Testing

# Image Processing

# Descriere

Modulul de procesare a imaginii se ocupa cu încărcarea unei imagini de pe disc (format jpg sau png) și *detectarea* în aceasta a diverselor obiecte ce se regăsesc în imagine. Această **detectare** a fost împărțită în **2 părți** – detectarea obiectelor generice (cum ar fi *uși, geamuri, scări, lifturi etc.* ) și eliminarea acestora din imagine și apoi detectarea *pereților* – care în principiu e detectare a *liniilor* ce formează pereții și apoi unificarea acestora (într-o anumită *marjă de eroare* pentru *lățime*) pentru crearea pereților actuali.

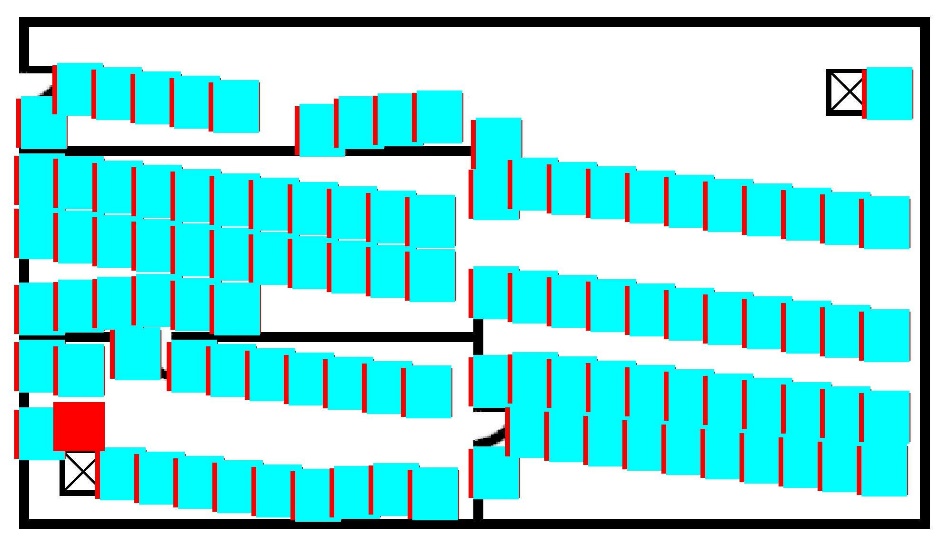
Desigur, pentru că vorbim despre procesare de imagini, există două lucruri ce trebuie luate în considerare: **timpul** necesar procesării, care tinde să fie cât se poate de mare, și **erorile**- pentru că până la urmă, cu toate preprocesările făcute, detectarea pe imagini este cât se poate de *probabilistă.*

# Detectarea Obiectelor

Detectarea obiectelor are loc in clasa ***DetectObject***al pachetului ***com.imgprocessor.processor***. Obiectele detectate in urma procesarii vor fi cele care au si un template asociat (imagine a obiectului respectiv) ex: scari, usi, hidranti, fereste, ascensoare, panouri electrice etc.

Primul pas in detectare acestora, a fost incercarea de a le identifica printr-un algoritm de ***template matching*** oferit de biblioteca **OpenCV** care se baza pe raportul de pixeli colorati din imaginea Template si imaginea Scena. Dar acesta s-a dovedit a nu fi unul foarte precis, avand marja de eroare foarte mare, din cauza ca lucram cu imagini bicolore (blueprint-uri)

Marja de eroare era destul de mare, o confirmare este urmatorul test esuat. Algoritmul are un comportament imprevizibil deoarece la modificarea dinamica a coeficientului de matching nu mai lucreaza template-Matching-ul :



Problema acestei metode de matching este urmatoarea : fiecare obiect are un *ratio* de matching si nu poate fi generalizata pentru toate template-urile, ceea ce creaza ingeuneaza adaugarea a noi template-uri pentru o eventuala noua imagine pentru procesare. Asadar am **renunțat** la aceasta metoda de detectare a obiectelor.

In urma unor cercetari am descoperit un alt algoritm de matching care sa bazeaza pe keyPoint-uri (**features2d**).

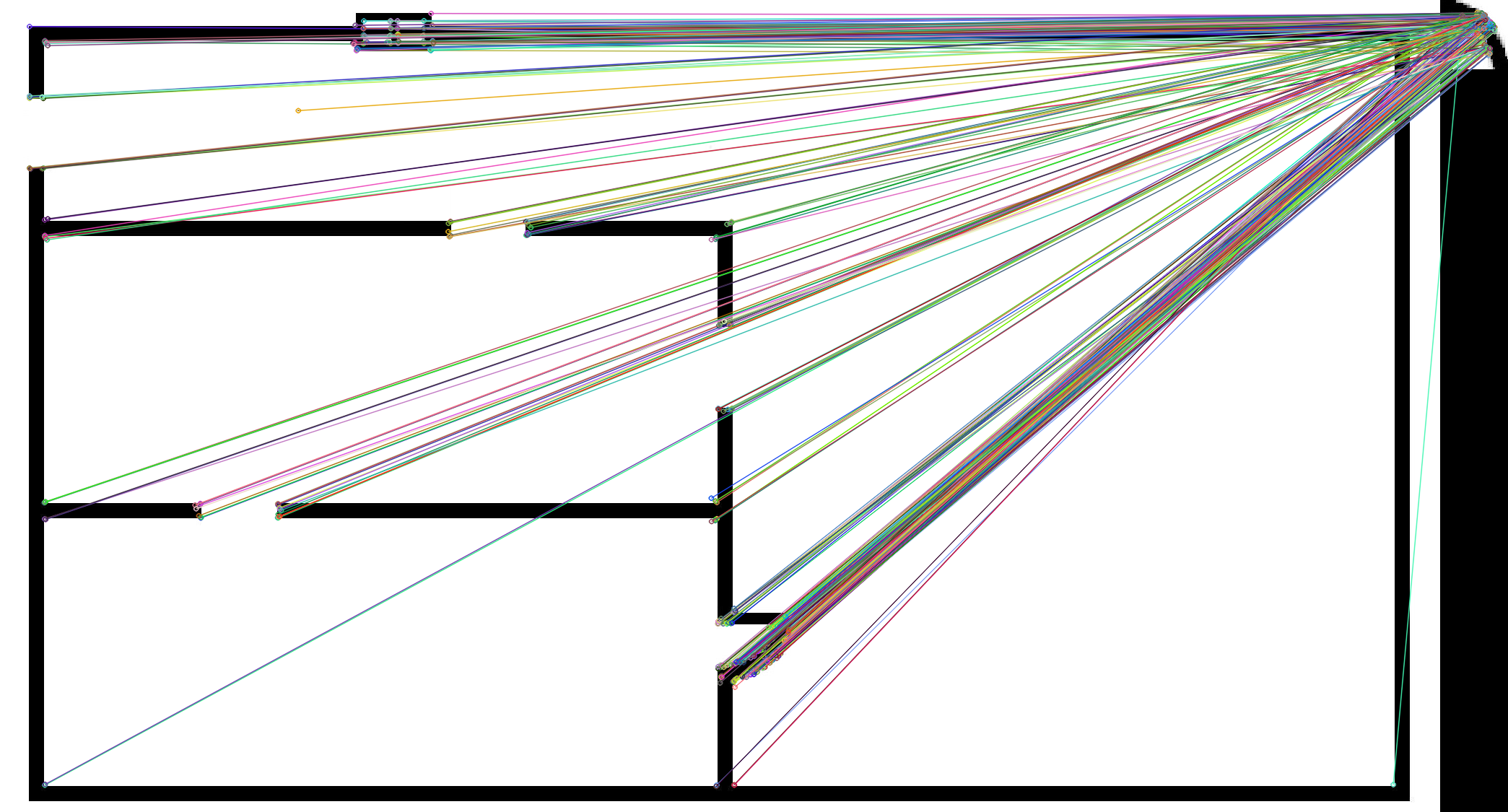
Avem 2 imagini ***Template***-ul si ***Scena***

1. Într-un prim pas, determinam keyPoint-uri in imaginea Template cu ajutorul FeautureDetector, apoi le extragem cu ajutorul DescriptorExtractor

featureDetector.detect(objectImage, objectKeyPoints);  
DescriptorExtractor descriptorExtractor = DescriptorExtractor.*create*(DescriptorExtractor.***SURF***);

descriptorExtractor.compute(objectImage, objectKeyPoints, objectDescriptors);

1. Determinam keyPoint-urile pentru imaginea Scena in aceeasi maniera .
2. Dupa apelam algoritmul knnMatch care va potrivi keyPoint-urile care fac match din acele 2 imagini



1. Algoritmul gaseste mai multe match-uri unele din fiind incorecte de aceea ele trebuiesc filtrate, conform unor calcule pentru “Clusterizare”. Le alegem pe cele mai apropiate intre ele.

float nndrRatio = 0.7f;

for (int i = 0; i < matches.size(); i++)

{

MatOfDMatch matofDMatch = (MatOfDMatch) matches.get(i);

DMatch[] dmatcharray = matofDMatch.toArray();

DMatch m1 = dmatcharray[0];

DMatch m2 = dmatcharray[1];

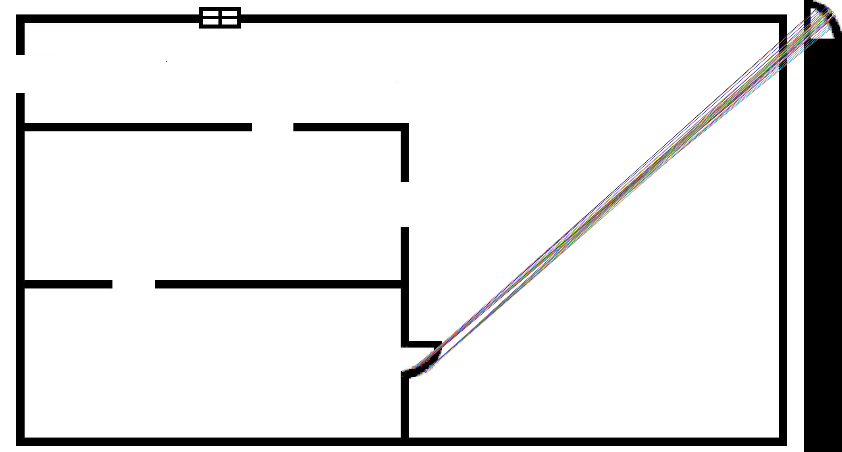
if (m1.distance <= m2.distance \* nndrRatio)

{

goodMatchesList.addLast(m1);

}

}

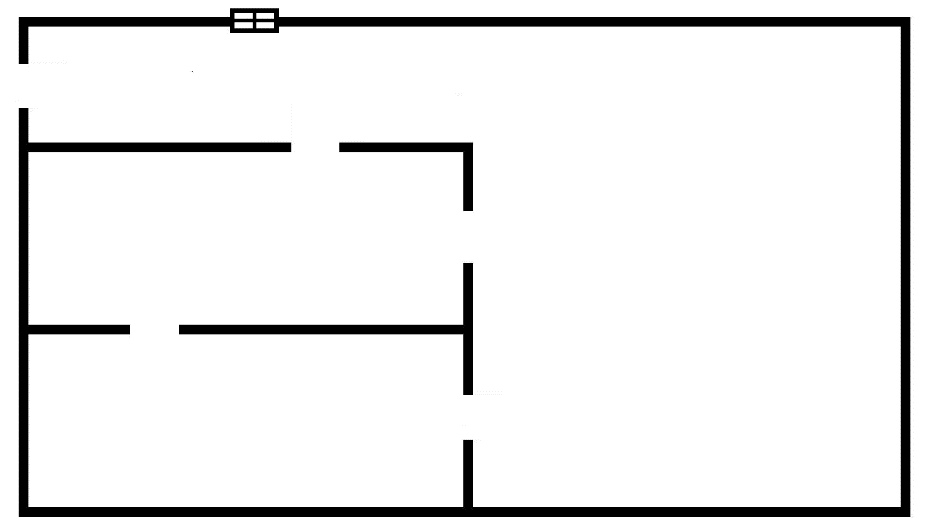


1. Avand aceste key-point-uri care se potrivesc putem determina coordonatele template-ului in cadrul scenei mari

(obs) Detectarea acestei zone e posibila daca au fost gasite cel putin 4 (pentru precizie am pus 7 KeyPoint-uri).

* 1. Daca imaginea Template este prea mica, algoritmul nu va da match deoarece nu vor fi gasite destule keyPoint-uri.

1. Avand coordonatele zonei care da match memoram obiectul gasit si il stergem din Scena

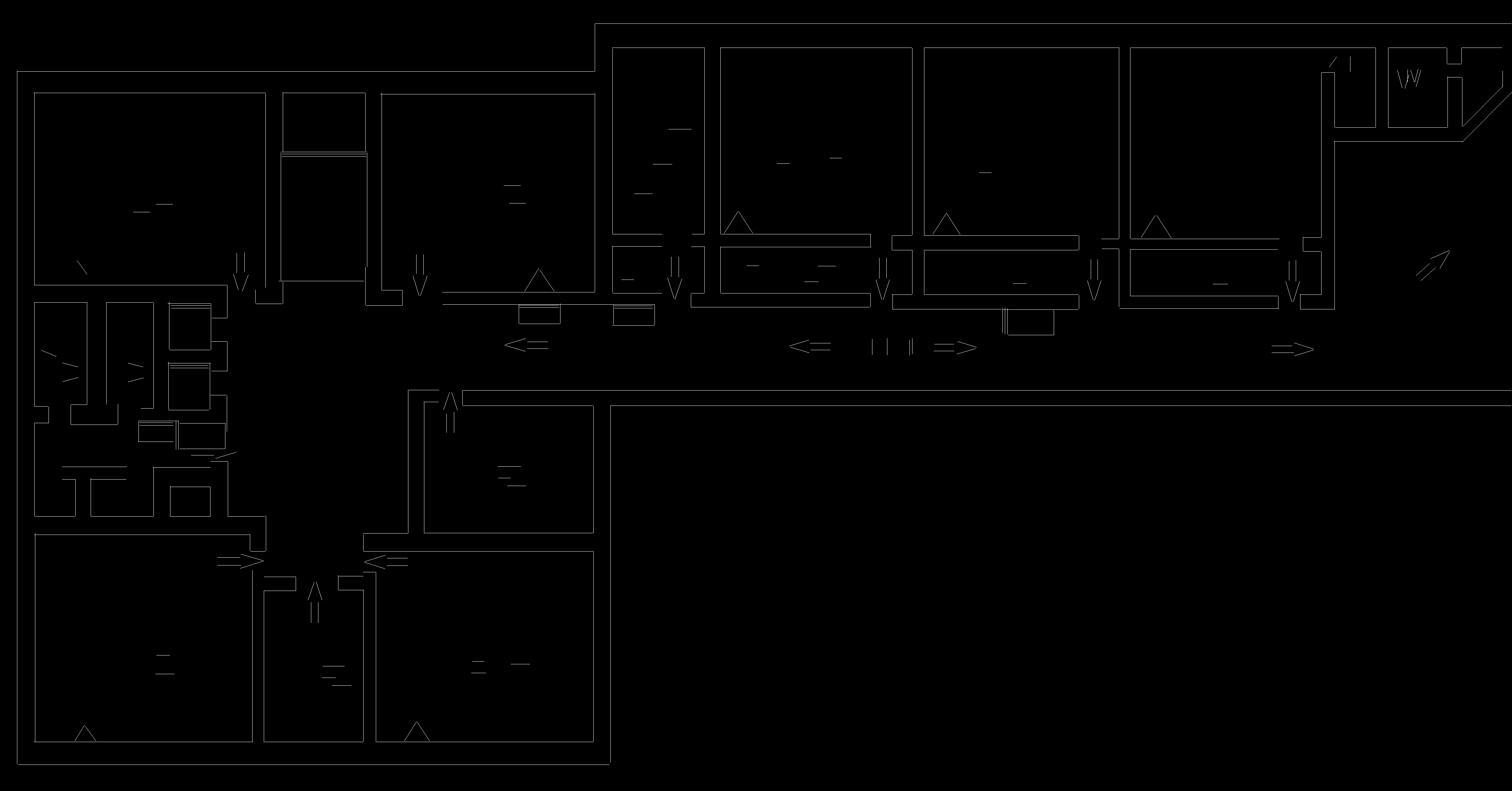


# Detectarea pereților

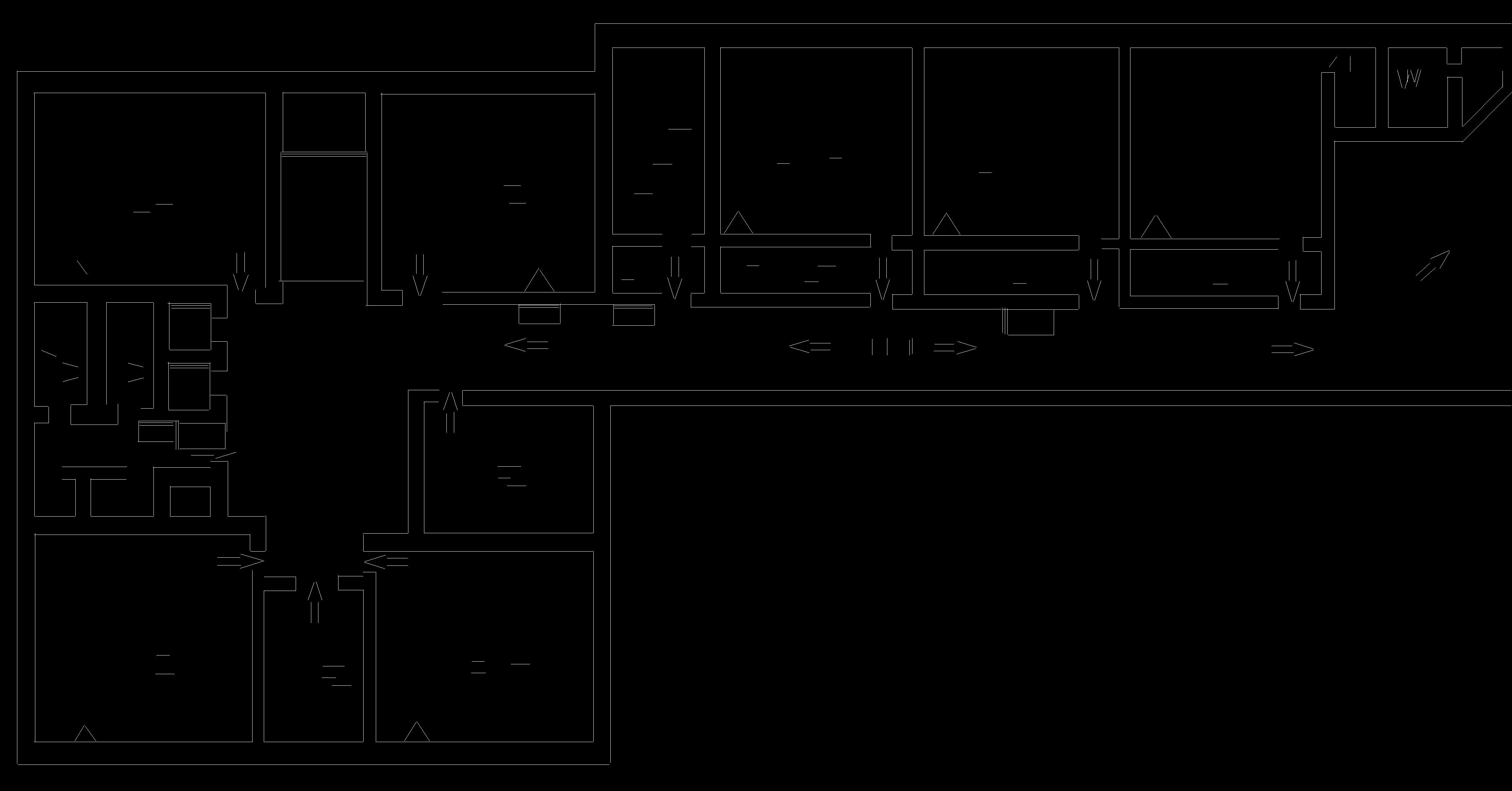
Pentru detectarea pereților, într-o primă instanță se obține imaginea de la punctul 1, cu obiectele eliminate (pentru că algoritmul se folosește de **detectarea liniilor** de o anumită dimensiune, deci ar putea interfera).

Pașii (cu erorile posibile) sunt:

* 1. Convertirea imaginii în una **alb – negru**, pentru a elimina diversele nuanțe de culori și a avea clar posibile linii cu o culoare, restul cu cealaltă.
  2. **Blurring** (prin metoda Billateral Filtering) este folosită pentru a elimina din imagine diversele imperfecțiuni. Desigur, această metodă nu are mereu succes, putând uneori să îmbunătățească detecția, alteori să o înrăutățească.
  3. Poate cea mai costisitoare dintre operații este **denoising**-ul. Acesta este folosit pentru eliminareea noise-ului din imagini, care mai mereu interfereaza cu liniile și poate genera detectări false. Desigur, metoda nu este perfectă pentru orice imagine și nu garantează că nu va înrăutăți procesul.
  4. Este rulat apoi efectiv algoritmul de detectare a liniilor. Pe o imagine simplă, rezultatul acestuia este dat mai jos. Se poate observa că detectarea are marje de erori.



* 1. Urmează o serie de **post-procesări** efectuate pe imagine. Îndreptarea liniilor, eliminarea unor linii micuțe, anumită unificare a anumitor linii care au capete foarte apropiate. Un rezultat ar fi mai jos.



* 1. În următoarea instanță se observă că avem **conturul liniilor**. Deci, trebuie să unim cele două linii (sau mai multe – pentru că o linie poate fi unificată cu mai multe linii, fiind una mare care cuprinde mai multe sub-linii). Această unificare o facem după o **marjă de eroare pentru lățime**, ce poate varia de la schemă la schemă (și poate fi modificată în interfață). Rezultatele pot varia, cum se poate observa la punctul următor.

# Testarea aplicației per total

La **integrarea** întregii aplicații a apărut o problemă *neașteptată –* cele două părți ale detectării se foloseau de funcții ale bibliotecii OpenCV, dar din versiuni diferite, incompatibile una cu alta. Așa că rezultatul a fost crearea a două proiecte separate.

Una din soluții era să creăm fișiere executabile, sau jar-uri, și apoi să apelăm pe unul din ele din celălalte. Problema iarăși apărută este că OpenCV este scris nativ în C++, deci la rulare face apeluri de sistem, iar jar-urile se descurcă cam greu cu apelurile de sistem. Deci soluția aceasta nu a mers (după multiple încercări).

Soluția următoare era referențierea unuia din proiecte din celălalt. Dar și aceasta a eșuat aproximativ din aceeași problemă.

Soluția finală a fost una tricky – în primă instanță **referențierea** librăriei OpenCV 3.1 (cea mai nouă), care avea toate funcțiile disponibile în cea veche, înafară de features2d, deci interfețele claselor nu ar fi produs erori dacă importam și biblioteca veche (pentru că nu se suprascriau una pe alta în acest mod). Iar apoi, în cod, acestea trebuiau să execute *apeluri de sistem* pentru a-și folosi funcționalitățile. Am aflat *numele de sistem* pe care-l foloseau cele două versiuni și am pus apelurile de sistem fix când e nevoie specific doar de unul din ele, pentru a nu se intersecta și a produce iarăși erori.

# Test 1

1. Planul original al etajului unei cladiri:



In urma rularii programului vom identifica 4 tipuri de obiecte prezente in plan: panouri electrice,lifturi,hidranti,scari.

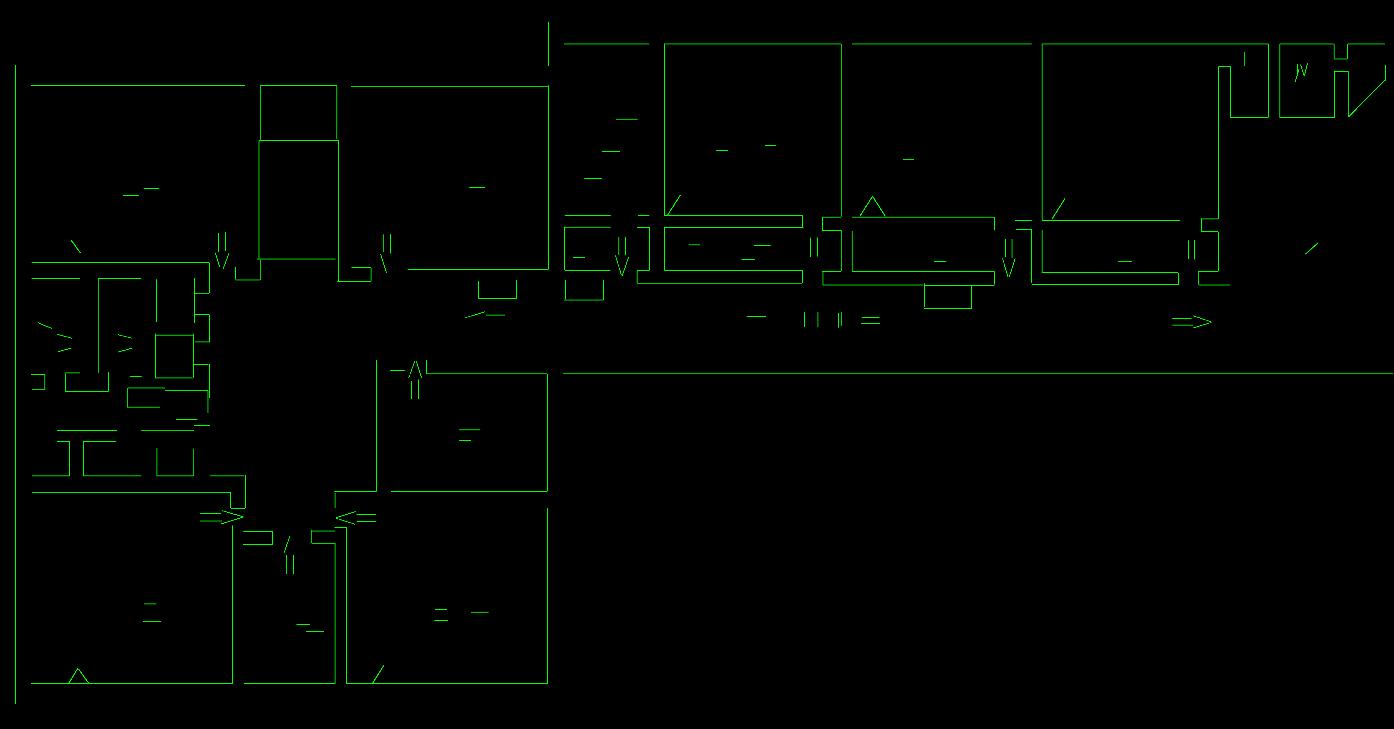
## Imaginea obtinuta dupa identificarea obiectelor este:



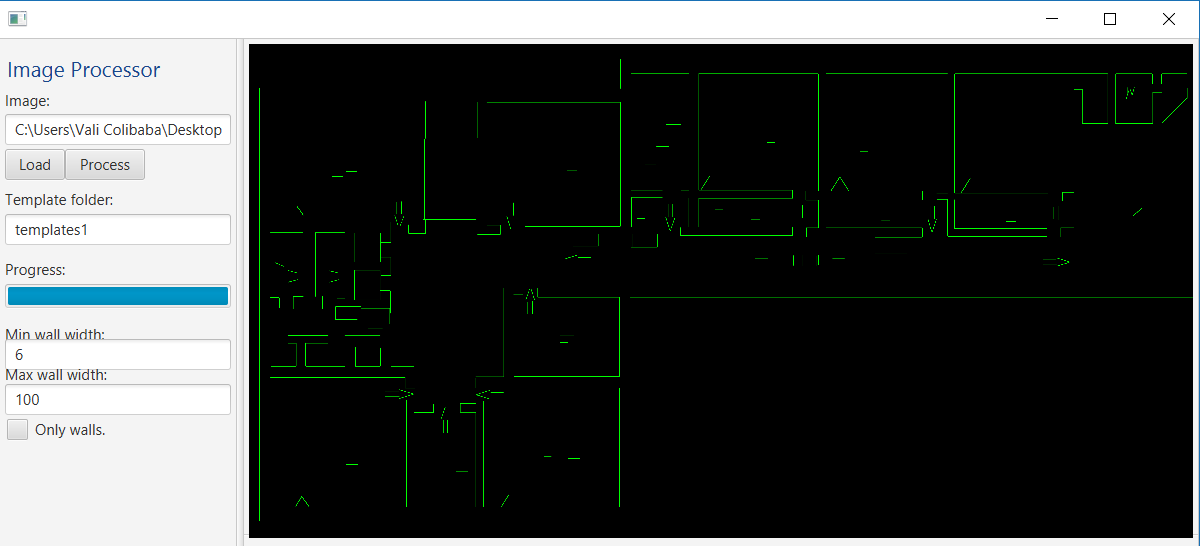
S-au identificat cu succes toate tipurile de obiecte vizate astfel incat am obtinut: 2 hidranti,2 lifturi,2 panouri electrice si o scara. Numarul de obiecte identificate corespunde cu numarul real de obiecte prezente in planul original ,prin urmare si pozitionarea lor in plan, ceea ce ne duce la o rata de succes de 100% in identificarea obiectelor.

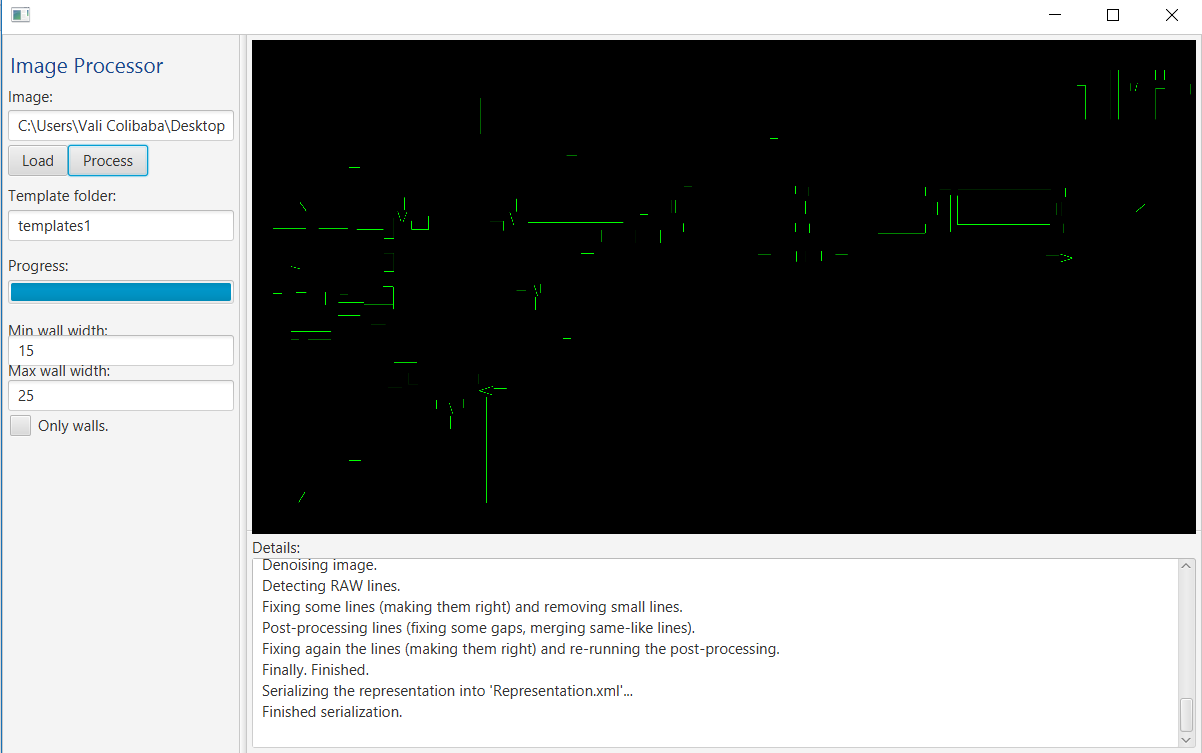
## Urmatorul pas in procesarea planului original este detectarea peretilor.

Am obtinut urmatoarea imagine:



Peretii au fost identificati cu o acuratete de aproximativ 92%.Erorile sunt cauzate de inconsistenta liniilor ce formeaza peretii ,fapt ce duce si la deconectarea in anumite zone a liniilor.In urma numeroaselor teste parametrii optimi pentru obtinerea acestui rezultat au fost : min\_wall\_width=6 si max\_wall\_width=100.



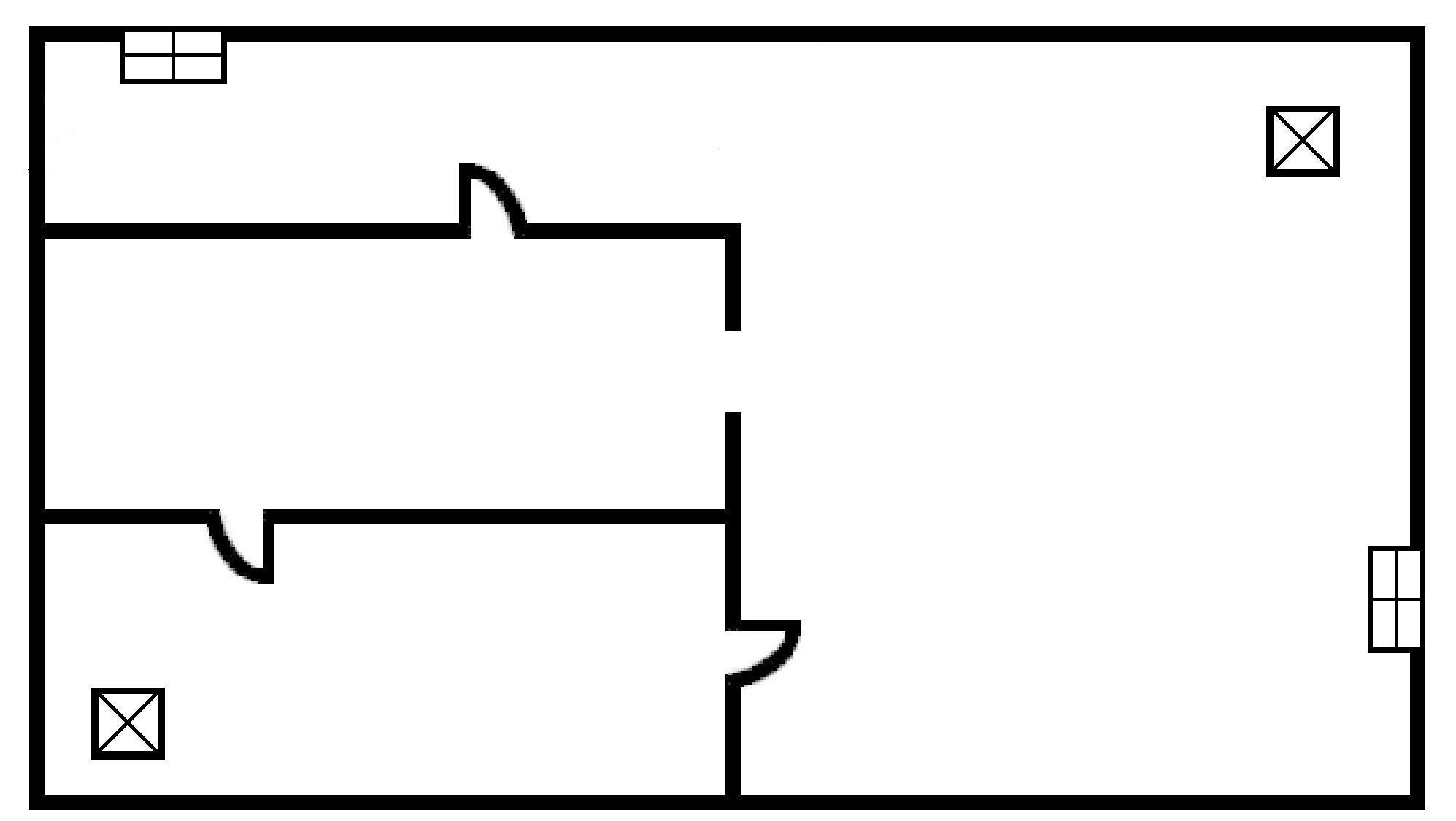
*Un rezultat obtinut utilizand altii parametri este:* 

Se observa ca precizia si completitudinea peretilor a scazut drastic la circa 35% prin schimbarea parametrilor.

In concluzie imaginea a fost procesata folosind parametrii optimi cu o exactitate de circa 90% ,deci o finalizare cu succes a testului 1.

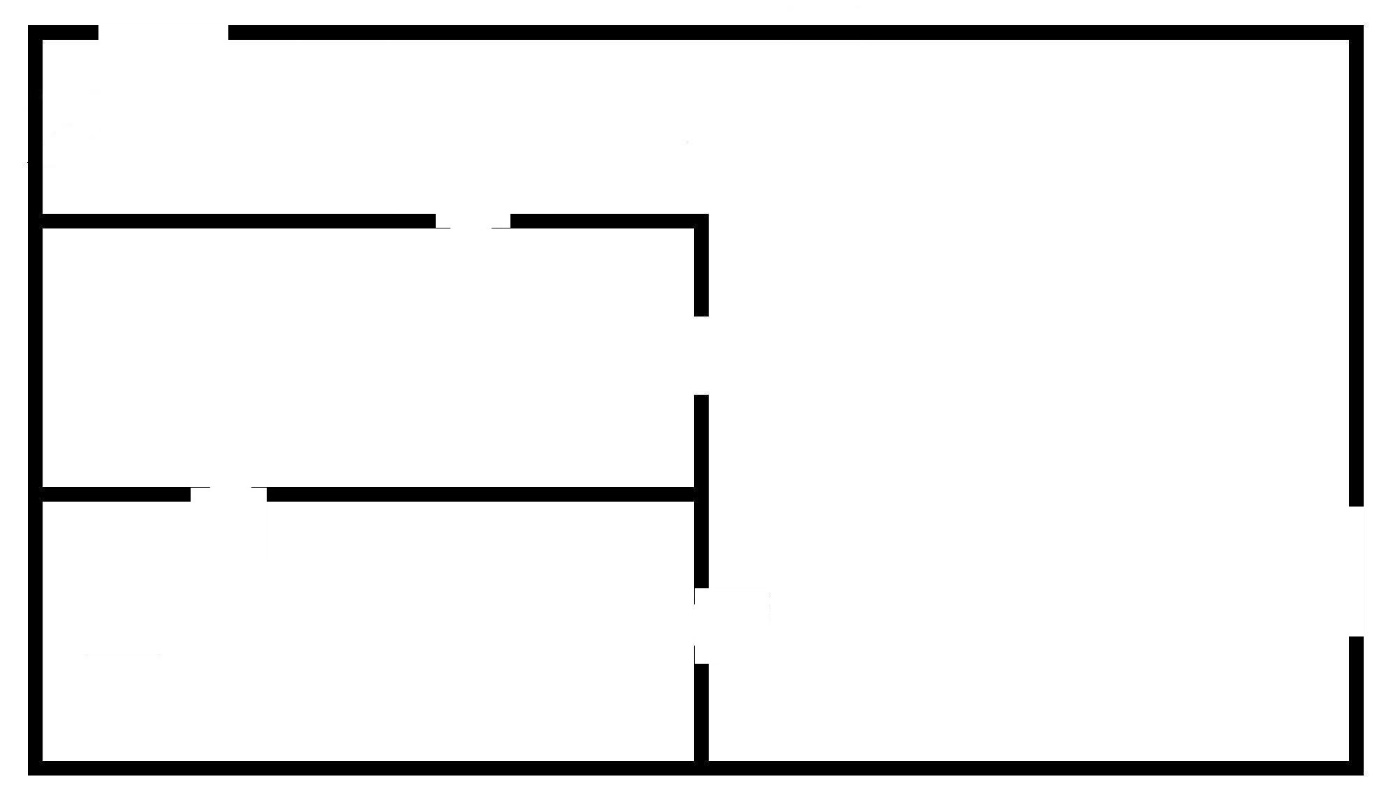
# Testul 2

## Planul original al unei etaj de cladire:



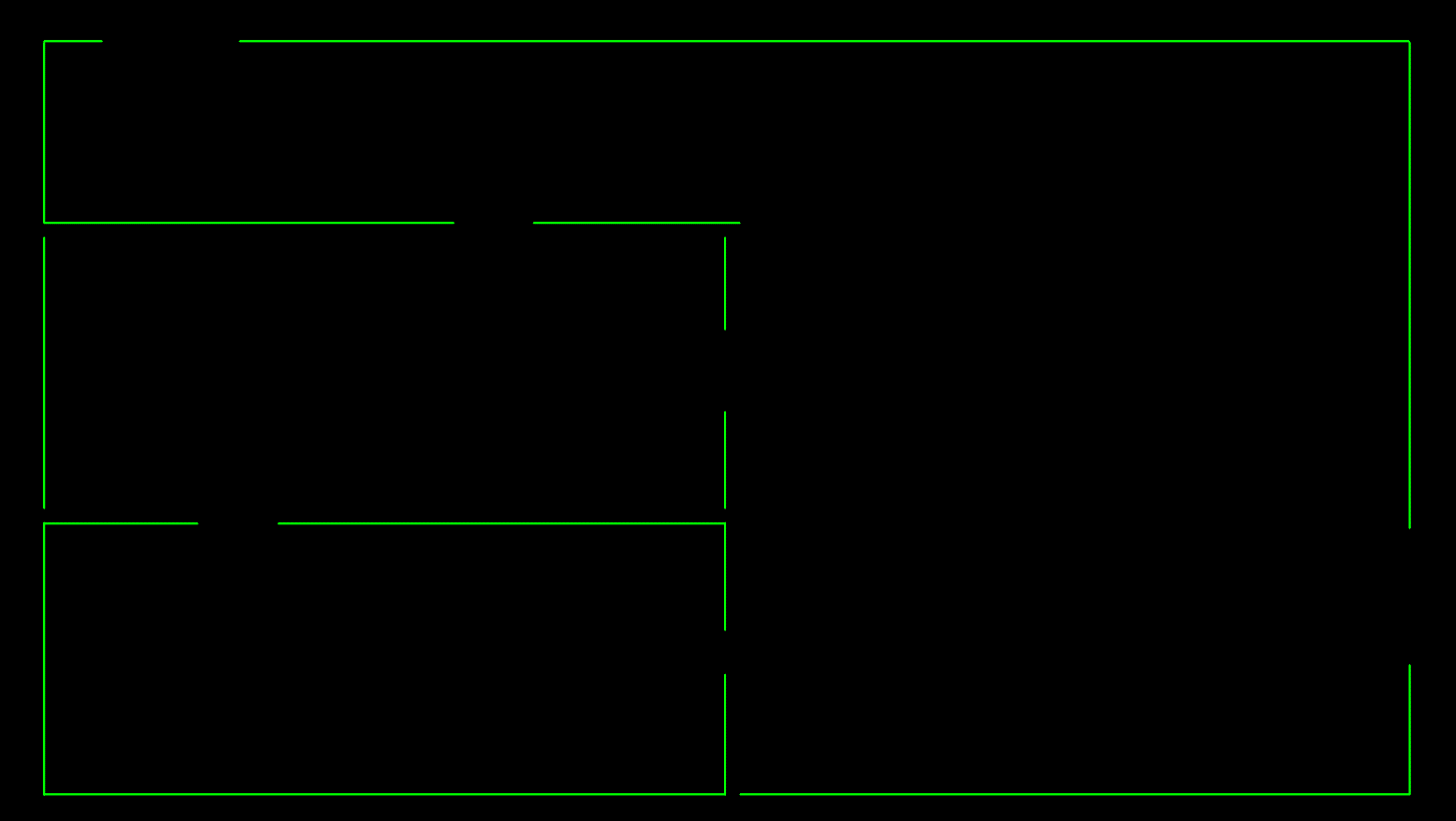
In urma rularii programului vom identifica 3 tipuri de obiecte: usi,lifturi si ferestre.

## Imaginea dupa identificarea obiectelor este:

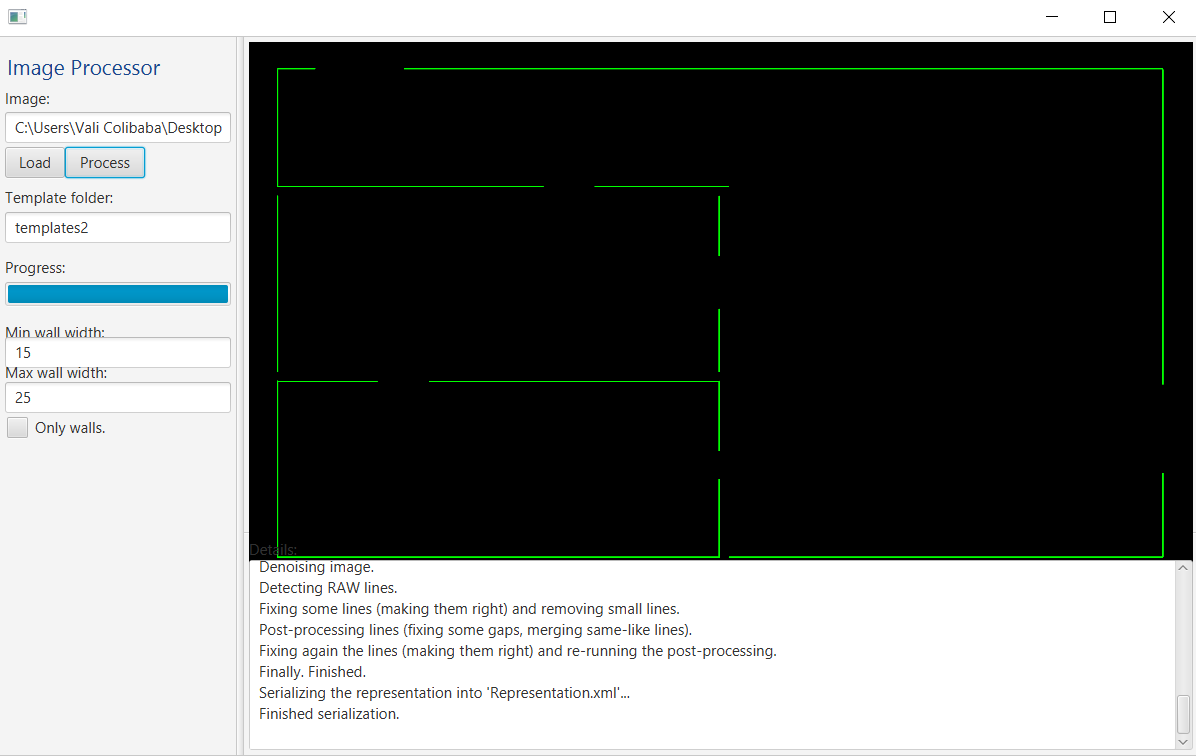


Observam ca s-au identificat cu succes toate tipurile de obiecte si anume: 2 lifturi,2 ferestre si 3 usi.Pozitia obiectelor identificate corespunde cu pozitia lor initiala in proportie de 99% .

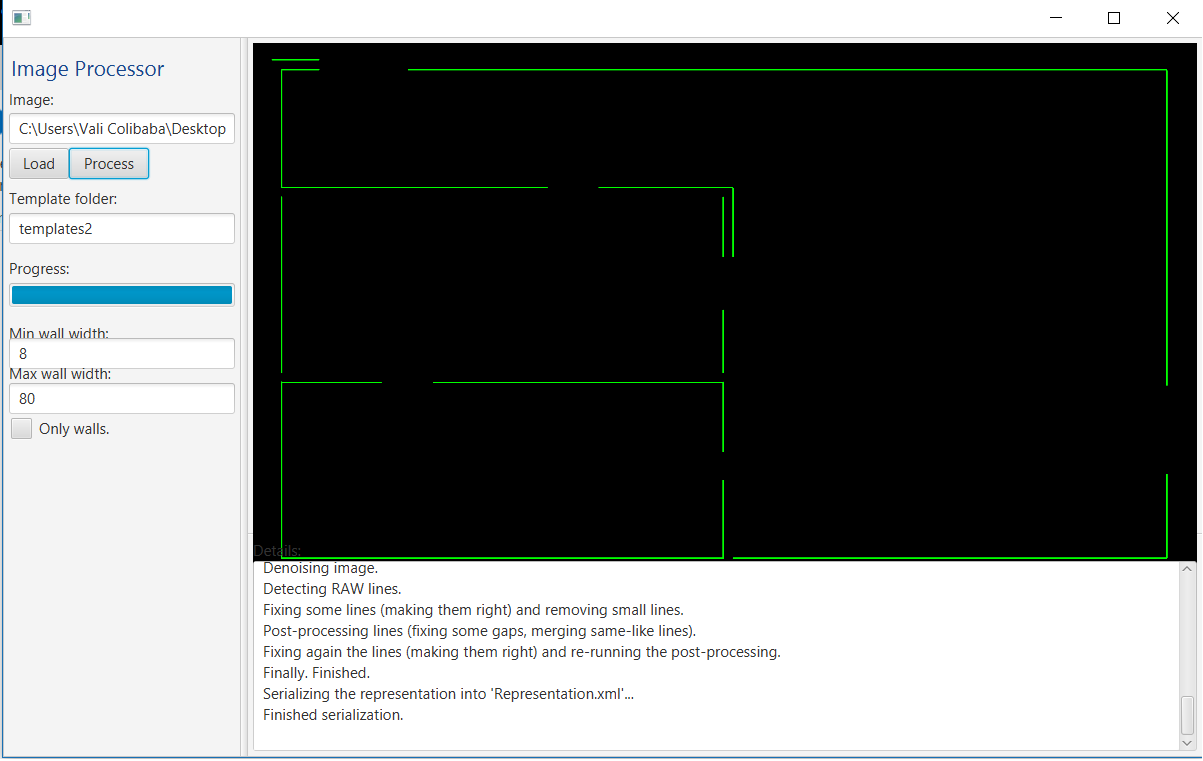
## Ultima etapa reprezentata de identificarea peretilor a dus la obtinerea urmatorului rezultat:



Peretii au fost identificat cu o proportie de 90% , iar deconectarea liniilor a aparut in 5 din cele 10 intersectii posibile. In urma testelor parametrii optimi s-au dovedit a fii urmatorii: min\_wall\_width=15 si max\_wall\_width=25.



Schimbarea parametrilor poate duce la urmatoarele rezultate:



Intr-un final testul al doilea s-a incheiat cu succes in proportie de aproximativ 90%.