \in \mathbb{R}^2$, cum pot verifica dacă acesta se află în interiorul cercului unitate?
5. Avem următorul algoritm: luăm aleator perechi de numere din $[-1,1]$ (adică puncte din pătratul descris mai sus). Dacă punctul descris de această pereche de numere este în interiorul cercului unitate îl contorizăm, altfel îl ignorăm. La final, împărțim numărul de puncte care au căzut în interiorul cercului la numărul total de puncte luate în considerare. Ce valoare aproximează acest raport?
6. Ce fel de algoritm este cel descris anterior? Monte Carlo sau Las Vegas?

Soluție:

1. Aria cercului unitate $A_\bigcirc = \pi \cdot 1^2 = \pi$.
2. Laturile definite de acele puncte au toate lungimea $|1-(-1)| = 2$, deci pătratul corespunzător are aria $A_\square = 2^2 = 4$.
3. Având în vedere că cercul unitate este conținut în pătratul descris anterior, probabilitatea în acest caz va fi raportul dintre aria cercului și aria pătratului, $\mathbb{P}(X) = A_\bigcirc / A_\square = \pi/4$.
4. Cel mai ușor este să folosim formula pentru distanța Euclidiană față de origine, $d = \sqrt{x^2+y^2}$, și să verificăm că $d \leq 1$.
5. Acest raport aproximează probabilitatea ca un punct aleator din pătratul descris anterior să fie în cercul unitate. Cu alte cuvinte, dacă n este numărul de puncte din eșantion, avem că

$$\lim_{n\to+\infty} \frac{\text{nr. puncte în cercul unitate}}{\text{nr. total de puncte}} = \frac{A_\bigcirc}{A_\square} = \frac{\pi}{4}.$$

Dacă înmulțim rezultatul cu 4, avem o metodă probabilistă de a calcula numeric valoarea lui π.

6. Algoritmul este unul de tip Monte Carlo. Cu cât rulăm pentru mai mult timp algoritmul și eșantionăm mai multe puncte, cu atât crește precizia rezultatului nostru.

□

3 Programare liniară și algoritmul simplex

Unele dintre cele mai simple tipuri de probleme de optimizare (și cele mai des întâlnite în practică) sunt cele de *programare liniară*[1]. Din fericire, pentru acestea avem o interpretare vizuală destul de clară (cel puțin, în cazul 2D) și un algoritm eficient de rezolvare (metoda simplex).

Pentru a înțelege cum arată aceste probleme și cum le putem rezolva, vă recomand să vă uitați la [9], la capitolul 13 din [10] (notele de curs sunt disponibile aici), la această lecție [11] dintr-un curs de la MIT sau la acest video [12].

[1] În acest context, la fel ca în cazul „programării dinamice”, termenul de „programare” se referă la o metodă tabelară de rezolvare a unei probleme, nu neapărat la programare/dezvoltare software.

Referințe

[1] David Rydeheard, *Probabilistic (Randomized) algorithms*, URL: http://www.cs.man.ac.uk/~david/courses/advalgorithms/probabilistic.pdf.

[2] Roger Eckhardt, „Stan Ulam, John von Neumann, and the Monte Carlo method", în *Los Alamos Science* (15 1987), pp. 131–137, URL: https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068.

[3] Wikipedia contributors, *Monte Carlo method*, URL: https://en.wikipedia.org/wiki/Monte_Carlo_method.

[4] Wikipedia contributors, *Random number generation*, URL: https://en.wikipedia.org/wiki/Random_number_generation.

[5] Wikipedia contributors, *Pseudorandom number generator*, URL: https://en.wikipedia.org/wiki/Pseudorandom_number_generator.

[6] Wikipedia contributors, *Cryptographically-secure pseudorandom number generator*, URL: https://en.wikipedia.org/wiki/Cryptographically-secure_pseudorandom_number_generator.

[7] Wikipedia contributors, *Monte Carlo algorithm*, URL: https://en.wikipedia.org/wiki/Monte_Carlo_algorithm.

[8] Wikipedia contributors, *Las Vegas algorithm*, URL: https://en.wikipedia.org/wiki/Las_Vegas_algorithm.

[9] Trefor Bazett, *Intro to Linear Programming and the Simplex Method*, URL: https://www.youtube.com/watch?v=K7TL5NMlKIk.

[10] Georgia Tech, *Computability, Complexity & Algorithms*, URL: https://www.udacity.com/course/computability-complexity-algorithms--ud061.

[11] MIT OpenCourseWare, *15. Linear Programming: LP, reductions, Simplex*, URL: https://www.youtube.com/watch?v=WwMz2fJwUCg.

[12] The Organic Chemistry Tutor, *Linear Programming*, URL: https://www.youtube.com/watch?v=Bzzqx1F23a8.

Algoritmi avansaţi

Seminar 5 (săpt. 9 şi 10)

1. *Fie punctele* $A = (1, 2, 3), B = (4, 5, 6) \in \mathbb{R}^3$.

a) *Fie* $C = (a, 7, 8)$. *Arătaţi că există* a *astfel ca punctele* A, B, C *să fie coliniare şi pentru* a *astfel determinat calculaţi raportul* $r(A, B, C)$.

b) *Determinaţi punctul* P *astfel ca raportul* $r(A, P, B) = 1$.

c) *Daţi exemplu de punct* Q *astfel ca* $r(A, B, Q) < 0$ *şi* $r(A, Q, B) < 0$.

Soluţie.

a) Condiţia de coliniaritate a punctelor A, B, C este echivalentă cu coliniaritatea vectorilor $\overrightarrow{AB}$ şi $\overrightarrow{BC}$. Au loc relaţiile:

$$\overrightarrow{AB} = B - A = (3, 3, 3), \quad \overrightarrow{BC} = (a - 4, 2, 2).$$

Vectorii daţi sunt proporţionali dacă şi numai dacă $a - 4 = 2$, deci $a = 6$. De fapt, dreapta AB este direcţionată de vectorul $(1, 1, 1)$ (şi de orice vector proporţional cu acesta).

În acest caz, avem $\overrightarrow{AB} = B - A = (3, 3, 3), \quad \overrightarrow{BC} = (2, 2, 2)$, deci

$$\overrightarrow{AB} = \frac{3}{2}\,\overrightarrow{BC},$$

adică $r(A, B, C) = \frac{3}{2}$ (raportul $r(A, B, C)$ este acel scalar r pentru care are loc relaţia $\overrightarrow{AB} = r\,\overrightarrow{BC}$).

b) Condiţia $r(A, P, B) = 1$ este echivalentă cu $\overrightarrow{AP} = \overrightarrow{PB}$. Punctul P care verifică această condiţie este mijlocul segmentului $[AB]$, deci $P = \frac{1}{2}A + \frac{1}{2}B = (\frac{5}{2}, \frac{7}{2}, \frac{9}{2})$.

c) Semnele rapoartelor indică faptul că (i) B nu este între A şi Q; (ii) Q nu este între A şi B. Trebuie deci ca A să fie situat între Q şi B. Un astfel de punct este $Q = (0, 1, 2)$ (l-am ales ca fiind $A - (1, 1, 1)$). Au loc relaţiile

$$\overrightarrow{AB} = (3, 3, 3), \quad \overrightarrow{BQ} = (-4, -4, -4), \quad r(A, B, Q) = -\frac{3}{4},$$

$$\overrightarrow{AQ} = (-1, -1, -1), \quad \overrightarrow{QB} = (4, 4, 4), \quad r(A, Q, B) = -\frac{1}{4},$$

deci sunt verificate cerinţele din enunţ.

2. *Fie punctele* $P = (1, -1), Q = (3, 3)$.

a) *Calculaţi valoarea determinantului care apare în testul de orientare pentru muchia orientată* $\overrightarrow{PQ}$ *şi punctul de testare* $O = (0, 0)$.

b) *Fie* $R_\alpha = (\alpha, -\alpha)$, *unde* $\alpha \in \mathbb{R}$. *Determinaţi valorile lui* α *pentru care punctul* R_α *este situat în dreapta muchiei orientate* $\overrightarrow{PQ}$.

Soluţie.

a) Conform teoriei,

$$\Delta(P, Q, R) = \begin{vmatrix} 1 & 1 & 1 \\ p_1 & q_1 & r_1 \\ p_2 & q_2 & r_2 \end{vmatrix}.$$

În exemplu avem:

$$\Delta(P, Q, R) = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 3 & 0 \\ -1 & 3 & 0 \end{vmatrix} = 6 \text{ (dezvoltare după ultima coloană).}$$

Se poate verifica şi pe un desen că O este la stânga muchiei orientate $\overrightarrow{PQ}$.

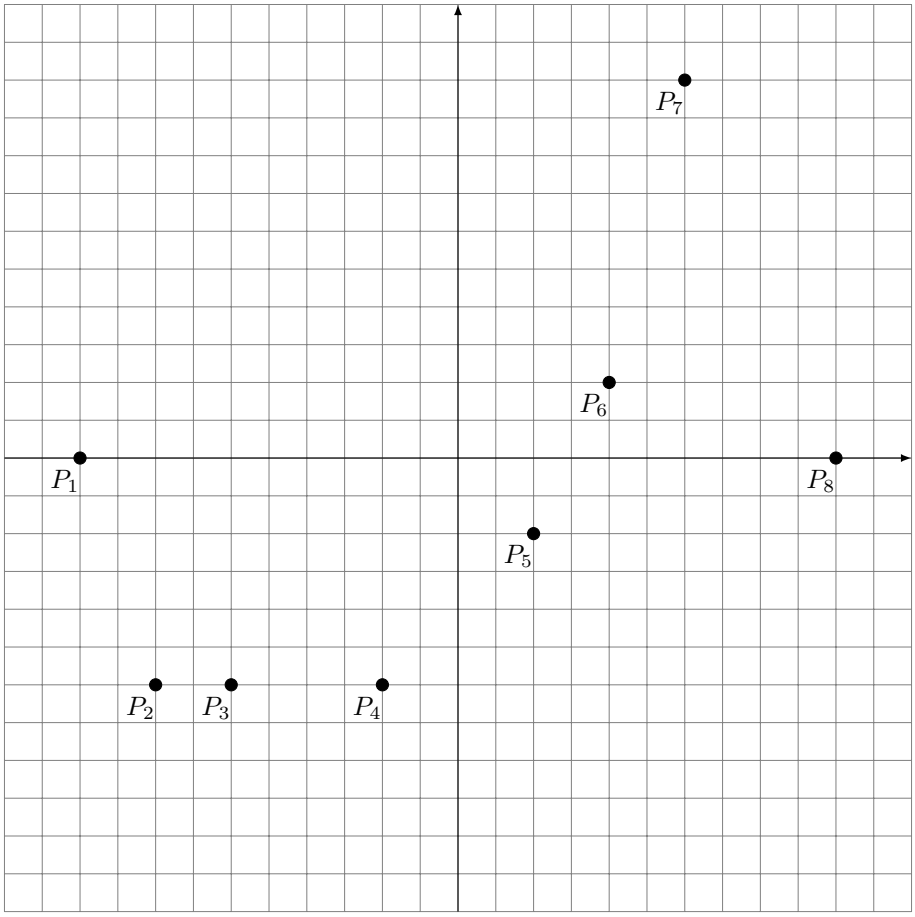
b) Calculăm, pentru un α, valoarea $\Delta(P, Q, R_\alpha)$:

$$\Delta(P, Q, R_\alpha) = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 3 & \alpha \\ -1 & 3 & -\alpha \end{vmatrix} = 6(1 - \alpha).$$

Punctul R_α este situat în dreapta muchiei orientate $\overrightarrow{PQ} \Leftrightarrow \Delta(P, Q, R_\alpha) < 0 \Leftrightarrow \alpha > 1$. Acest lucru poate fi verificat şi pe desen, punctul R_α este variabil pe cea de-a doua bisectoare (de ecuaţie $x + y = 0$), iar pentru $\alpha > 1$ acest punct este situat în dreapta muchiei orientate $\overrightarrow{PQ}$.

3. *Fie* $\mathcal{M} = \{P_1, P_2, \ldots, P_9\}$, *unde* $P_1 = (-5, 0)$, $P_2 = (-4, -3)$, $P_3 = (-3, -3)$, $P_4 = (-1, -3)$, $P_5 = (1, -1)$, $P_6 = (2, 1)$, $P_7 = (3, 5)$, $P_8 = (5, 0)$. *Detaliaţi cum evoluează lista* $\mathcal{L}_i$ *a vârfurilor care determină marginea inferioară a frontierei acoperirii convexe a lui* $\mathcal{M}$, *obţinută pe parcursul Graham's scan, varianta Andrew.*

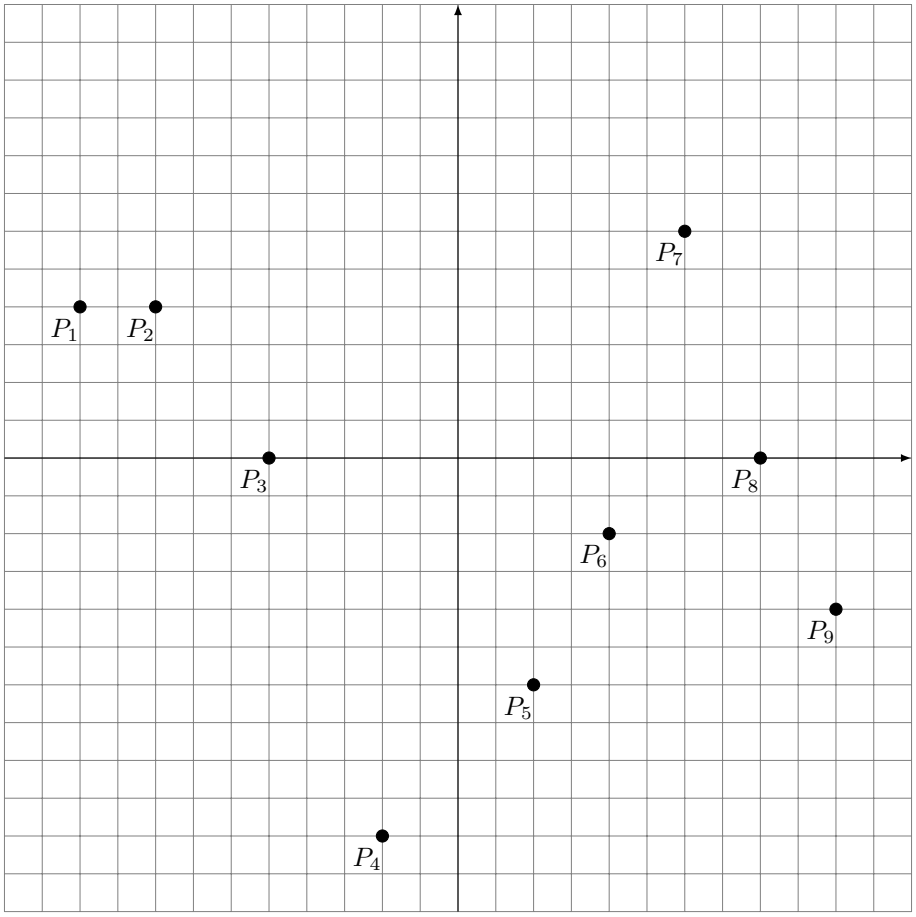
Soluţie.



Lista $\mathcal{L}_i$ evoluează astfel:
P_1P_2
$P_1P_2P_3$
$P_1P_2P_3P_4$ // este eliminat P_3, deoarece P_2, P_3, P_4 coliniare (nu viraj la stânga)
$P_1P_2P_4$
$P_1P_2P_4P_5$
$P_1P_2P_4P_5P_6$
$P_1P_2P_4P_5P_6P_7$
$P_1P_2P_4P_5P_6P_7P_8$ // punctele P_7, P_6, P_5 sunt eliminate în această ordine
$P_1P_2P_4P_8$ // lista finală ($\mathcal{L}_i$) a vârfurilor care determină marginea inferioară

4. *Daţi un exemplu de mulţime $\mathcal{M}$ din planul $\mathbb{R}^2$ pentru care, la final, $\mathcal{L}_i$ are 3 elemente, dar, pe parcursul algoritmului, numărul maxim de elemente al lui $\mathcal{L}_i$ este egal cu 6 ($\mathcal{L}_i$ este lista vârfurilor care determină marginea inferioară a frontierei acoperirii convexe a lui $\mathcal{M}$, obţinută pe parcursul Graham's scan, varianta Andrew). Justificaţi!*

Soluţie.



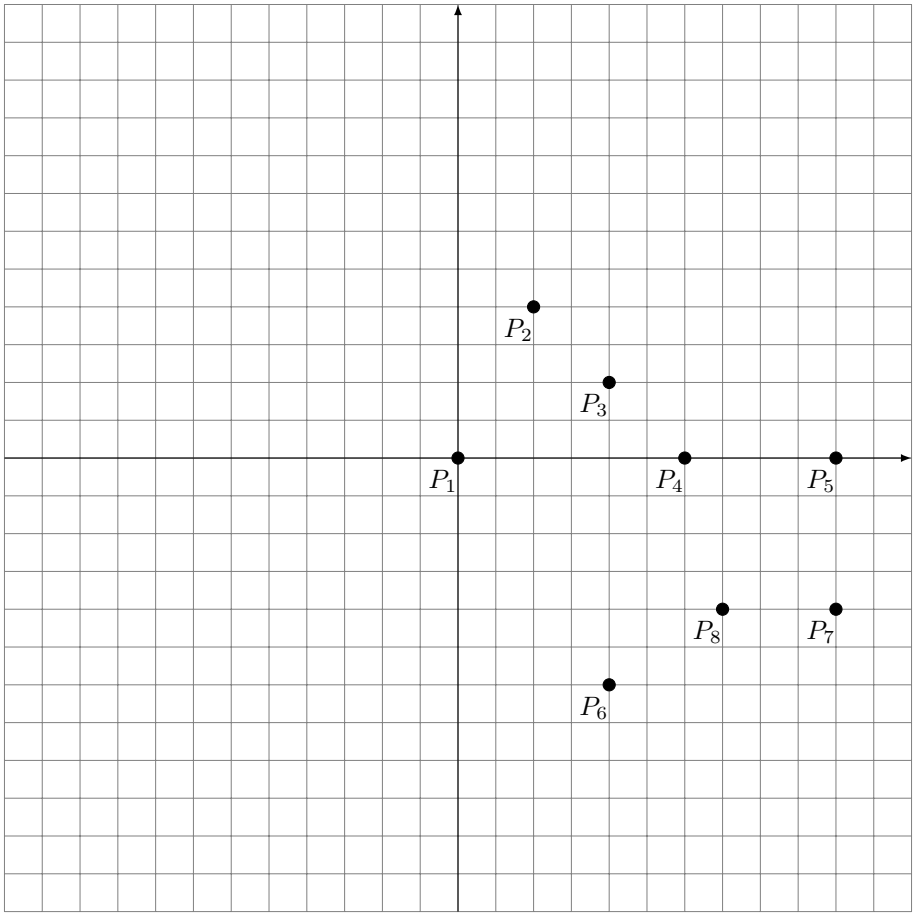
Lista $\mathcal{L}_i$ are la final 3 elemente (P_1, P_4, P_9).

Numărul maxim de elemente este 6: $P_1P_4P_5P_6P_7P_8$ (la adăugarea lui P_8 în listă).

Obs. Numărul maxim de elemente *după verificări ale virajelor* este 5: $P_1P_4P_5P_6P_7$.

5. *Fie mulţimea* $\mathcal{P} = \{P_1, P_2, \ldots, P_7\}$, *unde* $P_1 = (0,0), P_2 = (1,2)$, $P_3 = (2,1), P_4 = (3,0), P_5 = (5,0), P_6 = (2,-3), P_7 = (5,-2)$. *Indicaţi testele care trebuie făcute pentru a găsi succesorul lui* P_1 *atunci când aplicăm Jarvis' march pentru a determina marginea inferioară a acoperirii convexe a lui* $\mathcal{P}$, *parcursă în sens trigonometric (drept pivot iniţial va fi considerat* P_2*).*

Soluţie.



Pentru a găsi succesorul lui P_1 este adăugat pivotul P_2 (S cu notaţia din suportul de curs). Punctele sunt apoi testate, iar dacă un punct P este la dreapta muchiei orientate P_1S, punctul P devine noul pivot.
Punctul P_3: este în dreapta muchiei P_1P_2, deci pivotul P_2 este înlocuit cu P_3.
Punctul P_4: este în dreapta muchiei P_1P_3, deci pivotul P_3 este înlocuit cu P_4.
Punctul P_5: nu este în dreapta muchiei P_1P_4, deci pivotul P_4 rămâne.
Punctul P_6: este în dreapta muchiei P_1P_4, deci pivotul P_4 este înlocuit cu P_6.
Punctul P_7: nu este în dreapta muchiei P_1P_6, deci pivotul P_6 rămâne.
Punctul P_8: nu este în dreapta muchiei P_1P_6, deci pivotul P_6 rămâne.

Au fost parcurse toate punctele. Ultimul pivot (P_6) este succesorul lui P_1 în parcurgerea frontierei acoperirii convexe a mulţimii date în sens trigonometric.

6. *Discutaţi un algoritm bazat pe paradigma Divide et impera pentru determinarea acoperirii convexe. Analizaţi complexitatea-timp.*

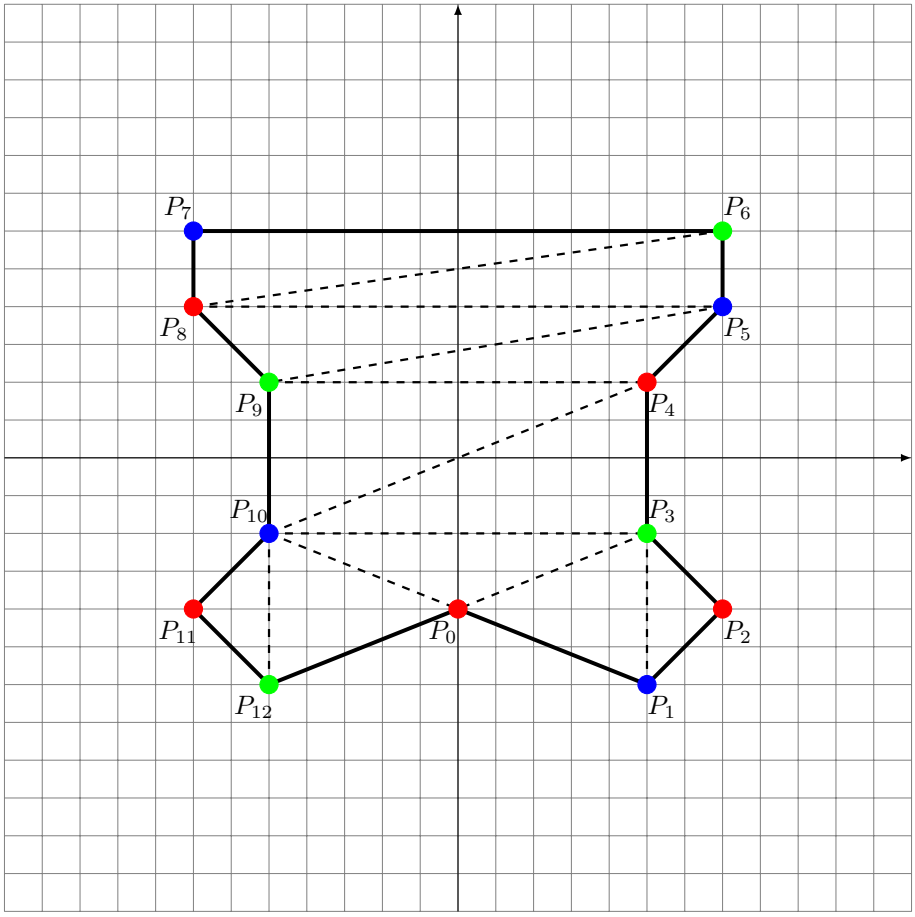
Soluţie. Complexitatea-timp este $O(n \log n)$. O descriere a algoritmului şi a analizei complexităţii poate fi găsită în survey-ul [Lee & Preparata, 1984].

Algoritmi avansaţi

Seminar 6 (săpt. 11 şi 12)

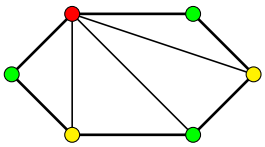
1. *Aplicaţi metoda din demonstraţia teoremei galeriei de artă, indicând o posibilă amplasare a camerelor de supraveghere în cazul poligonului* $P_0P_1P_2\dots P_{12}$, *unde* $P_0 = (0,-2), P_1 = (5,-6), P_2 = (7,-4), P_3 = (5,-2), P_4 = (5,2), P_5 = (7,4)$, $P_6 = (7,6)$ *iar punctele* $P_7,\dots,P_{12}$ *sunt respectiv simetricele punctelor* $P_6,\dots,P_1$ *faţă de axa* Oy.

Soluţie. În figură sunt reprezentate o posibilă triangulare şi 3-colorarea asociată - există şi alte variante corecte.

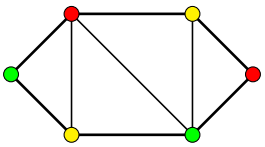


2. *Fie poligonul* $\mathcal{P} = (P_1P_2P_3P_4P_5P_6)$, *unde* $P_1 = (5, 0)$, $P_2 = (3, 2)$, $P_3 = (-1, 2)$, $P_4 = (-3, 0)$, $P_5 = (-1, -2)$, $P_6 = (3, -2)$. *Arătaţi că Teorema Galeriei de Artă poate fi aplicată în două moduri diferite, aşa încât, aplicând metoda din teoremă şi mecanismul de 3-colorare, în prima variantă să fie suficientă o singură cameră, iar în cea de-a doua variantă să fie necesare şi suficiente două camere pentru supravegeherea unei galerii având forma poligonului* $\mathcal{P}$.

Soluţie. Poligonul este un hexagon convex, deci pentru triangularea sa vor fi folosite $3 \cdot 6 - 6 - 3 = 9$ muchii. Aceasta înseamnă că vom trasa 3 diagonale. Sunt posibile două situaţii: (a) cele trei diagonale au un vârf comun; (b) nu există un vârf comun al celor trei diagonale (acest lucru se poate demonstra trasând una dintre diagonale şi apoi raţionând inductiv - este esenţial că poligonul este un hexagon convex). În cazul (a) este suficientă o cameră, iar în cazul (b) 3-colorarea indică utilizarea a două camere.



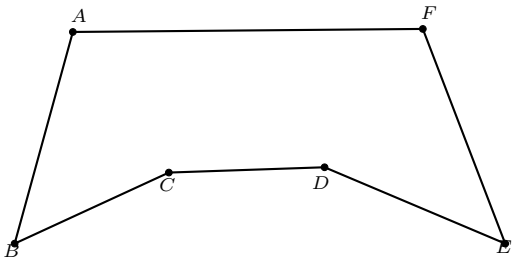
(a)



(b)

3. *Daţi exemplu de poligon cu* 6 *vârfuri care să aibă atât vârfuri convexe, cât şi concave şi toate să fie principale.*

Soluţie. În figură este desenat un poligon cu 4 vârfuri convexe şi 2 vârfuri concave. Pot fi luate în considerare şi alte variante (de exemplu cu un singur vârf concav, cu doar 3 vârfuri convexe, etc.).



4. *Fie* $\mathcal{M} = \{A_i \,|\, i = 0, \ldots, 50\} \cup \{B_i \,|\, i = 0, \ldots, 40\} \cup \{C_i \,|\, i = 0, \ldots, 30\}$*, dată de punctele* $A_i = (i + 10, 0)$*,* $i = 0, 1, \ldots, 50$*,* $B_i = (0, i + 30)$*,* $i = 0, 1, \ldots, 40$*,* $C_i = (-i, -i)$*,* $i = 0, 1, \ldots, 30$*. Determinaţi numărul de triunghiuri şi numărul de muchii ale unei triangulări a lui* $\mathcal{M}$*.*

Soluţie. Trebuie stabilite mai întâi numărul de puncte n şi numărul de puncte de pe frontiera acoperirii convexe k (atenţie la numărarea punctelor, nu trebuie numărat un punct de două ori...). Pe o schiţă se observă că sunt în total 123 de puncte (punctele din mulţimile $\{A_i \,|\, i = 0, \ldots, 50\}$, $\{B_i \,|\, i = 0, \ldots, 40\}$, respectiv $\{C_i \,|\, i = 0, \ldots, 30\}$ sunt diferite între ele). Obţinem $n = 123$, $k = 3$, apoi aplicăm formulele pentru determinarea numărului de triunghiuri, respectiv a numărului de muchii.

$$n_t = 2n - k - 2 = 241, \qquad n_m = 3n - k - 3 = 343.$$

5. *Daţi un exemplu de mulţime din* $\mathbb{R}^2$ *care să admită o triangulare având* 6 *triunghiuri şi* 11 *muchii.*

Soluţie. Fie n numărul de puncte ale unei astfel de mulţimi şi k numărul de puncte de pe frontiera acoperirii convexe. Au loc relaţiile

$$\begin{cases} 2n - k - 2 = 6 \\ 3n - k - 3 = 11 \end{cases}$$

Rezolvând acest sistem obţinem $n = 6$, $k = 4$, deci o astfel de mulţime are 6 puncte, din care 4 sunt situate pe frontiera acoperirii convexe.

Un posibil exemplu: $\{(0, 0), (5, 0), (5, 3), (0, 3), (1, 1), (3, 1)\}$.

6. *În* $\mathbb{R}^2$ *fie punctele* $P_1 = (1, 7)$*,* $P_2 = (5, 7)$*,* $P_3 = (7, 5)$*,* $P_4 = (1, 3)$*,* $P_5 = (5, 3)$*,* $P_6 = (\alpha - 1, 5)$*, cu* $\alpha \in \mathbb{R}$*. Discutaţi, în funcţie de* α*, numărul de muchii ale unei triangulări asociate mulţimii* $\{P_1, P_2, P_3, P_4, P_5, P_6\}$*.*

Soluţie. Trebuie analizată configuraţia punctelor $P_1, P_2, \ldots P_6$ şi determinate numărul n de puncte şi numărul k de puncte de pe frontiera acoperirii convexe.

Punctele $P_1P_2P_3P_4P_5$ determină un pentagon convex. Punctul P_6 descrie o dreaptă paralelă cu Ox care trece prin punctul P_3.

- Pentru $\alpha - 1 \leq 1$, adică $\alpha \in (-\infty, 2]$ punctul P_6 este situat în exteriorul sau pe laturile pentagonului $P_1P_2P_3P_4P_5$. Avem $n = 6, k = 6$, deci 4 feţe şi 9 muchii.
- Pentru $\alpha - 1 > 1$ şi $\alpha - 1 < 7$, adică $\alpha \in (2, 8)$ punctul P_6 este situat în interiorul pentagonului $P_1P_2P_3P_4P_5$. Avem $n = 6, k = 5$, deci 5 feţe şi 10 muchii.
- Pentru $\alpha - 1 = 7$, adică $\alpha \in \{8\}$ punctul P_6 coincide cu P_3. Avem $n = 5, k = 5$, deci 3 feţe şi 7 muchii.
- Pentru $\alpha - 1 > 7$, adică $\alpha \in (8, \infty)$ punctul P_6 este situat în exteriorul pentagonului $P_1P_2P_3P_4P_5$. Avem $n = 6, k = 6$, deci 4 feţe şi 9 muchii.

7. *Fie $\mathcal{G}$ un graf planar conex, v numărul de noduri, m numărul de muchii, f numărul de feţe. Se presupune că fiecare vârf are gradul ≥ 3. Demonstraţi inegalităţile*

$$v \leq \tfrac{2}{3}m, \qquad m \leq 3v - 6$$
$$m \leq 3f - 6, \quad f \leq \tfrac{2}{3}m$$
$$v \leq 2f - 4, \quad f \leq 2v - 4$$

Daţi exemplu de grafuri în care au loc egalităţi în relaţiile de mai sus.

Algoritmi avansaţi

Seminar 7 (săpt. 13 şi 14)

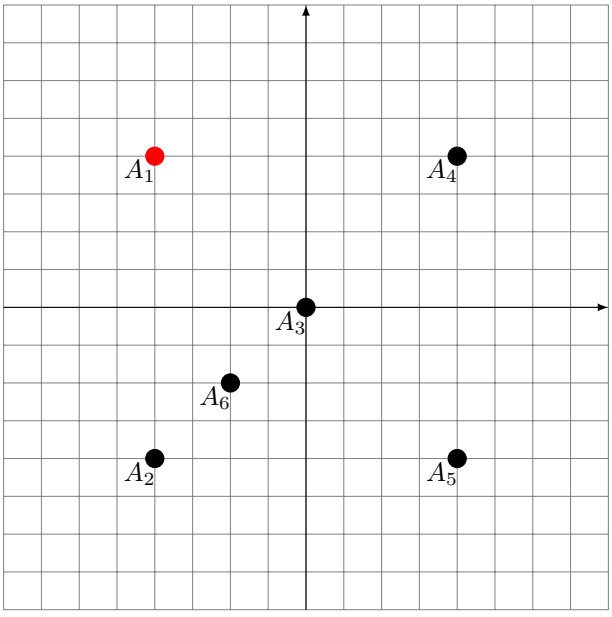
1. *Daţi exemplu de mulţime $\mathcal{M} = \{A_1, A_2, A_3, A_4, A_5, A_6\}$ din $\mathbb{R}^2$ astfel ca diagrama Voronoi asociată lui $\mathcal{M}$ să conţină exact patru semidrepte, iar diagrama Voronoi asociată lui $\mathcal{M} \setminus \{A_1\}$ să conţină exact cinci semidrepte. Justificaţi alegerea făcută.*

Soluţie. Numărul de semidrepte ale unei diagrame Voronoi este egal cu numărul de puncte de pe frontiera acoperirii convexe.

Alegem mulţimea $\mathcal{M}$ astfel ca A_1, A_2, A_3, A_4 să determine frontiera acoperirii convexe, iar pentru $\mathcal{M} \setminus \{A_1\}$ toate punctele să fie situate pe frontiera acoperirii convexe.

În figură este indicată configuraţia punctelor, însă, întrucât în enunţ se cere ca punctele să fie din $\mathbb{R}^2$, le indicăm în coordonate. În exemplul ales $A_1 = (-4, 4)$, $A_2 = (-4, -4)$, $A_3 = (0, 0)$, $A_4 = (4, 4)$, $A_5 = (4, -4)$, $A_6 = (-2, -2)$.

O altă soluţie posibilă (în care punctele A_2, A_6, A_3, A_4 nu mai sunt coliniare, este $A_1 = (-4, 4)$. $A_2 = (-4, -4)$, $A_3 = (0, 3)$, $A_4 = (4, 4)$, $A_5 = (4, -4)$, $A_6 = (-3, 0)$ (verificaţi prin desen!).



2. *a) Fie o mulţime cu n situri necoliniare. Atunci, pentru diagrama Voronoi asociată au loc inegalităţile*

$$n_v \leq 2n - 5, \qquad n_m \leq 3n - 6,$$

unde n_v este numărul de vârfuri ale diagramei şi n_m este numărul de muchii al acesteia.

b) Câte vârfuri poate avea diagrama Voronoi $\mathcal{D}$ asociată unei mulţimi cu cinci puncte din $\mathbb{R}^2$ ştiind că $\mathcal{D}$ are exact cinci semidrepte? Analizaţi toate cazurile. Este atins numărul maxim de vârfuri posibile ($n_v = 2n - 5$)? Justificaţi!

Soluţie. a) Cum punctele nu sunt coliniare, diagrama Voronoi are muchii de tip segment şi de tip semidreaptă. Se consideră un vârf suplimentar, notat v_∞, prin care, prin convenţie, trece fiecare muchie de tip semidreaptă. Se construieşte un graf $\mathcal{G}$ care are

- *vârfuri*: vârfurile diagramei Voronoi şi vârful v_∞ (deci $n_v + 1$ vârfuri);
- *muchii*: muchiile diagramei Voronoi - fiecare uneşte exact două vârfuri ale lui $\mathcal{G}$ (deci n_m muchii);
- *feţe*: în corespondenţă biunivocă cu siturile iniţiale (deci n feţe).

Graful $\mathcal{G}$ este un graf planar conex.

(i) Conform relaţiei lui Euler avem

$$(n_v + 1) - n_m + n = 2.$$

(ii) Analizăm incidenţele dintre muchii şi vârfuri.

- fiecare muchie este incidentă cu exact două vârfuri;
- fiecare vârf (inclusiv v_∞) este incident cu cel puţin trei muchii;
- numărând incidenţele în două moduri avem

$$2n_m \geq 3(n_v + 1).$$

Din (i) şi din (ii) rezultă inegalităţile dorite.

Observaţie. Pentru ca relaţiile din enunţ să fie egalităţi este necesar şi suficient ca fiecare vârf al lui $\mathcal{G}$ să fie incident cu exact trei muchii. Aceasta înseamnă să nu existe grupuri de (cel puţin) patru puncte conciclice şi pe frontiera acoperirii convexe a mulţimii de situri să fie exact trei situri.

b) Fie $\mathcal{M}$ o mulţime ca în enunţ. Diagrama Voronoi a lui $\mathcal{M}$ are cinci semidrepte, deci toate cele cinci puncte ale lui $\mathcal{M}$ sunt situate pe frontiera acoperirii convexe. În particular, conform observaţiei anterioare, nu este posibil să fie atins numărul maxim de vârfuri posibile ($n_v = 2n - 5 = 5$). Mai mult, urmând paşii din raţionament, observăm că, de fapt, $n_v \leq 3$. Sunt posibile trei situaţii:

- toate punctele sunt conciclice, $n_v = 1$,

- patru dintre puncte sunt conciclice, dar al cincilea nu este situat pe acelaşi cerc cu acestea, $n_v = 2$,
- oricare patru puncte nu sunt conciclice, $n_v = 3$.

3. *Fie punctele $O = (0,0)$, $A = (\alpha, 0)$, $B = (1,1)$, $C = (2,0)$, $D = (1,-1)$, unde $\alpha \in \mathbb{R}$ este un parametru. Discutaţi, în funcţie de α, numărul de muchii de tip semidreaptă ale diagramei Voronoi asociate mulţimii $\{O, A, B, C, D\}$.*

Soluţie. Numărul de muchii de tip semidreaptă este egal cu numărul de puncte de pe frontiera acoperirii convexe (le notăm cu m).

Punctul A este variabil pe axa Ox. Punctele O, B, C, D determină un pătrat şi trebuie doar să stabilim, în funcţie de α, poziţia lui A faţă de acest pătrat. Distingem următoarele cazuri:

- $\alpha < 0$: punctul A este situat în exteriorul pătratului $OBCD$ şi avem $m = 4$ (punctele de frontieră sunt A, B, C, D);
- $\alpha = 0$: avem $A = O$, deci $m = 4$;
- $0 < \alpha < 2$: punctul A este situat în interiorul pătratului $OBCD$ şi avem $m = 4$ (punctele de frontieră sunt O, B, C, D);
- $\alpha = 2$: avem $A = C$, deci $m = 4$;
- $\alpha > 2$: punctul A este situat în exteriorul pătratului $OBCD$ şi avem $m = 4$ (punctele de frontieră sunt O, B, D, A).

4. *(i) Fie punctul $A = (1,2)$. Alegeţi două drepte distincte d, g care trec prin A, determinaţi dualele A^*, d^*, g^* şi verificaţi că A^* este dreapta determinată de punctele d^* şi g^*.*

(ii) Determinaţi duala următoarei configuraţii: Fie patru drepte care trec printr-un acelaşi punct M. Se aleg două dintre ele; pe fiecare din aceste două drepte se consideră câte un punct diferit de M şi se consideră dreapta determinată de cele două puncte. *Desenaţi ambele configuraţii. Completaţi configuraţia iniţială (adăugând puncte/drepte) astfel încât să obţineţi o configuraţie autoduală (i.e. configuraţia duală să aibă aceleaşi elemente geometrice şi aceleaşi incidenţe ca cea iniţială).*

Soluţie. a) Conform teoriei, pentru un punct $p = (p_x, p_y)$ se defineşte dreapta $p^* : (y = p_x x - p_y)$ (duala lui p), iar pentru o dreaptă neverticală $d : (y = mx + n)$, se defineşte punctul $d^* = (m, -n)$ (dualul lui d).

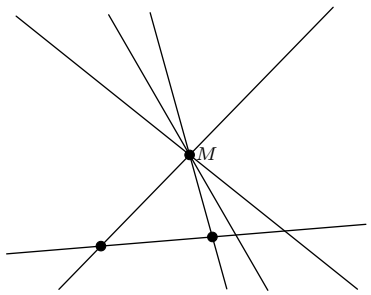
Alegem dreptele $d : (y = 2x)$ şi $g : (y = -x + 3)$. Avem:

$$A^* : (y = x - 2), \quad d^* = (2,0), \quad g^* = (-1,-3).$$

Se verifică imediat că punctele d^* şi g^* determină dreapta A^*.

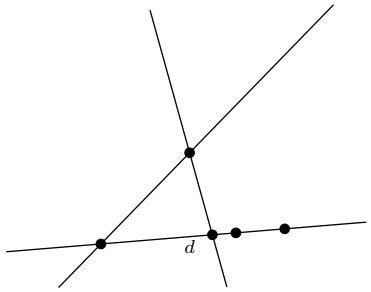
b) Configuraţia iniţială:

Fie patru drepte care trec printr-un acelaşi punct M. Se aleg două dintre ele; pe fiecare din aceste două drepte se consideră câte un punct diferit de M şi se consideră dreapta determinată de cele două puncte.



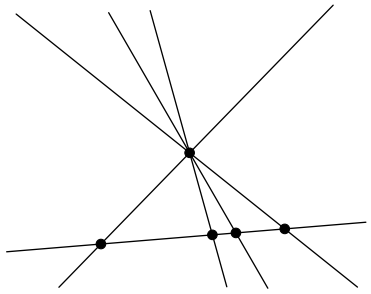
Configuraţia duală:

Fie patru puncte situate pe o aceeaşi dreaptă d. Se aleg două dintre ele; prin fiecare din aceste două puncte se consideră câte o dreaptă diferită de d şi se consideră punctul de intersecţie al acestor două drepte.



Configuraţia completată (astfel ca să fie autoduală):

Fie patru drepte care trec printr-un acelaşi punct. Pe fiecare dreaptă se alege câte un punct diferit de punctul comun, astfel ca aceste patru puncte să fie coloniare şi se consideră dreapta determinată de aceste patru puncte.



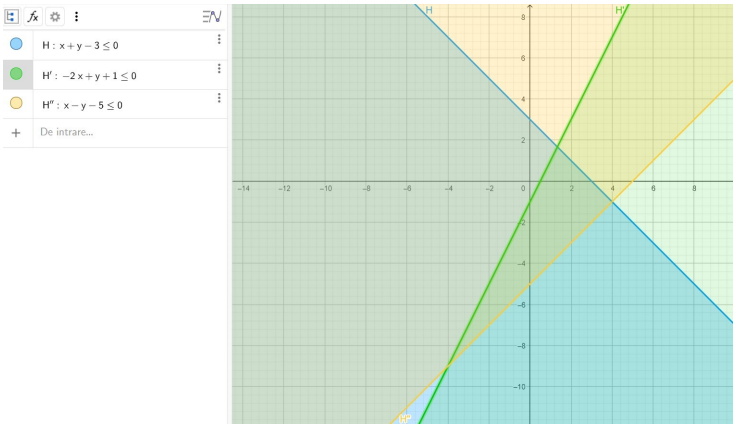
5. *a) Fie semiplanele* $H : x + y - 3 \leq 0$ *şi* $H' : -2x + y + 1 \leq 0$. *Daţi exemplu de semiplan* H'' *astfel ca intersecţia* $H \cap H' \cap H''$ *să fie un triunghi dreptunghic.*

b) Fie semiplanele H_1, H_2, H_3, H_4 *date de inecuaţiile*

$$H_1 : -y + 1 \leq 0; \qquad H_2 : y - 5 \leq 0; \qquad H_3 : -x \leq 0; \qquad H_4 : x - y + a \leq 0,$$

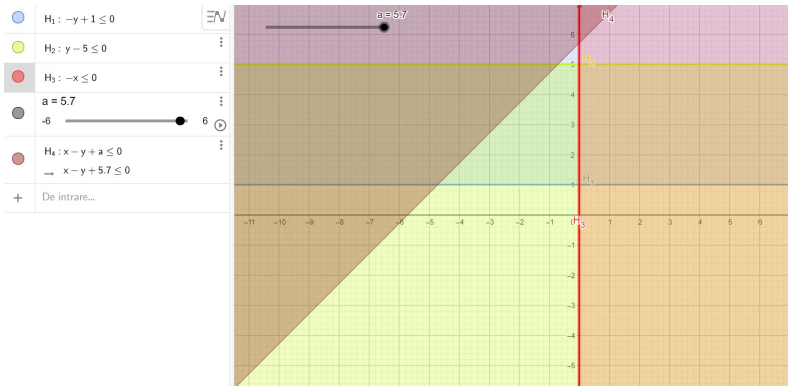
unde $a \in \mathbb{R}$ *este un parametru. Discutaţi, în funcţie de parametrul* a*, natura intersecţiei* $H_1 \cap H_2 \cap H_3 \cap H_4$.

Soluţie. a) Alegem semimplanul $H'' : x - y - 5 \leq 0$. Dreapta suport $x - y - 5 = 0$ este perpendiculară pe dreapta suport a lui H, $x + y - 3 = 0$. Intersecţia semiplanelor este un triunghi (cf. figură, se pot determina vârfurile acestuia), deci H'' verifică cerinţele din enunţ.

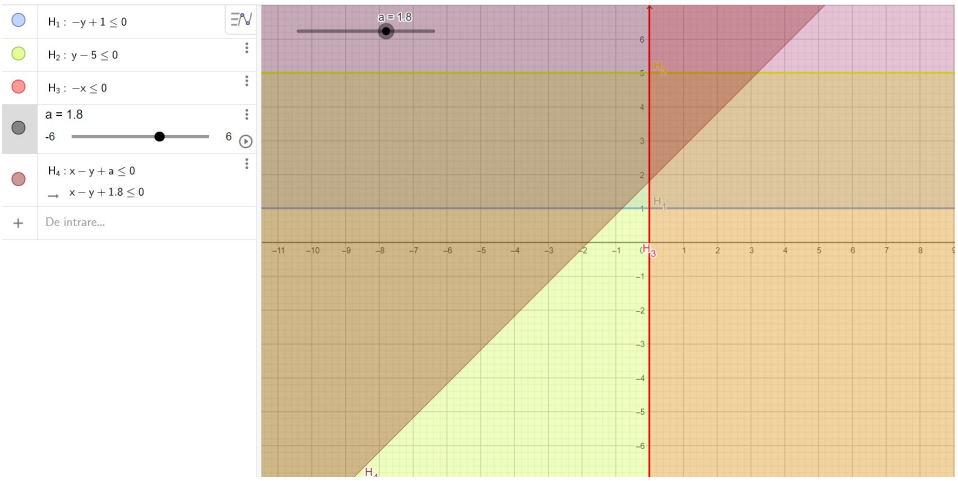


b) Intersecţia $H_1 \cap H_2 \cap H_3$ este o mulţime convexă nemărginită delimitată de dreptele $y = 1$, $y = 5$, respectiv $x = 0$. Vârfurile relevante sunt punctele $(0, 1)$ şi $(0, 5)$.

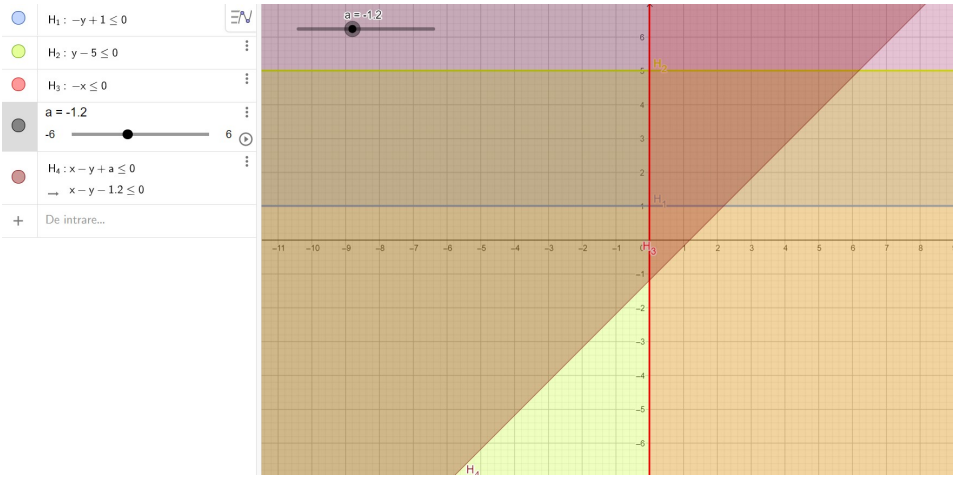
- Pentru $a > 5$ se obţine $0 \geq x - y + a > x - y + 5 \geq 0$ (deoarece $x \geq 0$ şi $-y + 5 \geq 0$), contradicţie. În acest caz intersecţia este mulţimea vidă.



- Pentru $a = 5$ se obţine punctul $(0, 5)$.
- Pentru $1 \leq a < 5$ se obţine un triunghi având unul dintre vârfuri $(0, 5)$, un vârf la intersecţia dintre dreptele $x = 0$ şi $x - y + a = 0$ şi un vârf la intersecţia dintre dreptele $y = 5$ şi $x - y + a = 0$. Pentru $a = 1$ punctul $(0, 1)$ este vârf al acestui triunghi.



- Pentru $a < 1$ se obţine un trapez. Laturile sale au ca drepte suport exact dreptele suport ale celor patru semiplane.



6. *Scrieţi inecuaţiile semiplanelor corespunzătoare şi studiaţi intersecţia acestora, dacă normalele exterioare ale feţelor standard sunt coliniare cu vectorii*

$$(0,1,-1),\ (0,1,0),\ (0,0,-1),\ (0,-1,0),\ (0,-1,-1).$$

Soluţie. Conform teoriei, dacă normala exterioară a unei feţe f este coliniară cu un vector $\vec{v}_f = (a,b,c)$, semiplanul corespunzător feţei f este

$$ax + by + c \leq 0.$$

Se ajunge la inegalităţile

$$\begin{aligned} y - 1 &\leq 0 \\ y &\leq 0 \\ -1 &\leq 0 \\ -y &\leq 0 \\ -y - 1 &\leq 0 \end{aligned}$$

Acest sistem este compatibil, admiţând soluţia $y = 0$.

Notă. Ca în exemplul de la curs, am făcut abstracţie de componenta x.

7. ***(Suplimentar)*** *Demonstraţi că arborele parţial de cost minim al lui $\mathcal{P}$ este un subgraf al triangulării Delaunay.*

Soluţie. D. Mount, *Computational Geometry*, pag. 75.