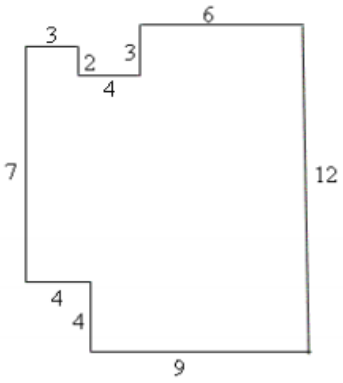


25) Problema amplasării optime a trei camere video

Student: Petre Tiberiu

Să se determine o amplasare optimă a trei camere mobile de luat vederi pentru supravegherea continuă a spațiului din fig. De mai jos a.î. acoperirea acestora să fie maximă. Soluția se va da prin coordonatele celor trei puncte de amplasare, considerând originea axelor de coordonate carteziene în colțul din stânga jos.

- *Date de intrare:*
  - Se consideră încăperea ca fiind definită de coordonatele colțurilor acesteia
  - $P_k(x_k, y_k), P_k\ k = \overline{1,n}$ colțurile încăperii

Model de optimizare

- *Necunoscuta:*
  - $v = (x_1, y_1, x_2, y_2, x_3, y_3)$
- *Restricții:*
  - Camera video trebuie amplasată pe tavanul încăperii.

(Se considera o dreaptă orizontală ce trece prin punctul $Cam_i(x_i, y_i)$ și se numără de câte ori se intersectează cu o latură aflată în dreptul camerei video și la dreapta pe graficul încăperii. Dacă se intersectează de un număr par de ori atunci nu este validă, altfel, este validă.)

- *Funcția obiectiv:*

- Se cere să se maximizeze aria acoperită de o cameră video dată de intersecția a două dreptunghiuri (orizontal și vertical). Funcția obiectiv este următoarea:

$$maximul\ lui\ F: \mathbb{R}^6 \to \mathbb{R}, F(x_1, y_1, x_2, y_2, x_3, y_3) = \bigcup_{i=1}^{3} A_i(h_i \cap v_i)$$

$$(x_i, y_i) \in (h_i \cap v_i)\ şi\ h_i, v_i\ sunt\ dreptunghiurile\ orizontal\ şi\ vertical$$

- Determinată folosind următoarele formule:
  - Distanța Manhattan dintre două puncte: $d(A_k, B_k) = \left|x_{B_k} - x_{A_k}\right| + \left|y_{B_k} - y_{A_k}\right|$
  - Ecuația unei drepte $d_k: \frac{x - x_{B_k}}{x_{A_k} - x_{B_k}} = \frac{y - y_{B_k}}{y_{A_k} - y_{B_k}} => d_k: a_k x - b_k y - c_k = 0$
  - Intersecția a două drepte:
    - $S = \begin{cases} ax - by - c = 0 \\ dx - ey - f = 0 \end{cases} => S: \begin{pmatrix} a & -b \\ d & -e \end{pmatrix} * \begin{pmatrix} x_P \\ y_P \end{pmatrix} = \begin{pmatrix} c \\ f \end{pmatrix} => \begin{pmatrix} x_P \\ y_P \end{pmatrix} = A^{-1} * \begin{pmatrix} c \\ f \end{pmatrix}$
    - $A^{-1} = \frac{1}{-(ae+bd)} * \begin{pmatrix} -e & d \\ -b & a \end{pmatrix} = \begin{pmatrix} \frac{e}{ae+bd} & \frac{-d}{ae+bd} \\ \frac{b}{ae+bd} & \frac{-a}{ae+bd} \end{pmatrix}$
    - $P = \begin{pmatrix} x_P \\ y_P \end{pmatrix} = \begin{pmatrix} + \frac{ce+fb}{ae+bd} \\ - \frac{af+cd}{ae+bd} \end{pmatrix}$
  - Aria unui dreptunghi: $A_{(x_i, y_i) \in dr} = l_1 * l_2,\ unde\ l_1, l_2\ laturile\ dreptunghiului$

Caracterizarea problemei:

- *Optimizare matematică* deoarece se folosesc formule matematice pentru a determina soluțiile optime
- *Optimizare statică* deoarece datele de intrare a problemei rămân neschimbate pe parcursul optimizării
- *Optimizare parametrică* deoarece necunoscuta este un vector cu 6 componente (coordonatele celor trei camere video)
- *Optimizare cu restricții* deoarece are restricții peste necunoscută. (Camerele video trebuie amplasate pe tavanul încăperii)
- *Programare continuă* deoarece spațiul de căutare este unul infinit. (Are o infinitate de soluții fezabile)
- *Programare neliniară* deoarece funcția obiectiv este una neliniară.
- *Programare în numere reale* deoarece datele de intrare, coordonatele necunoscutei și rezultatul întors de funcția obiectiv sunt numere reale.
- *Programare neconvexă* deoarece funcția obiectiv poate avea mai mult de un singur optim local
- *Programare deterministă* deoarece datele de intrare sunt cunoscute a priori procesului de optimizare
- *Optimizare uniobiectiv* deoarece se folosește doar o singură funcție obiectiv pentru optimizare

Dimensiunea spațiului de căutare

a. Deoarece funcția obiectiv nu este bijectivă (pentru un număr de necunoscute diferite se poate obține aceeași arie) și camerele video pot fi amplasate într-o infinitate de poziții diferite atunci și *spațiul soluțiilor este unul infinit*.

```
# se determină timpul necesar generării a 1000 de soluții valide
start_time = time.time()
for i in tqdm(range(1000)):
    x = camereVideo.gen_nec(False)
    arie = camereVideo.F(x)
    solutii.append(np.hstack((x.flatten(), arie)))
timp = time.time()-start_time
```

Timp scurs: 13.47 secunde / 1000 soluții
Timp total pentru 1.000.000 soluții: 3 ore 44 minute

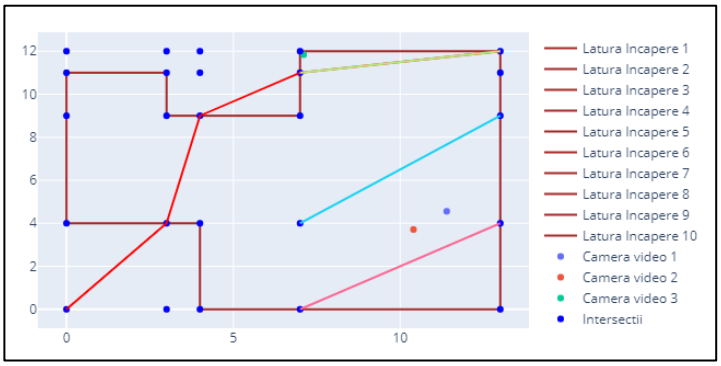
b. Nu se pot găsi toate soluțiile fezabile într-un timp scurt folosind Brute Force deoarece pentru 1.000.000 de soluții ar lua ~4 ore pe un Intel Core i5

c. S-a constatat că în aceste 1000 soluții aleatorii se poate identifica un optim global de valoare 69 pe instanța de problemă dată ca exemplu.

Aplicarea a două tehnici de optimizare

a. S-a plecat de la o soluție inițială aleasă aleatoriu:

```
x = camereVideo.gen_nec(False)
print(
    "Necunoscută:", x.flatten(),
    '->', camereVideo.F(x)
)
camereVideo.plot(x)
```

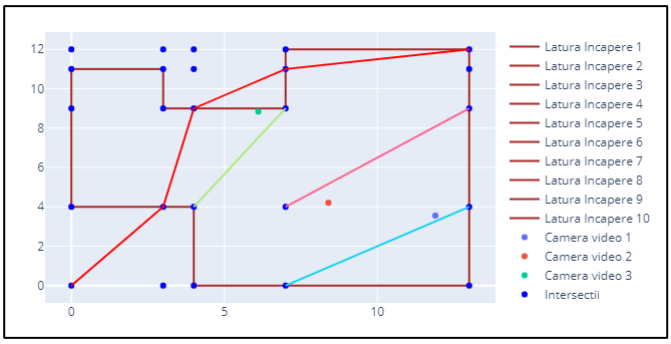


*Necunoscută: [11.39493602 4.55748469 10.39939467 3.71054938 7.10815816 11.83986702] -> **60.0***

b. Tehnica ***Hill Climbing***: aceasta a găsit o soluție optimă, abia prin a treia metodă, cea cu reporniri aleatorii.

```
# o distribuție aleatoare a fracțiunilor de timp alocați
T = self._distributie_temporala(timp_alocat)
# valoarea optimă inițială
x = self.copie(self._x)
x_optim = x
timp1 = time.time()
while True:
    # se alege aleatoriu un timp din T
    t = rand.choice(T)
    timp2 = time.time()
    # execută cat timp nu s-a scurs fracțiunea de timp sau timpul total
    while True:
        # se modifică x a.î. să rămână valid
        r = self.modifica(self.copie(x), p)
        # dacă nu este valid mă opresc
        if r is None: break
        # dacă este mai bun îl aleg
        if self.valoare(r)>self.valoare(x):
            x=r
        delta_time1 = time.time()-timp2
        delta_time2 = time.time()-timp1
        # dacă a expirat timpul t sau cel total, mă opresc
        if delta_time1>t or delta_time2>timp_alocat: break
    # în cazul în care x este mai bun de ce aveam bun îl salvez
    if self.valoare(x)>self.valoare(x_optim):
        x_optim = x
    # dacă timpul total a expirat
    delta_time = time.time()-timp1
    # mă opresc
    if delta_time>timp_alocat: break
# întorc valoarea optimă găsită
return x_optim
```

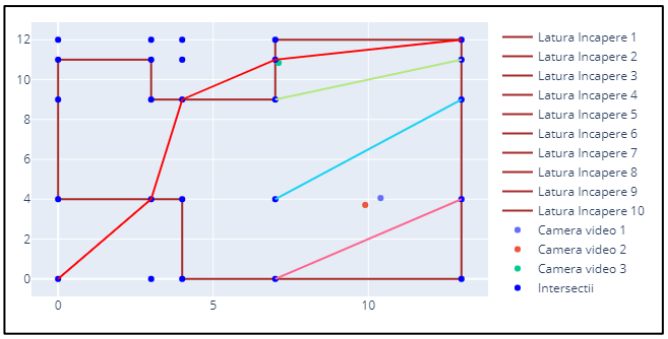
*Necunoscută: [11.89493602 3.55748469 8.39939467 4.21054938 6.10815816 8.83986702] -> **69.0***



c. Tehnica ***Simulated Annealing***: aceasta a găsit o soluție optimă, mai proastă comparativ cu Hill Climbing

```
# Lista soluțiilor candidat
xt = np.array([self.copie(self._x)])
t = 0
x = xt[t]
x_optim = x
# Temperatura inițială este un număr foarte mare
self.beta0 = rand.random()*rand.randint(10000, 1000000)
# Temperatura beta1 este mai mică decât beta0 cu un raport de 0.7
self.beta = self.beta1 = 0.7*self.beta0
timp = time.time()
# Cât timp nu am ajuns la soluția optimă (metalul nu s-a călit suficient)
while True:
    # Repet de z ori procesul de călire la temperatura beta
    for _ in range(z):
        # Aleg un vecin aleator a lui xt[t]
        r = rand.choice(self.vecini(self.copie(xt[t])))
```

```
            if self.q(r)>self.q(xt[t]): # este mai bun ca xt[t]
                xt = np.vstack((xt, r))
            if self.q(r)>self.q(x_optim): # este mai bun ca x_optim
                x_optim = r
            # Repetă până la cvasiechilibru
            if self.q(r)<=self.q(xt[t]): # este mai slab ca xt[t]
                u = rand.random() # generez aleator un u ~U(0,1)
                # mai mic ca probabilitatea de acceptare
                if u <= self.probabilitate(xt[t],r):
                    xt = np.vstack((xt, r)) # îl accept și ma opresc
                    break
                else: # îl resping și consider pe xt[t]
                    xt = np.vstack((xt, xt[t]))
                    # repetă până la cvasiechilibru
        # Actualizez temperatura beta
        self.beta = math.pow(self.beta1/self.beta0, k)*self.beta
        t = t+1
        delta_time = time.time()-timp
        # Dacă temperatura este suficient de
        # mică sau nu a expirat timpul
        if self.beta <= 1e-3
            or delta_time>timp_alocat:
            break # mă opresc
    return x_optim # întorc soluția optimă
```



*Necunoscută: [10.39493602 4.05748469 9.89939467 3.71 054938 7.10815816 10.83986702] -> **66.0***

Inițial	***11.39493602***	***4.55748469***	***10.39939467***	***3.71054938***	***7.10815816***	***11.83986702***	***60.0***
HC	*11.89493602*	*3.55748469*	*8.39939467*	*4.21054938*	*6.10815816*	*8.83986702*	***69.0***
SA	*10.39493602*	*4.05748469*	*9.89939467*	*3.71054938*	*7.10815816*	*10.83986702*	***66.0***

Tabel 1 Analiză comparativă a soluțiilor găsite