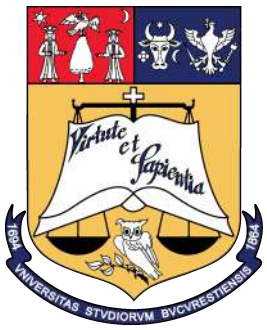


UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

SOLUȚIE PENTRU SUDOKU 3D, BAZATĂ PE COMPUTER VISION

Absolvent
Rusu Andrei-Cristian

Coordonator științific
Conf. Dr. Alexe Bogdan

București, iunie 2022

Rezumat

Vederea artificială este o subramură a inteligenței artificiale ce se află într-o continuă dezvoltare și care oferă posibilitatea de a analiza și a extrage informații din imagini digitale.

Lucrarea prezentă își propune dezvoltarea unei aplicații pentru pasionații de Sudoku, care le oferă posibilitatea de a găsi soluția unei configurații de Sudoku 3D folosind o imagine a acesteia. Acest proces pleacă de la o bază de date cu imaginile unor configurații de Sudoku 3D, scanate din cartea *The 3D Sudoku Puzzle by Parragon*. Imaginile sunt analizate și prelucrate folosind tehnici din domeniul vederii artificiale pentru a extrage informații despre configurații, apoi este folosit un algoritm de tip backtracking pentru a genera soluția.

Algoritmul final generează cu succes soluția pentru fiecare imagine din baza de date, într-un timp mediu de rulare de $\approx 1.27s$.

Limbajul de programare ales pentru implementarea soluției este *Python*, întrucat este un limbaj de nivel înalt care permite o dezvoltare rapidă, folosind librării precum *OpenCV* pentru prelucrarea de imagini și *NumPy* pentru operații matematice.

Abstract

Computer vision is a continuously growing sub-branch of artificial intelligence, which offers the possibility of analysing and extracting information from digital images.

This paper aims at building an application for people that are passionate about Sudoku, which offers them the possibility to find a solution for a Sudoku 3D configuration, using a scanned image of it. This process starts with a database of Sudoku 3D configurations, scanned from the book *The 3D Sudoku Puzzle by Parragon*. The images are analysed and processed using computer vision techniques in order to extract information about the configurations, then a backtracking algorithm is used in order to generate the solution.

The final algorithm can generate a solution for all the images in the database, in an average time of $\approx 1.27s$.

The programming language used for implementing the solution is *Python*, as it's a high level language that allows for fast development, using libraries such as *OpenCV* for image processing and *NumPy* for mathematical operations.

Cuprins

Capitolul 1

Introducere

1.1 Informații istorice

Sudoku este un joc de logică, apărut pentru prima dată la sfârșitul secolului al XIX-lea. Inițial, acesta avea reguli diferite, și a fost inspirat din *pătratele magice* [1], care sunt matrice de numere unde suma pe fiecare linie, fiecare coloană și ambele diagonale este aceeași (**Figura** *1.1*).

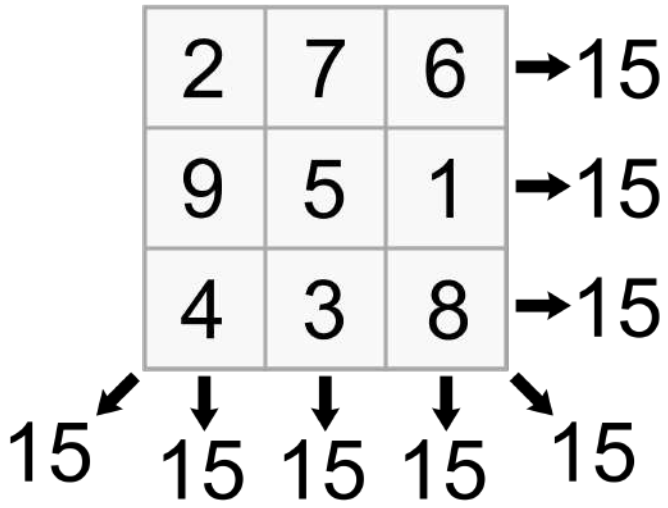
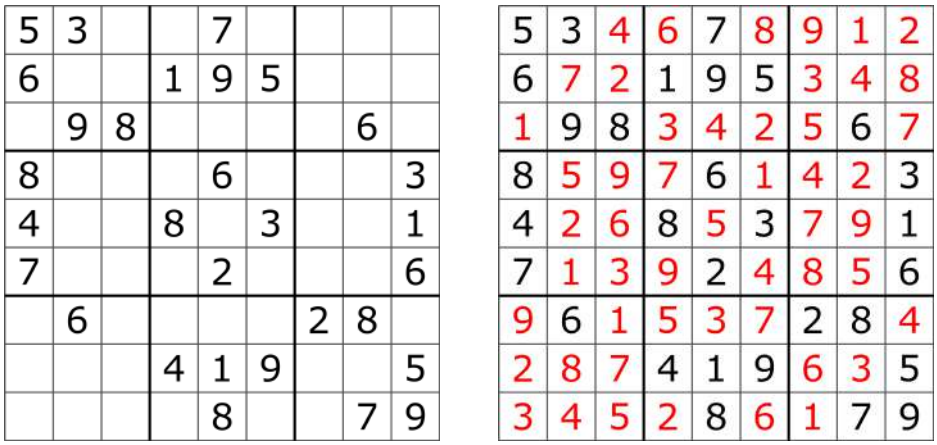


Figura 1.1: Exemplu de pătrat magic, unde suma liniilor, coloanelor și diagonalelor este 15

Varianta modernă a puzzle-ului Sudoku, de dimensiune 9×9, pe care o știm astăzi, a apărut pentru prima dată în anul 1984, în Japonia. Regulile jocului sunt următoarele:

- Fiecare celulă de dimensiune 3×3 trebuie să conțină toate cifrele de la 1 la 9, o singură dată

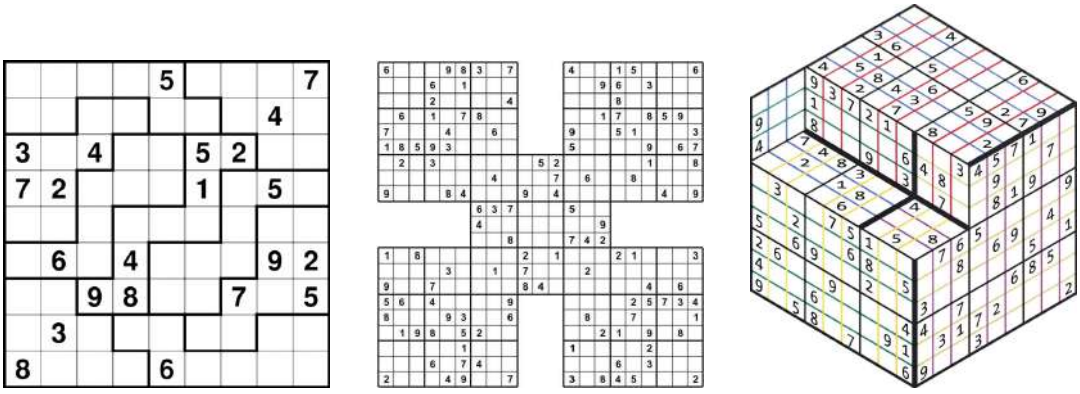
- Fiecare linie și fiecare coloană trebuie să conțină toate cifrele de la 1 la 9, o singură dată



(a) (b)

Figura 1.2: a) Exemplu de Sudoku clasic; b) soluția acestuia

În timp, au apărut variațiuni ale acestui puzzle, cu reguli similare, printre care se numără *Sudoku Jigsaw*, *Sudoku Samurai* și *Sudoku 3D*.



(a) (b) (c)

Figura 1.3: a) Sudoku Jigsaw; b) Sudoku Samurai; c) Sudoku 3D

1.2 Sudoku 3D

Această variațiune este reprezentată ca fiind un cub de dimensiune 3×3 cu vizibilitate către trei dintre fețele 9×9 ale acestuia și care conține un număr de cuburi mai mici. Constrângerile sunt similare cu cele ale variantei clasice, singura diferență fiind poziționarea fețelor 3×3. Acestea nu se mai află toate pe același plan, ci există *drumuri*, indicate prin culorile caroiajelor și separate de segmente negre îngroșate. În **Figura** *1.3c* se observă că pentru fiecare față de dimensiune 3×3, caroiajul este format din două culori. Acest lucru indică faptul că fiecare față face parte din două drumuri, iar pentru a rezolva această

variațiune, cifrele din fiecare față trebuie completate în raport cu cele două drumuri din care face parte.

În această lucrare implementăm un algoritm care identifică segmentele îngroșate, parcurge drumurile și extrage cifrele din fețe, iar soluția este generată cu un algoritm de tip backtracking.

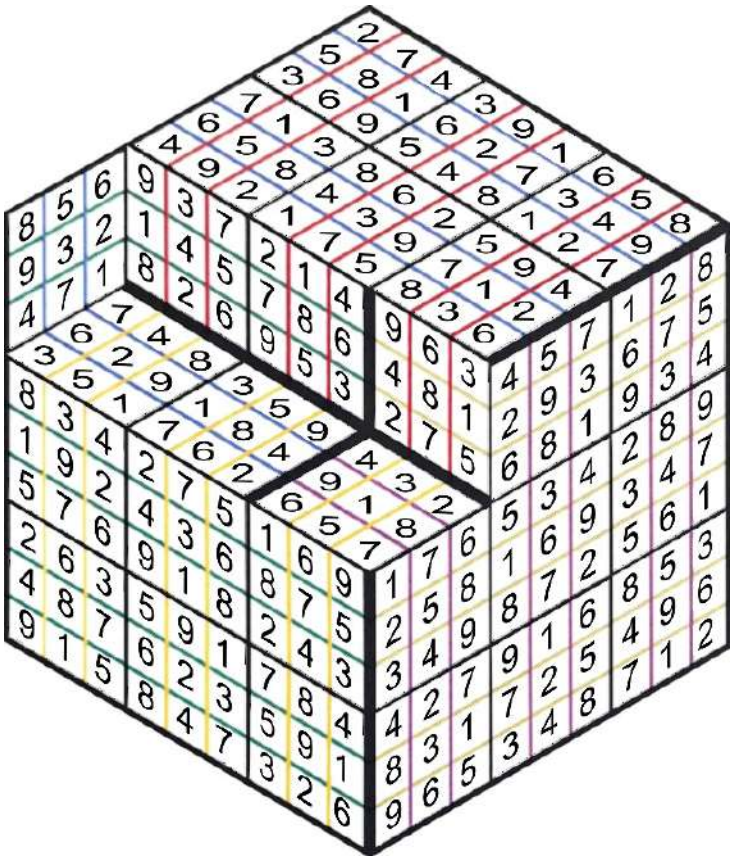
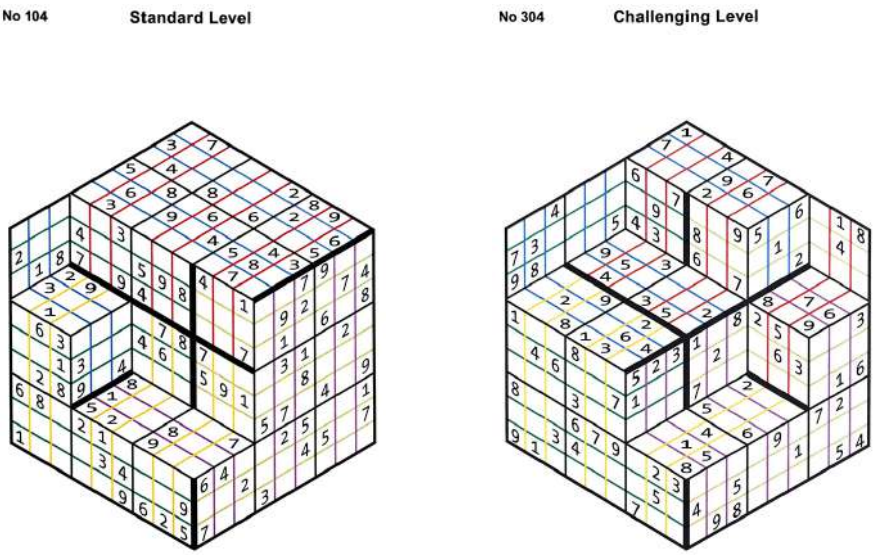


Figura 1.4: Soluția pentru configurația din **Figura** *1.3c*

Capitolul 2

Baza de date și preprocesarea datelor

Imaginile folosite în procesul dezvoltării algoritmului de rezolvare au fost preluate din cartea *The 3D Sudoku Puzzle by Parragon* [2], fiind scanate și transformate în imagini digitale. Baza de date constă în 325 de astfel de imagini, dintre care 163 având o dificultate standard, iar restul de 162 având un grad de dificultate mai ridicat. Diferența între acestea constă în numărul de cifre cunoscute inițial (configurațiile de dificultate standard având mai multe cifre înscrise).

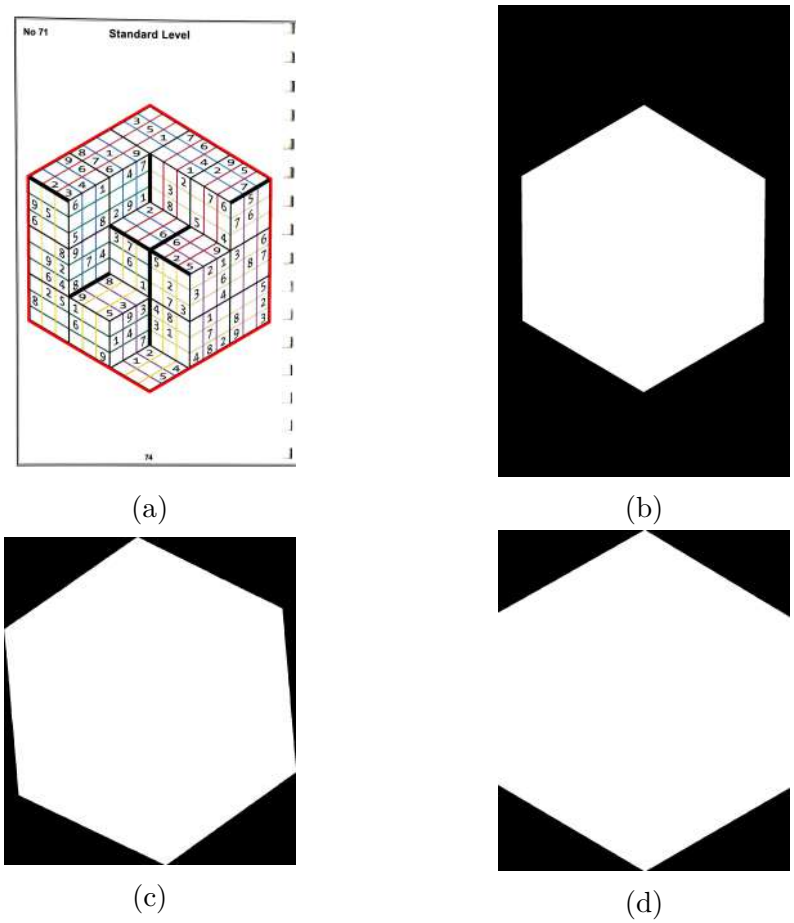


(a) (b)

Figura 2.1: Exemple de imagini din baza de date: a) configurație de dificultate standard (conținând 104 cifre); b) configurație de dificultate mai ridicată (conținând 98 de cifre)

Înainte de a putea utiliza imaginile, acestea sunt trecute printr-o etapă de preprocesare, care presu pune doi pași: îndreptarea liniilor exterioare ale cubului și extragerea conturului acestuia. Primul pas este necesar deoarece imaginile din baza de date nu sunt scanate

perfect, deci liniile exterioare ale cubului nu vor fi perfect paralele (deși liniile paralele sunt conservate). Pentru asta, implementăm un algoritm de corectare a înclinării. Având o imagine scanată a unei configurații de Sudoku 3D, acesta va identifica toate contururile [3] prezente în imagine, și va extrage conturul cubului, folosindu-se de faptul că acesta are aria cea mai mare (**Figura** 2.2a). Mai departe, creăm o mască [4] pe baza acestui contur (**Figura** *2.2b*), care este rotită, pe fiecare axă (x, y și z), cu unghiuri α, β, și respectiv $\gamma \in \{0°, 0.2°, 0.4°, ..., 2°\}$. Pentru fiecare imagine rotită, realizăm o detecție a conturului, iar imaginea este decupată la extremitățile conturului. Astfel, obținem o încadrare a cubului (**Figura** 2.2c). După ce am ajuns la această formă, putem aplica tehnica de *potrivire a șabloanelor* [5], cu masca unui contur încadrat perfect (2.2d).



(a) (b)

(c) (d)

Figura 2.2: a) conturul; b) masca pentru contur; c) masca rotită cu $\alpha = -2$, $\beta = -5$, $\gamma = 1$; d) șablonul

Imaginea cubului este redimensionată la imaginea șablonului, apoi este calculat

$$\text{coeficient de corelație} = \frac{|\{x \mid x = 255 \text{ și } x \in T \cap M\}|}{|\{x \mid x = 255 \text{ și } x \in T\}|} \tag{2.1}$$

unde T este imaginea șablonului, M este imaginea conturului iar x este un pixel. La final, aplicăm aceeași transformare folosind α, β și γ pentru care coeficienul de corelație a fost maxim. Imaginile pe care le folosim în continuare sunt imaginile decupate la contururile

cuburilor, după corectarea înclinării și redimensionarea lor.

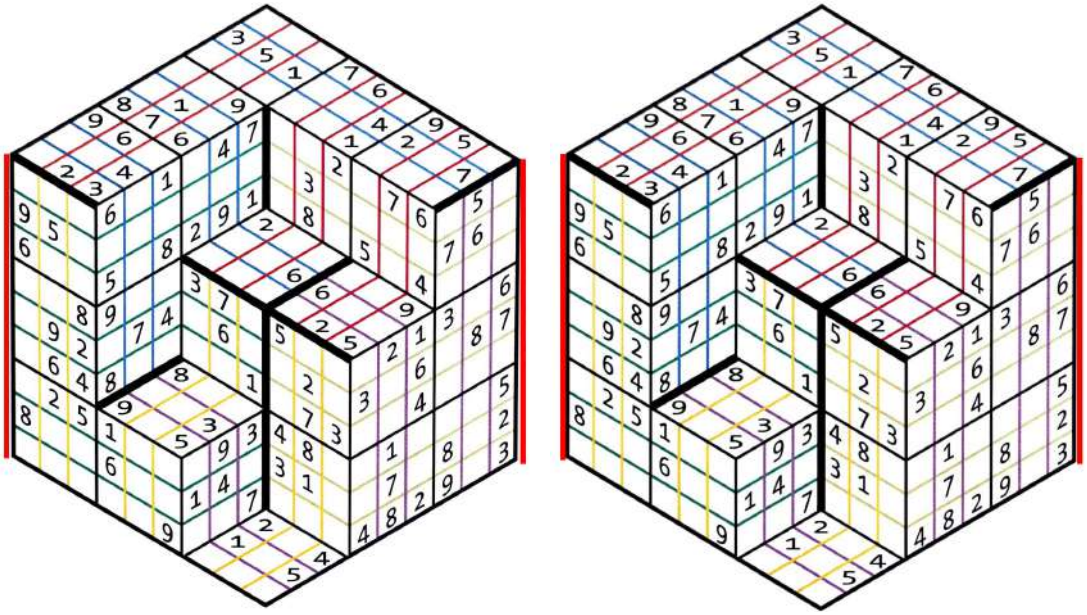


Figura 2.3: a) un exemplu de imagine înclinată; b) imaginea, după corectarea înclinării. Efectul se poate observa cel mai bine în partea de jos a liniilor roșii

Capitolul 3

Terminologie și convenții

3.1 Axele

O primă convenție este denumirea axelor cubului. În continuarea acestei lucrări, considerăm că fețele văzute din partea stângă se află pe axa X, cele văzute din partea dreaptă se află pe axa Y iar cele văzute de sus pe axa Z.

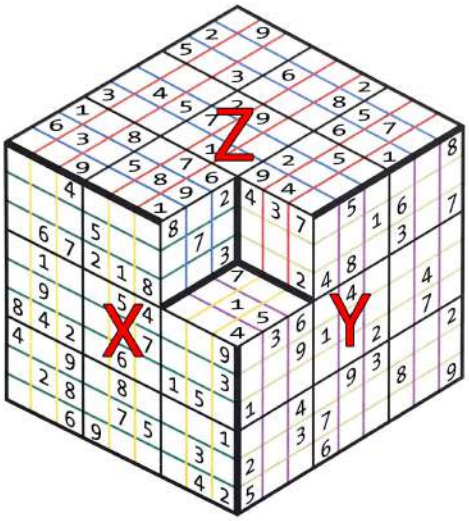


Figura 3.1: Axele fețelor unui cub

3.2 Notația punctelor fețelor

În procesul de extragere al fețelor notăm punctele acestora cu A, B, C și D, începând cu punctul din stânga sus (așa cum este văzută imaginea din față, pe axa respectivă) și parcurgându restul punctelor în sens invers trigonometric.

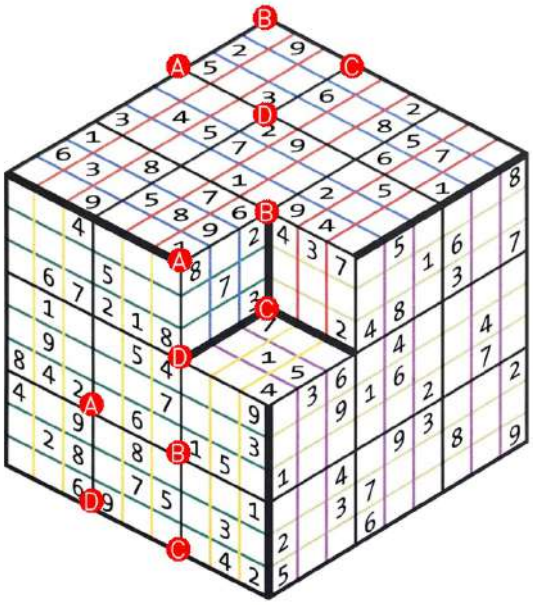
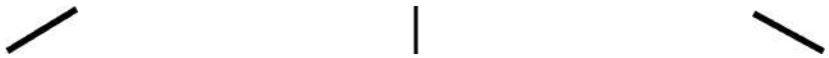


Figura 3.2: Punctele unei fețe de pe fiecare axă

3.3 Segmentele îngroșate

După cum am văzut în **Capitolul** *1*, în cub se regăsesc un număr de segmente negre îngroșate, care reprezintă *granițe* între drumuri. Acestea se găsesc în 3 variațiuni. Pe cele drepte le numim *segmente verticale*, pe cele înclinate la 60° (față de cele verticale) le numim *segmente de tip slash* iar pe cele înclinate la −60° le numim *segmente de tip backslash*.



(a) (b) (c)

Figura 3.3: a) Segment de tip slash; b) segment vertical; c) segment de tip backslash

Capitolul 4

Extragerea informațiilor din imagine

4.1 Calcularea coordonatelor vârfurilor cubului

Un prim pas în construirea soluției este identificarea vârfurilor cubului. Ținând cont de procesarea inițială a imaginilor, știm că acestea vor conține cuburi încadrate cât mai corect. Dându-se h = înălțimea și w = lățimea imaginii, acestea sunt calculate ușor, astfel:

- A = $(0, \frac{h}{4})$
- B = $(\frac{w}{2}, 0)$
- C = $(w, \frac{h}{4})$
- D = $(w, \frac{3 \cdot h}{4})$
- E = $(\frac{w}{2}, h)$
- F = $(0, \frac{3 \cdot h}{4})$
- G = $(\frac{w}{2}, \frac{h}{2})$

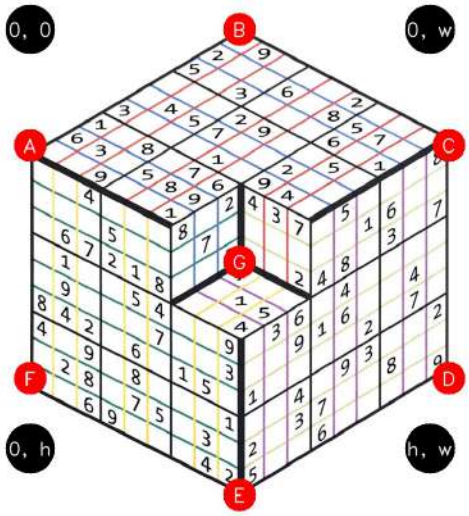


Figura 4.1: Vârfurile cubului (în roșu) și coordonatele imaginii (în negru)

4.2 Identificarea segmentelor îngroșate

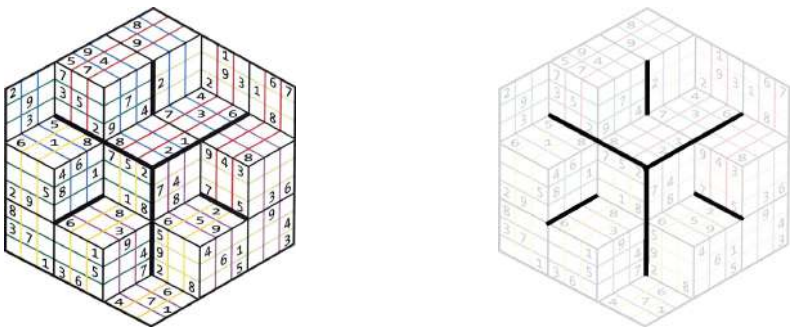
Pentru acest pas ne folosim de operația morfologică de *închidere* [6]. Cu alte cuvinte, construim un nucleu (*kernel*) de aceleași dimensiuni cu cele ale unui segment vertical. Se observă ușor că înălțimea segmentului vertical = $\frac{\text{înălțimea imaginii}}{6}$. Pentru a putea calcula lățimea, am observat că aceasta este direct proporțională cu înălțimea segmentului, iar factorul de scalare este $\simeq 0.09$. În continuare, adăugăm un factor de *eroare* dimensiunii nucleului - nu folosim dimensiunile exacte ale segmentului, ci mai întâi le scalăm cu un factor de 0.9, pentru a evita situațiile în care dimensiunile nucleului pot fi cu un număr de pixeli mai mari decât al segmentelor din imagine - caz în care operația de închidere nu mai păstrează nimic din imagine. În continuare, aplicăm o operație de închidere urmată de o operație de thresholding [7].

O observație importantă este legată de unghiul segmentului - în cazul unei imagini cu un cub perfect aliniat, segmentele ar fi perfect verticale. În realitate, deși imaginile au fost preprocesate, nu toate au fost îndreptate perfect - deci, segmentele pot prezenta o înclinație de $\pm 3°$. Pentru a asigura detectarea lor chiar și în situațiile în care imaginea este înclinată, vom aplica algoritmul descris anterior cu același nucleu rotit, pe rând, cu $\alpha \in [-3, 3]$. Imaginea finală va fi aceea pentru care cantitatea de pixeli negri este cea mai mare (ceea ce înseamnă că am identificat segmentele cât mai corect).

Detaliile descrise anterior funcționează în mod analog și pentru segmentele de tip *slash* și *backslash*. Acestea pot fi considerate segmente verticale, rotite la $60°$, respectiv $-60°$.

(a) (b) (c) (d) (e) (f)

Figura 4.2: Exemplu de identificare al segmentelor de tip slash;
a) 57° - 0.18%; b) 58° - 0.65%; c) 59° - 0.69%; d) 60° - 0.66%; e) 61° - 0.14%; f) 62° - 0.0%; în acest caz, vom alege c).



(a) (b)

Figura 4.3: a) Imaginea unei configurații; b) segmentele extrase, scoase în evidență

4.3 Extragerea vectorilor de deplasare a fețelor

Având ca informație punctul unei fețe (spre exemplu, punctul A) și axa pe care se află aceasta, avem nevoie să calculăm celelalte puncte (B, C și D). Soluția implementată în această lucrare calculează, pentru fiecare axă de coordonate, vectorii de deplasare (pe verticală și pe orizontală) a fețelor, folosindu-se de vârfurile conturului cubului. Vom folosi, pentru fiecare axă, trei vârfuri: cel din stânga sus (de referință), cel din dreapta sus (pentru a calcula deplasarea pe orizontală) și cel din stânga jos (pentru a calcula deplasarea pe verticală). Deci, pentru calculul vectorilor pe axa X folosim vârfurile A, G și F; pentru axa Y folosim G, C și E, iar pentru axa Z folosim A, B și G.

$$\begin{cases} \overrightarrow{D_o} = \overrightarrow{\text{deplasarea pe orizontală}} = \overrightarrow{\text{colțul din stânga sus}} - \overrightarrow{\text{colțul din dreapta sus}} \\ \overrightarrow{D_v} = \overrightarrow{\text{deplasarea pe verticală}} = \overrightarrow{\text{colțul din stânga sus}} - \overrightarrow{\text{colțul din stânga jos}} \end{cases} \tag{4.1}$$

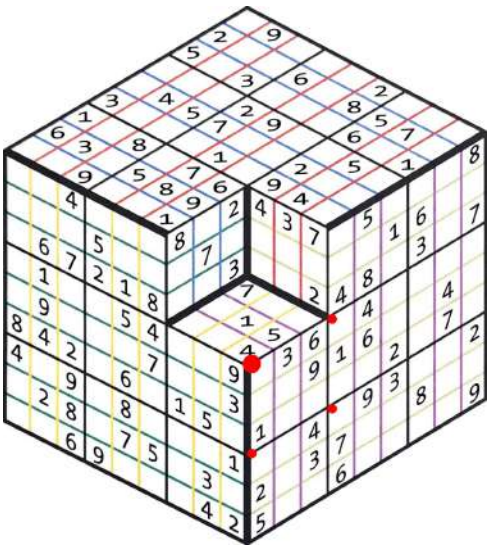
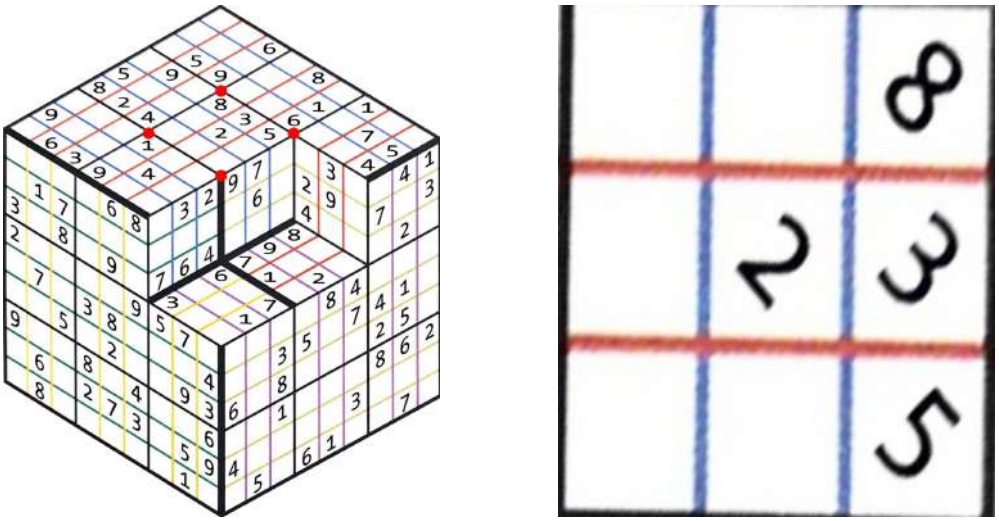


Figura 4.4: Un exemplu de calcul al punctelor pentru o față de pe axa Y; punctul roșu îngroșat este A, $B = A + \overrightarrow{D_o}$, $C = A + \overrightarrow{D_o} + \overrightarrow{D_v}$, $D = A + \overrightarrow{D_v}$.

4.4 Determinarea existenței unei fețe definite de patru puncte și extragerea acesteia

Odată ce putem calcula punctele unei fețe din cub, aplicăm o transformare de perspectivă [8] pentru a aduce imaginea din forma actuală (văzută din perspectivă) într-o formă văzută din față.

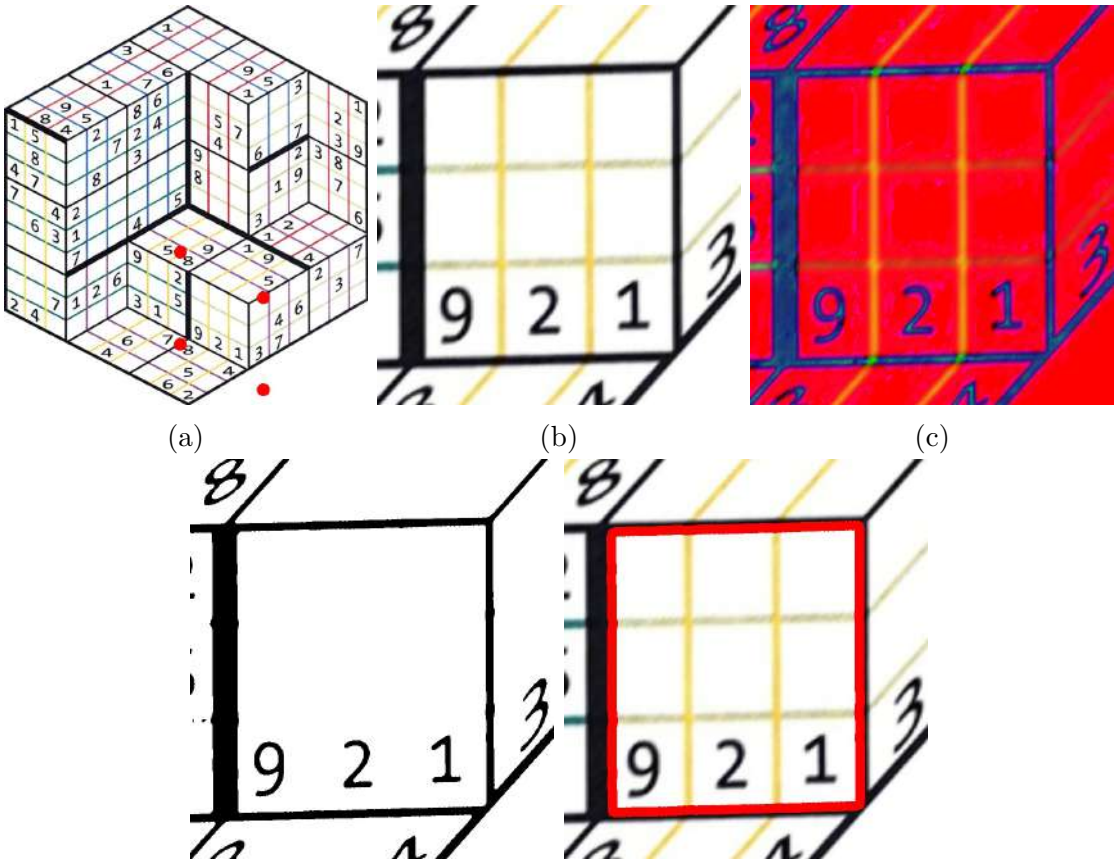


(a) (b)

Figura 4.5: Exemplu de transformare când imaginea rezultată din decuparea la punctele date formează o față; a) punctele feței; b) imaginea rezultată

După cum putem observa în **Figura** *4.5b*, extragerea directă a feței la punctele calculate va rezulta într-o imagine rotită cu câteva grade, din cauza înclinării inițiale a imaginii. Pentru a rezolva această problemă și a extrage fața cât mai corect, înainte de a aplica transformarea de perspectivă asupra punctelor fețelor, le vom adăuga un *padding* (pentru a ne asigura că reușim să încadrăm fața - **Figura** *4.6b*). În continuare, avem nevoie să găsim conturul feței, iar pentru asta, aducem imaginea din spațiul de culori RGB (*Red, Green, Blue*) în spațiul HSV (*Hue, Saturation, Value*) (**Figura** *4.6c*) și păstrăm doar pixelii pentru care $0 \leq hue \leq 360$, $0 \leq saturation \leq 255$ și $0 \leq value \leq 100$ (**Figura** *4.6d*). Aceste valori definesc o regiune de culori foarte apropiate de negru - astfel, reușim să eliminăm careul din interiorul feței și să găsim conturul (**Figura** *4.6e*).

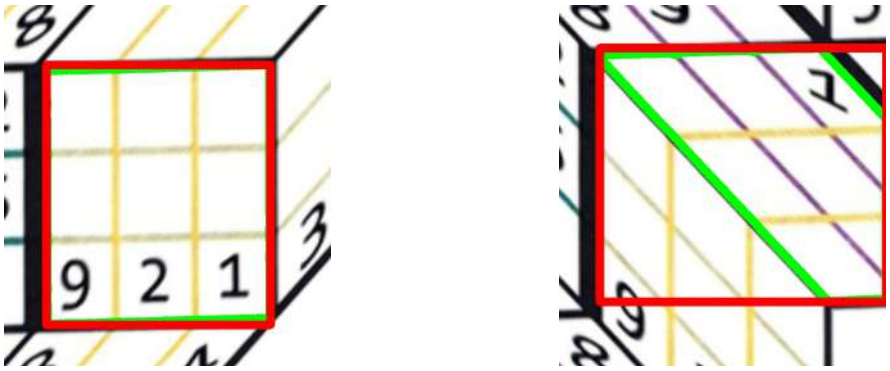
În continuare, trebuie să decidem dacă acest contur este al unei fețe. Determinăm punctele unui dreptunghi care încadrează conturul și calculăm raportul $r = \frac{\text{aria conturului}}{\text{aria dreptunghiului}}$. Dacă acesta este mai mare de 90% știm că dreptunghiul are o formă foarte similară cu cea a conturului - în acest caz, considerăm că este al unei fețe.



(a) (b) (c)

(d) (e)

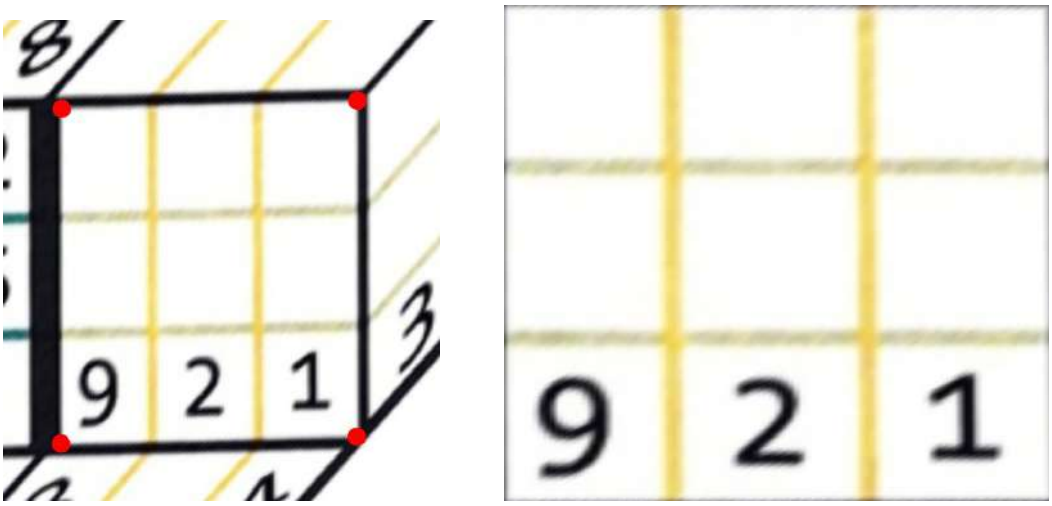
Figura 4.6: a) Punctele unei fețe, după adăugarea unui padding; b) imaginea rezultată în urma aplicării transformării de perspectivă; c) imaginea HSV asociată; d) imaginea rezultată în urma păstrării pixelilor negri; e) conturul feței



(a) (b)

Figura 4.7: Dreptunghiul (roșu) care încadrează conturul (verde); a) cazul în care există o față; b) cazul în care nu există o față

În cazul în care conturul este al unei fețe, calculăm punctele acestuia și extragem fața, aplicând o transformare de perspectivă.



(a) (b)

Figura 4.8: a) Punctele feței; b) imaginea extrasă în urma aplicării transformării de perspectivă

4.5 Extragerea drumurilor folosind segmentele îngroșate

După cum am observat, fiecare segment îngroșat reprezintă o graniță între două fețe, de unde pornesc două drumuri. Soluția implementată folosește segmentele îngroșate pentru a calcula coordonatele unui punct pentru prima față de pe fiecare drum. Mai departe, se folosește de vectorii de deplasăre și de algoritmul de extragere a fețelor pentru a putea calcula punctele feței și a o extrage (dacă există). Pentru a parcurge întregul drum, implementăm un algoritm recursiv.

4.5.1 Extragerea punctelor de început ale fețelor

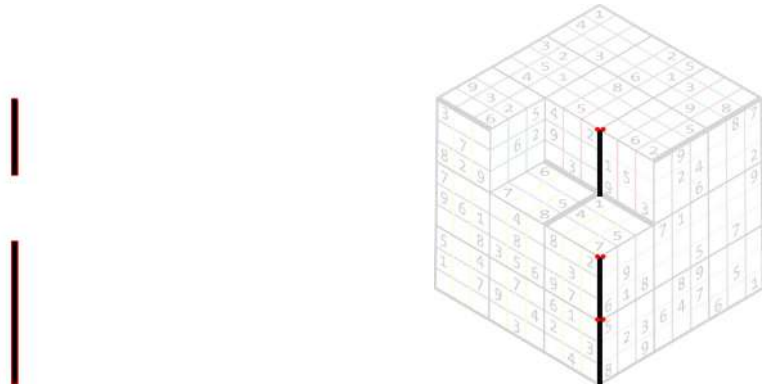
Distingem două cazuri în care algoritmul se comportă diferit: atunci când avem segmente verticale și atunci când avem segmente de tip slash sau segmente de tip backslash. Partea comună în ambele cazuri este detectarea contururilor acestora. Pentru fiecare contur, algoritmul caută extremitățile pe verticală ale acestuia, notate cu T (*top*), respectiv B (*bottom*), calculează numărul de segmente unite în contur și generează punctele algoritmic, folosindu-se de dimensiunile segmentelor.

$$\begin{aligned}
\text{distanța segmentului} &= \sqrt{(\text{înălțimea segmentului})^2 + (\text{lățimea segmentului})^2} \\
\text{distanța conturului} &= \sqrt{(x_T - y_T)^2 + (x_B - y_B)^2} \\
\text{numărul de segmente unite} &= round(\frac{\text{distanța conturului}}{\text{distanța segmentului}})
\end{aligned} \tag{4.2}$$

În cazul segmentelor verticale, pentru fiecare contur, considerăm punctul de început ca fiind T. Plecând de la acesta, construim punctele pe partea stângă și pe partea dreaptă, astfel:

$$\begin{aligned}
\text{punctul din stânga} &= (x_T, y_T + i \cdot (\text{înălțimea segmentului})) \\
\text{punctul din dreapta} &= (x_T - (\text{lățimea segmentului}), y_T + i \cdot (\text{înălțimea segmentului}))
\end{aligned} \tag{4.3}$$

unde i este indicele segmentului curent din contur, $i \in [0, \text{numărul de segmente unite})$.



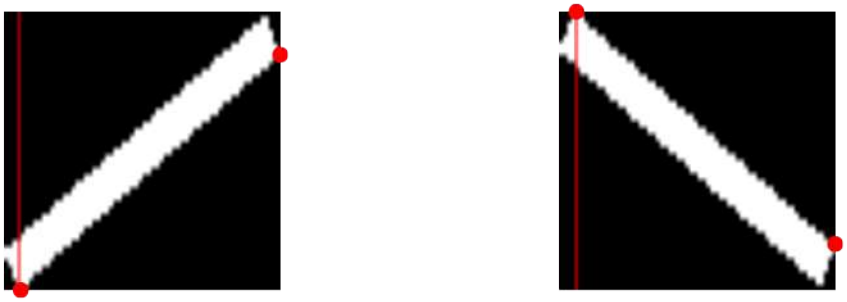
(a) (b)

Figura 4.9: a) Contururile din imaginea cu segmentele verticale; b) punctele calculate, scoase în evidență

În cazul segmentelor de tip slash și de tip backslash nu mai putem calcula distanța până la următorul punct în mod liniar - folosim ecuația dreptei, având cunoscute unghiul de înclinație ($30°$ pentru segmentele de tip slash, respectiv $-30°$ pentru cele de tip backslash) și punctul de început. Astfel, pentru fiecare contur, calculăm punctele în felul următor:

$$
\begin{aligned}
P = \text{punctul de început} &= \begin{cases} T, & \text{segmentul este de tip backslash} \\ B, & altfel \end{cases} \\
\text{semnul direcției} &= \begin{cases} 1, & \text{segmentul este de tip backslash} \\ -1, & altfel \end{cases} \\
m &= \tan\left(30^\circ \cdot (\text{semnul direcției})\right) \\
c &= y_P - m * x_P \\
\text{punctul pentru primul drum} &= P \\
\text{punctul pentru al doilea drum} &= (x_P, y_P - (\text{semnul direcției}) * (\text{lățimea segmentului vertical})) \\
x &= x_P + 0.96 * (\text{lățimea segmentului}) \\
P &= (x, y(x))
\end{aligned}
\tag{4.4}
$$

unde P este calculat pentru fiecare segment din contur.



(a) (b)

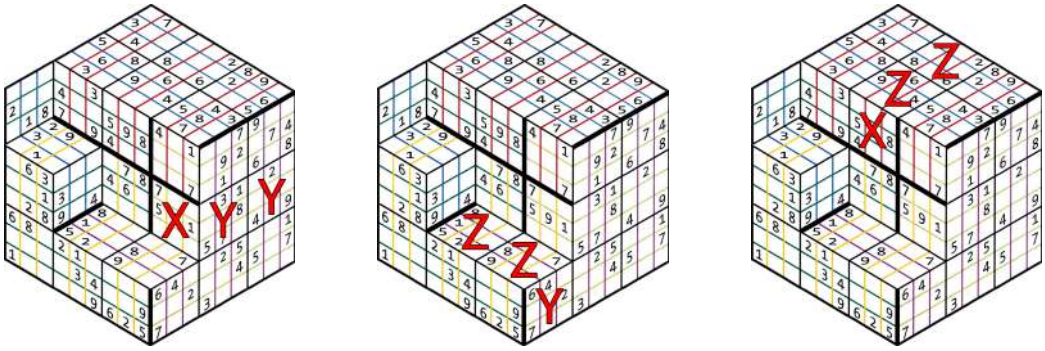
Figura 4.10: Explicația factorului de 0.96 în formula de calcul al punctului; se poate observa că, fiind date înălțimea și lățimea segmentului, următorul punct nu se află în capătul orizontal al imaginii; a) exemplu pentru un segment de tip slash; b) exemplu pentru un segment de tip backslash

(a) (b)

Figura 4.11: Exemplu de puncte calculate de algoritm; a) pentru segmente de tip slash; b) pentru segmente de tip backslash

4.5.2 Algoritmul

O observație importantă este aceea că fețele dintr-un drum se pot afla pe cel mult două axe diferite. Spre exemplu, pentru drumurile care pleacă de la un segment vertical, acestea conțin fețe doar de pe axele X și Y (**Figura** *4.12a*). Pentru segmentele de tip slash, drumurile conțin fețe doar de pe axele Z și Y (**Figura** *4.12b*). Pentru cele de tip backslash, ele conțin fețe doar de pe axele X și Z (**Figura** *4.12c*).



(a) (b) (c)

Figura 4.12: Exemplu de axe pe care se pot afla fețele dintr-un drum care pleacă de la un tip de segment; a) cazul segmentelor verticale, unde fețele se pot afla pe axele X și Y; b) cazul segmentelor de tip slash, unde fețele se pot afla pe axele Z și Y; c) cazul segmentelor de tip backslash, unde fețele se pot afla pe axele X și Z

Astfel, construim un algoritm recursiv care pleacă de la un punct extras dintr-un segment îngroșat, calculează punctele feței aflate pe prima din cele două axe și încearcă extragerea ei. Dacă fața nu există, calculează punctele feței de pe cea de-a doua axă și o extrage (fiind garantat că există). După ce identifică punctele feței, se folosește de acestea pentru a se deplasa în continuare pe drum. Recursivitatea se oprește atunci când algoritmul a găsit toate cele 3 fețe.

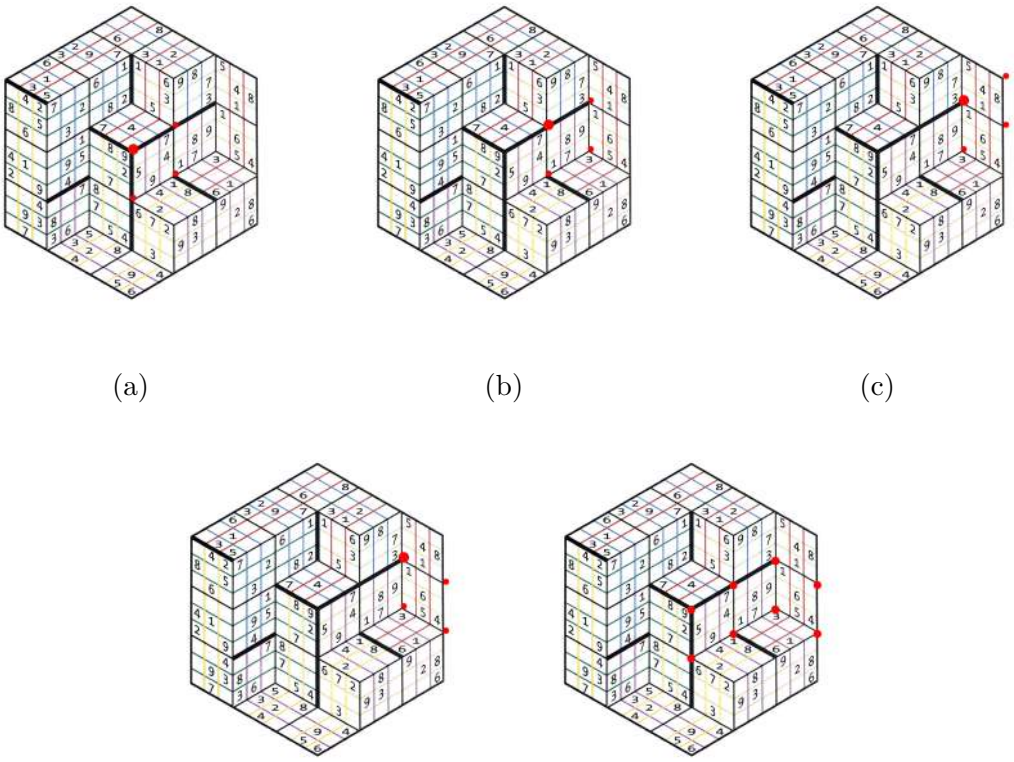
4.5.3 Formulele pentru calculul punctelor fețelor din drumuri

Amintim faptul că $\overrightarrow{D_o}$ reprezintă vectorul de deplasare pe orizontală, iar $\overrightarrow{D_v}$ reprezintă vectorul de deplasare pe verticală pentru o axă. De asemenea, reamintim formulele pentru calculul punctelor unei fețe, dat find punctul A:

$$\begin{aligned} B &= A + \overrightarrow{D_o} \\ C &= A + \overrightarrow{D_o} + \overrightarrow{D_v} \\ D &= A + \overrightarrow{D_v} \end{aligned} \tag{4.5}$$

Cazul punctelor din dreapta segmentelor verticale

Punctul de început este A (punctul din stânga sus al feței). Următorul punct de început va fi vârful B al feței găsite. Punctele sunt calculate folosind vectorii de deplasăre pentru axele X și Y. Aplicăm direct **Ecuațiile** *4.5*.



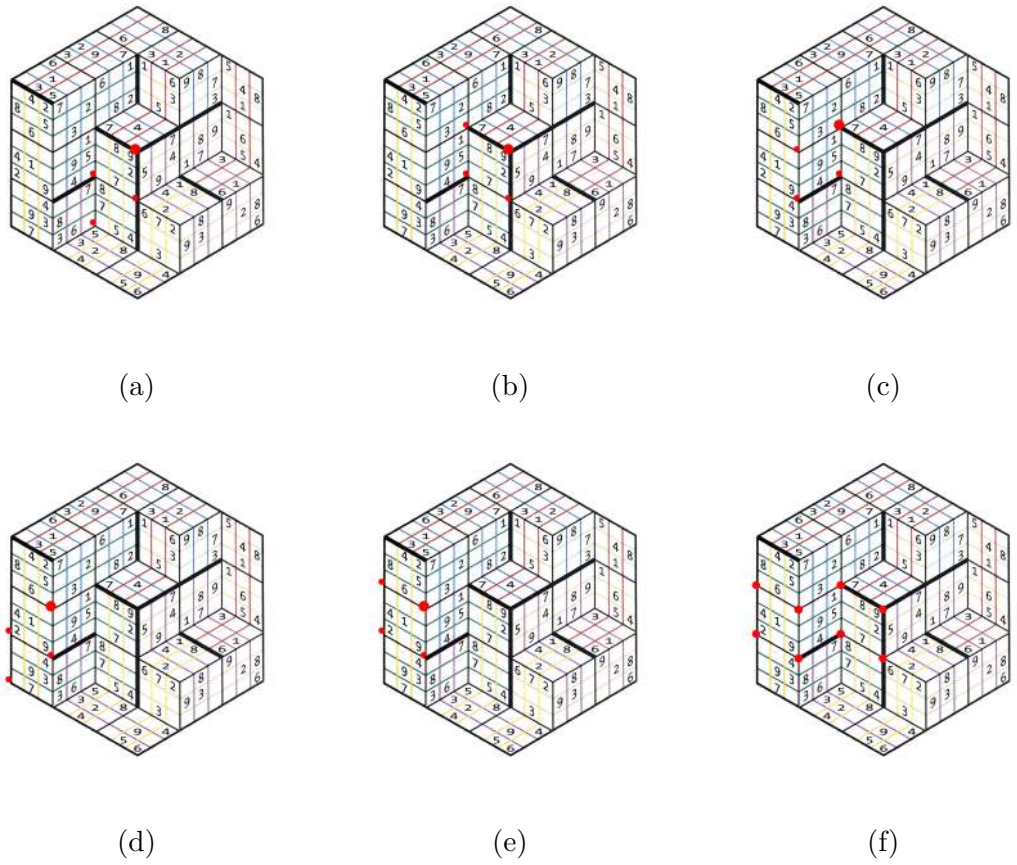
(a) (b) (c)

(d) (e)

Figura 4.13: Exemplu de parcurgere plecând de la un punct de pe partea dreaptă a unui segment vertical; a) prima față, identificată corect pe axa Y; b) a doua față, identificată corect pe axa Y; c) a treia față, identificată eronat pe axa Y; d) a treia față identificată corect pe axa X; e) drumul complet

Cazul punctelor din stânga segmentelor verticale

Punctul de început va fi B. Următorul punct va fi întotdeauna punctul A al feței găsite. Punctele sunt calculate folosind vectorii de deplasare pentru axele X și Y. Calculăm $A = B - \overrightarrow{D_o}$ iar apoi aplicăm **Ecuațiile** *4.5*.



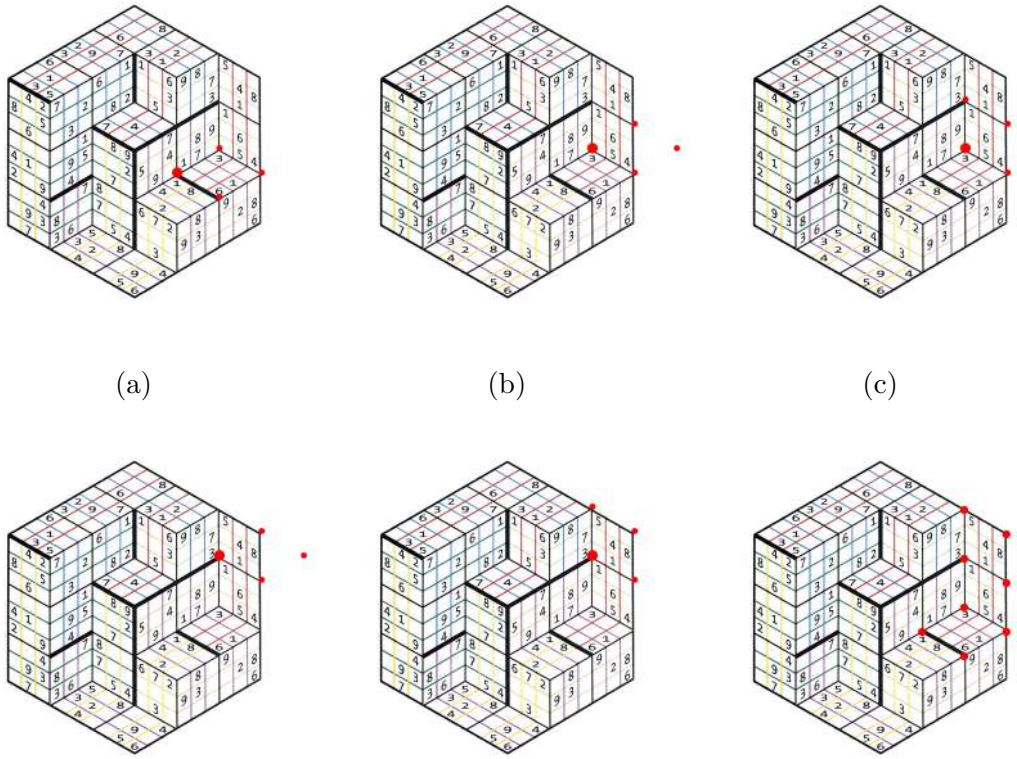
(a) (b) (c)

(d) (e) (f)

Figura 4.14: Exemplu de parcurgere plecând de la un punct de pe partea stângă a unui segment vertical; a) prima față, identificată eronat pe axa Y; b) prima față, identificată corect pe axa X; c) a doua față, identificată corect pe axa Y; d) a treia față, identificată eronat pe axa Y; e) a treia față, identificată corect pe axa X; f) drumul complet

Cazul punctelor din dreapta segmentelor de tip backslash

Punctul de început va fi A, în cazul unei fețe care se află pe axa Z și D pentru o față care se află pe axa X. Următorul punct va fi ales în funcție de axa pe care se află fața - acesta este B, dacă fața se află pe axa Z, altfel este A. Punctele sunt calculate folosind vectorii de deplasare pentru axele X și Z. Pentru cazul în care punctul de început este D, calculăm $A = D - \overrightarrow{D_v}$ și aplicăm **Ecuațiile** *4.5*.



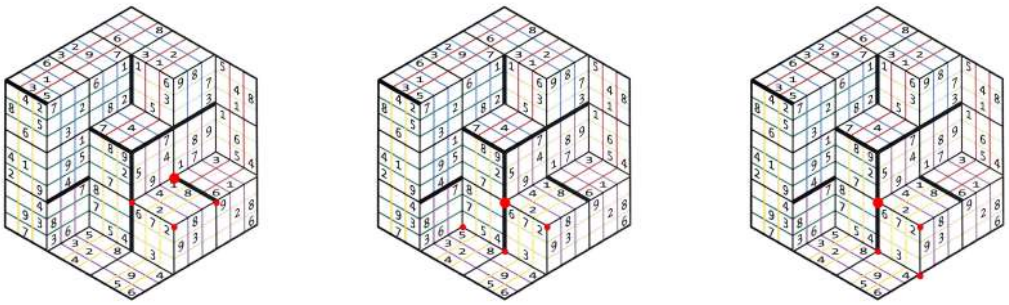
(a) (b) (c)

(d) (e) (f)

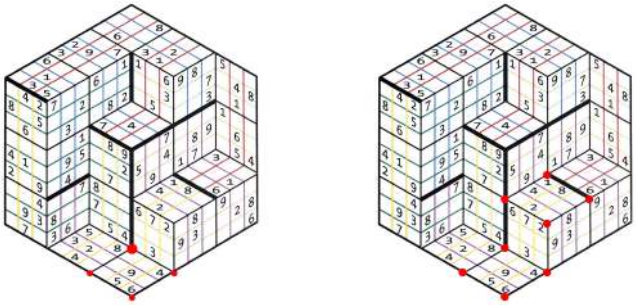
Figura 4.15: Exemplu de parcurgere plecând de la un punct din dreapta unui segment de tip backslash; a) prima față, identificată corect pe axa Z; b) a doua față, identificată eronat pe axa Z; c) a doua față, identificată corect pe axa X; d) a treia față, identificată eronat pe axa Z; e) a treia față, identificată corect pe axa X; f) drumul complet

Cazul punctelor de jos ale segmentelor de tip backslash

Punctul de început este B, în cazul unei fețe care se află pe axa Z, și A dacă fața se află pe axa X. Următorul punct este calculat în funcție de axa pe care se află fața, acesta fiind A, dacă axa este Z, altfel este D. Punctele sunt calculate folosind vectorii de deplasare pentru axele Z și X.



(a) (b) (c)

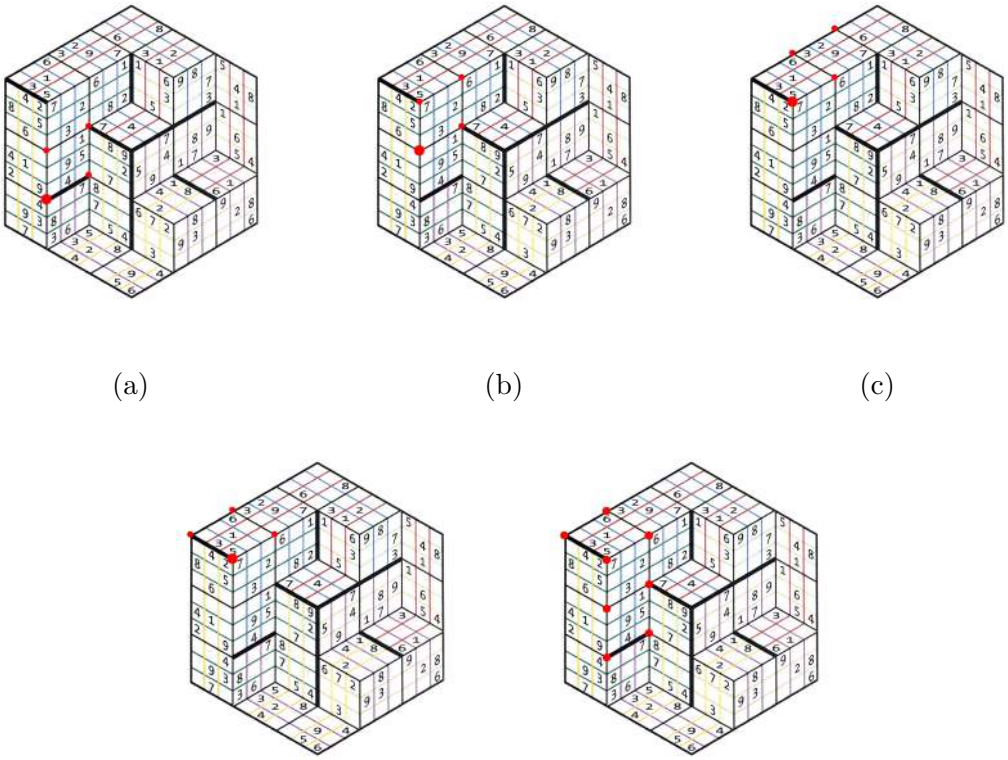


(d) (e)

Figura 4.16: Exemplu de parcurgere plecând de la un punct din josul unui segment de tip backslash; a) prima față, identificată corect pe axa Z; b) a doua față, identificată eronat pe axa Z; c) a doua față, identificată corect pe axa X; d) a treia față, identificată corect pe axa Z; e) drumul complet

Cazul punctelor de sus ale segmentelor de tip slash

Punctul de început este D. Următorul punct de început este punctul A al feței găsite. Punctele sunt calculate folosind vectorii de deplasare pentru axele Z și Y.



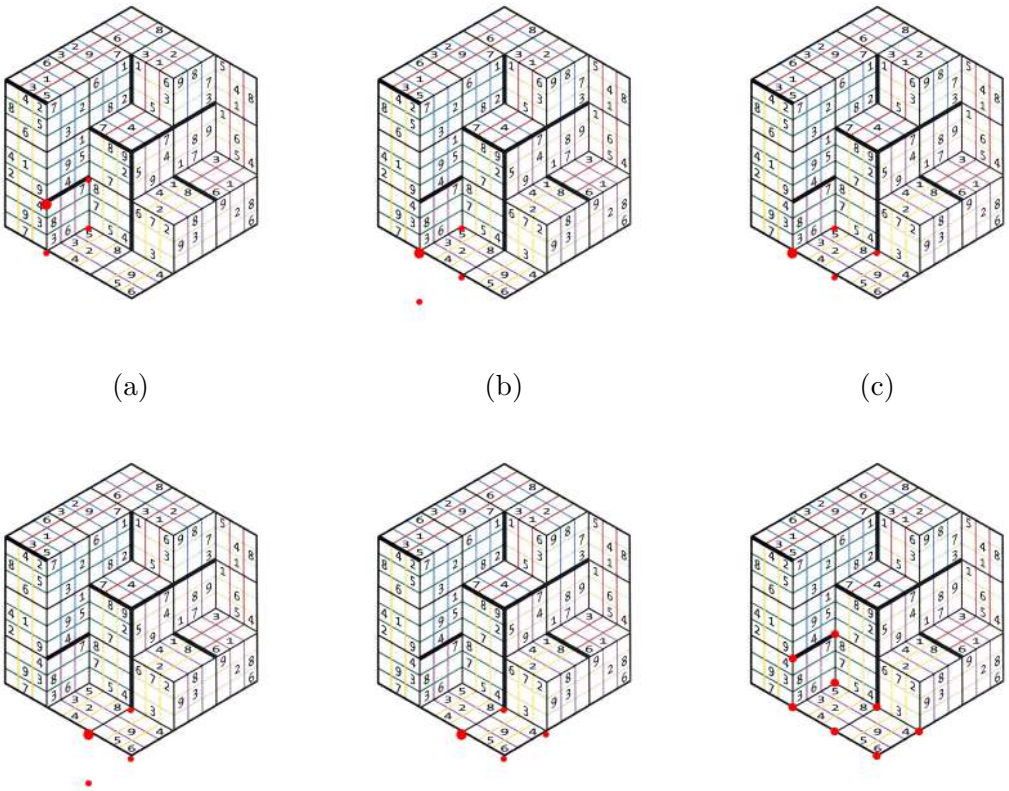
(a) (b) (c)

(d) (e)

Figura 4.17: Exemplu de parcurgere plecând de la un punct de deasupra unui segment de tip slash; a) prima față, identificată corect pe axa Y; b) a doua față, identificată corect pe axa Y; c) a treia față, identificată eronat pe axa Y; d) a treia față, identificată corect pe axa Z; e) drumul complet

Cazul punctelor de jos ale segmentelor de tip slash

Punctul de început este A, iar punctul următor este punctul D al feței identificate. Punctele sunt calculate folosind vectorii de deplasare pentru axele Z și Y.



(a) (b) (c)

(d) (e) (f)

Figura 4.18: Exemplu de parcurgere plecând de la un punct din josul unui segment de tip slash; a) prima față, identificată corect pe axa Y; b) a doua față, identificată eronat pe axa Y; c) a doua față, identificată corect pe axa Z; d) a treia față, identificată eronat pe axa Y; e) a treia față, identificată corect pe axa Z; f) drumul complet

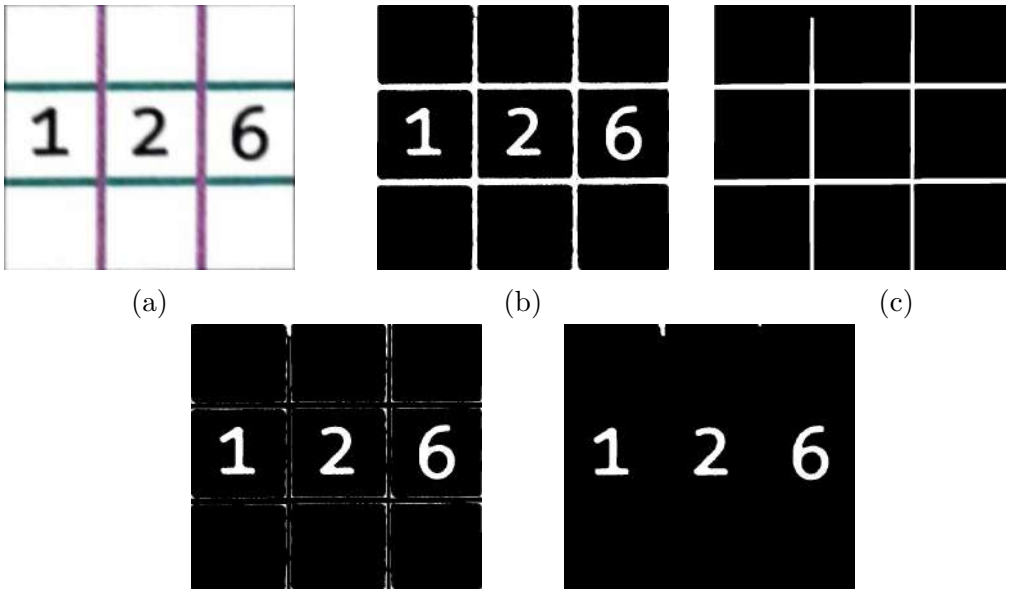
4.6 Extragerea cifrelor din imagini cu fețe

Având o imagine cu o față, folosim tehnica ferestrei glisante (*sliding window*) [9] pentru a parcurge imaginea de la stânga la dreapta și de sus în jos și a extrage informații din fiecare celulă.

4.6.1 Preprocesarea imaginii

Un prim pas este redimensionarea imaginii astfel încât să fie pătrată și transformarea ei la tonuri de gri (*grayscale*), urmată de aplicarea unei operații de thresholding (**Figura** *4.19b*). Extragem caroiajul, aplicând două operații morfologice de deschidere, prima folosind un nucleu de dimensiune $(\frac{\text{înălțimea imaginii}}{3}, 1)$ pentru segmentele verticale, iar cea

de-a doua folosind un nucleu de dimensiune $(1, \frac{\text{lățimea imaginii}}{3})$ pentru segmentele orizontale (**Figura** *4.19c*). În continuare, păstrăm în imagine pixelii care nu fac parte din caroiaj, obținând astfel o imagine care conține doar cifrele feței și, eventual, zgomot de fundal (**Figura** *4.19d*). Pentru diminuarea acestuia, aplicăm o operație morfologică de deschidere cu un nucleu de dimensiune $(4, 4)$ (**Figura** *4.19e*).



(a) (b) (c)

(d) (e)

Figura 4.19: a) Imaginea inițială; b) imaginea după redimensionare, conversie la tonuri de gri și aplicarea operației de thresholding; c) caroiajul extras; d) imaginea după eliminarea pixelilor ce fac parte din caroiaj; e) imaginea după eliminarea zgomotului de fundal

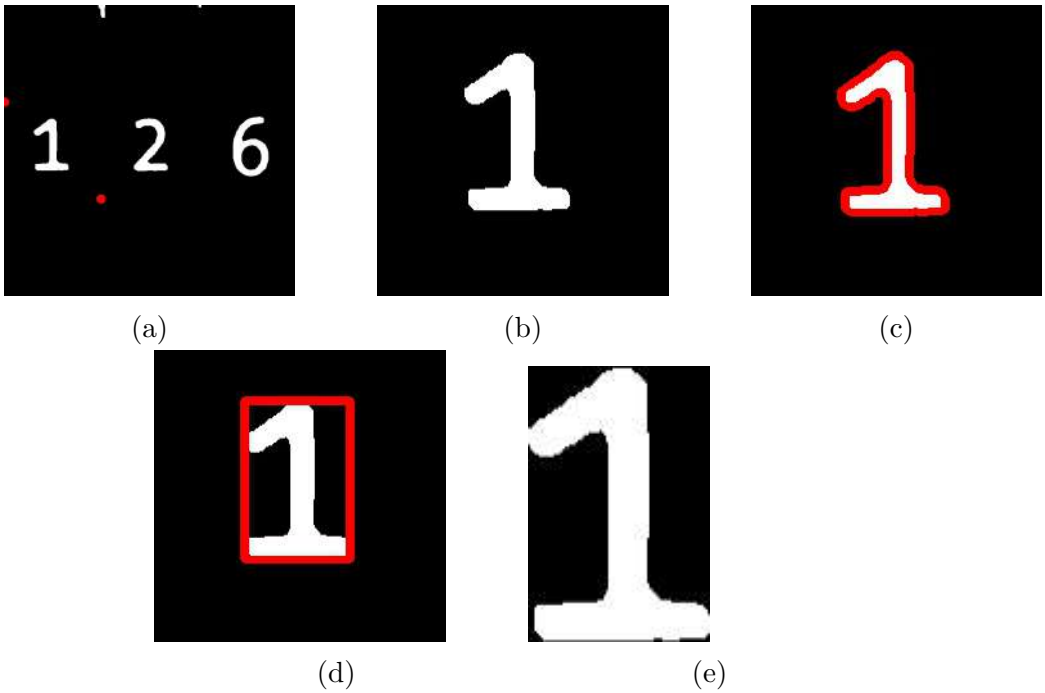
4.6.2 Extragerea informației din celule

Parcurgem imaginea folosind o fereastră de dimensiune $(\frac{\text{înălțimea imagini}}{3}, \frac{\text{lățimea imaginii}}{3})$ (exemplu: **Figura** *4.20a*). Pentru fiecare fereastră determinăm dacă există o cifră sau nu, folosind procentajul de pixeli albi din imagine - dacă este mai mic de 5%, știm că imaginea conține cel mult doar zgomot de fundal (**Figura** *4.20b*).

(a) (b)

Figura 4.20: a) Punctele din stânga sus și dreapta jos ale ferestrei; b) fereastra, cu procentajul de pixeli albi de 0.1%

În caz contrar, căutăm conturul cifrei (**Figura** *4.21c*) și calculăm punctele unui dreptunghi care încadrează acest contur (**Figura** *4.21d*). Imaginea cifrei rezultă din decuparea imaginii la vârfurile dreptunghiului (**Figura** *4.21e*).

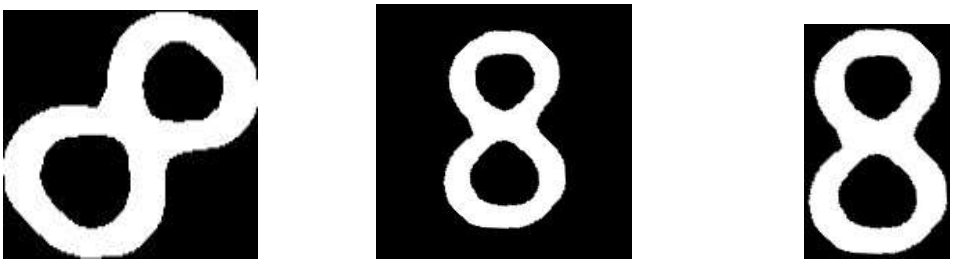


(a) (b) (c)

(d) (e)

Figura 4.21: a) Punctele din stânga sus și dreapta jos ale ferestrei; b) fereastra; c) conturul cifrei; d) dreptunghiul care încadrează conturul; e) cifra extrasă

De asemenea, ținem cont de faptul că fețele de pe axa Z au cifrele rotite la $-45°$, motiv pentru care, după ce extragem cifra, o rotim la $45°$ și aplicăm același procedeu pentru a o reîncadra.



(a) (b) (c)

Figura 4.22: a) Imaginea unei cifre extrase de pe axa Z; b) imaginea după o rotație la $45°$; c) imaginea după reîncadrare

4.6.3 Clasificarea cifrelor din imagini

Pentru acest pas, vom folosi tehnica de potrivire a șabloanelor. Folosind un set de imagini ce conțin cifre (extrase din imagini cu fețe 3×3 ale cuburilor din baza de date)

putem calcula potrivirea între o imagine ce încadrează o cifră și fiecare imagine șablon, clasificarea finală fiind dată de cifra pentru care potrivirea a fost cea mai mare.

Spre exemplu, pentru imaginea din **Figura** *4.22c*, obținem următoarele valori:

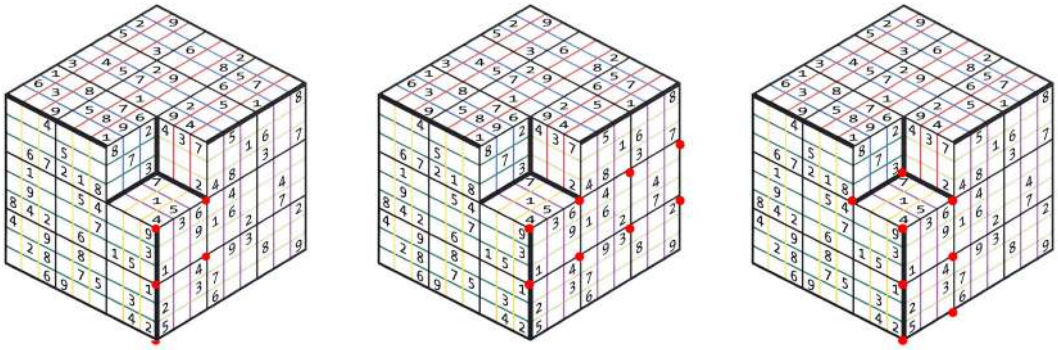
- 1: 0.07
- 2: 0.24
- 3: 0.49
- 4: 0.05
- 5: 0.38
- 6: 0.48
- 7: 0.09
- **8: 0.92**
- 9: 0.34

astfel, clasificând imaginea ca fiind cifra 8.

Capitolul 5

Generarea soluției

Pentru început, reamintim regulile jocului. Fiecare față de dimensiune 3×3 trebuie să conțină toate cifrele de la 1 la 9. De asemenea, fiecare față trebuie rezolvată în raport cu cele două drumuri din care face parte (**Figura** *5.1*), iar toate liniile și coloanele din fiecare drum trebuie să conțină toate cifrele de la 1 la 9.



(a) (b) (c)

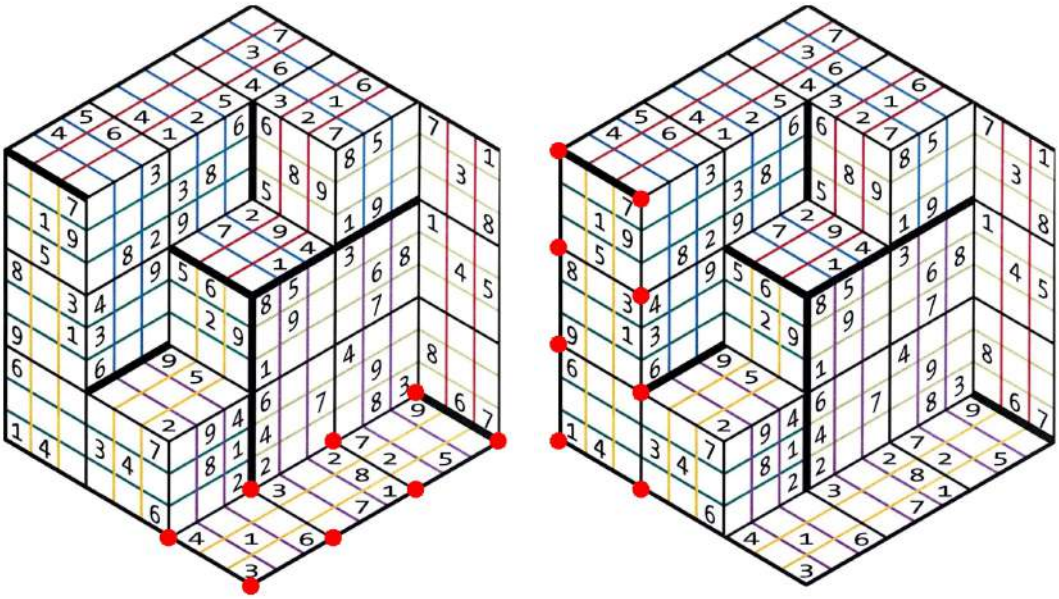
Figura 5.1: a) Punctele unei fețe; b) primul drum din care face parte fața; c) al doilea drum din care face parte fața

5.1 Algoritmul

Algoritmul implementat este unul de tip *backtracking*, construit pentru căutarea unei soluții unice și urmează structura standard.

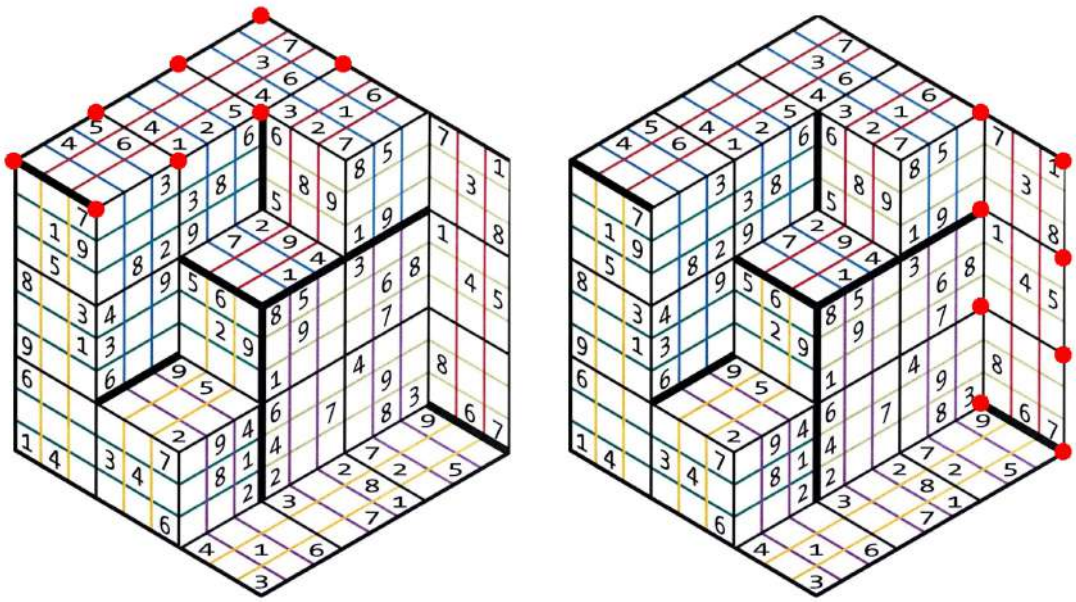
O convenție importantă în procesul realizării algoritmului este direcția drumurilor. Pentru cele care provin din punctele unor segmente de tip vertical, se observă că direcția este tot timpul pe orizontală. Pentru cele care provin din punctele unor segmente de tip slash, direcția este pe verticală. Segmentele de tip backslash reprezintă un caz special, în care direcția este stabilită dinamic, în funcție de axele pe care se află fețele. Atât pentru punctele de pe partea dreaptă, cât și pentru cele de jos, direcția este pe verticală dacă toate fețele se află pe axa X și pe orizontală dacă toate fețele se află pe axa Z (**Figurile**

5.3, *5.2*). Dacă fețele provin atât de pe axa X cât și de pe axa Z, atunci direcția va fi interpretată în timpul rulării algoritmului de rezolvare.



(a) (b)

Figura 5.2: Drumuri extrase din punctele de jos, cu direcții diferite; a) drum cu direcție orizontală; b) drum cu direcție verticală



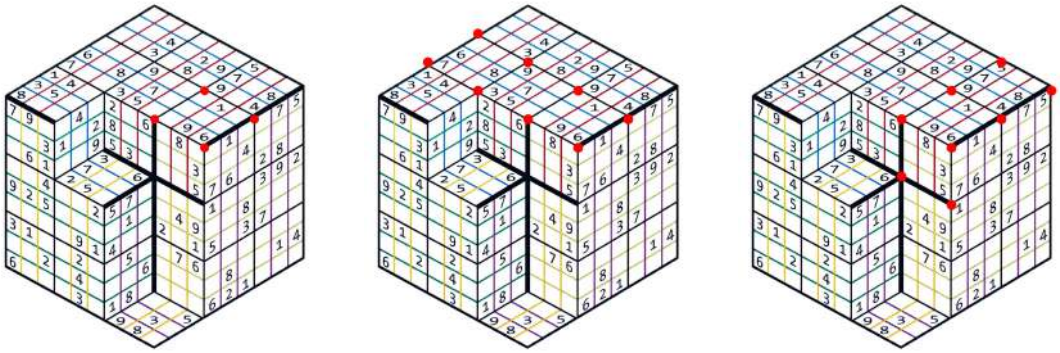
(a) (b)

Figura 5.3: Drumuri extrase din punctele de pe dreapta, cu direcții diferite; a) drum cu direcție orizontală; b) drum cu direcție verticală

5.2 Verificarea validității unei configurații

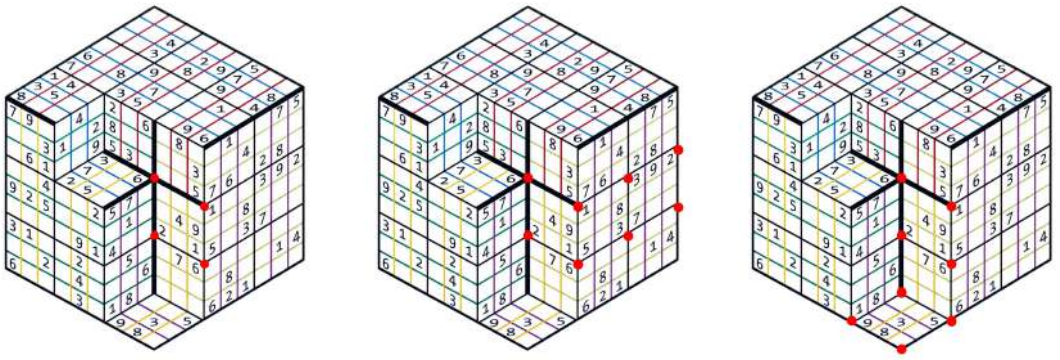
Cazurile triviale sunt cele în care încercăm asignarea unei cifre pe o poziție unde fața are deja o cifră, sau atunci când cifra dată se află deja pe altă poziție în față, caz în care configurația ar fi invalidă. Dacă nu suntem într-un caz trivial, este necesar să parcurgem ambele drumuri și să verificăm dacă cifra se află pe linia și pe coloana dată.

O observație importantă este faptul că fiecare față face parte din două drumuri ce provin din una dintre următoarele combinații de segmente:



(a) (b) (c)

Figura 5.4: a) Imaginea unei fețe; b) primul drum, care provine dintr-un segment de tip slash; c) al doilea drum, care provine dintr-un segment de tip backslash



(a) (b) (c)

Figura 5.5: a) Imaginea unei fețe; b) primul drum, care provine dintr-un segment de tip vertical; c) al doilea drum, care provine dintr-un segment de tip backslash

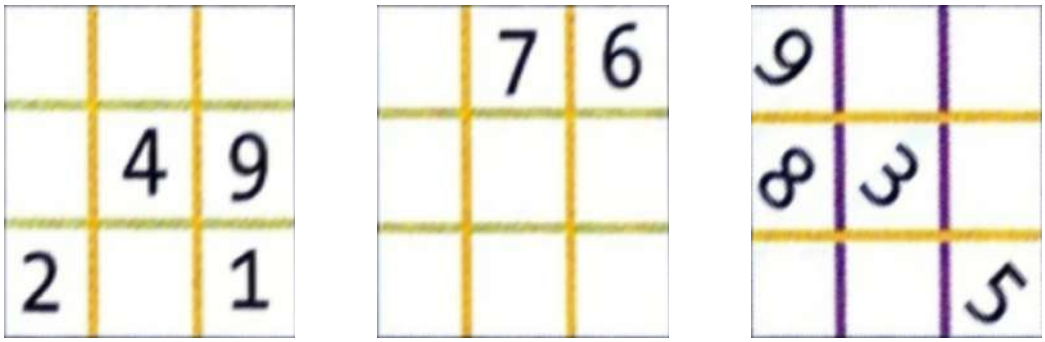
Astfel, observăm că fețele dintr-un drum pot proveni fie toate de pe aceeași axă, fie dintr-o combinație de axe: X cu Y, X cu Z sau Y cu Z. Pentru a putea verifica dacă asignarea unei cifre pe o față la o linie și la o coloană dată este validă, distingem două cazuri.

Cazul drumurilor în care toate fețele provin de pe aceeași axă sau dintr-o combinație între axele X și Y

În acest caz, se observă că este suficient să parcurgem drumurile, extrăgând cifrele de pe linia dată (dacă direcția drumului este pe orizontală) sau de pe coloană (dacă direcția este pe verticală) și să verificăm dacă cifra pe care vrem să o asignăm se află printre acestea - în acest caz, configurația este invalidă.

Cazul drumurilor în care fețele se află pe axele Z și X

În **Figura** *5.5c*) putem observa că orientarea feței de pe axa Z este diferită față de cea a imaginii extrase (**Figura** *5.6*).



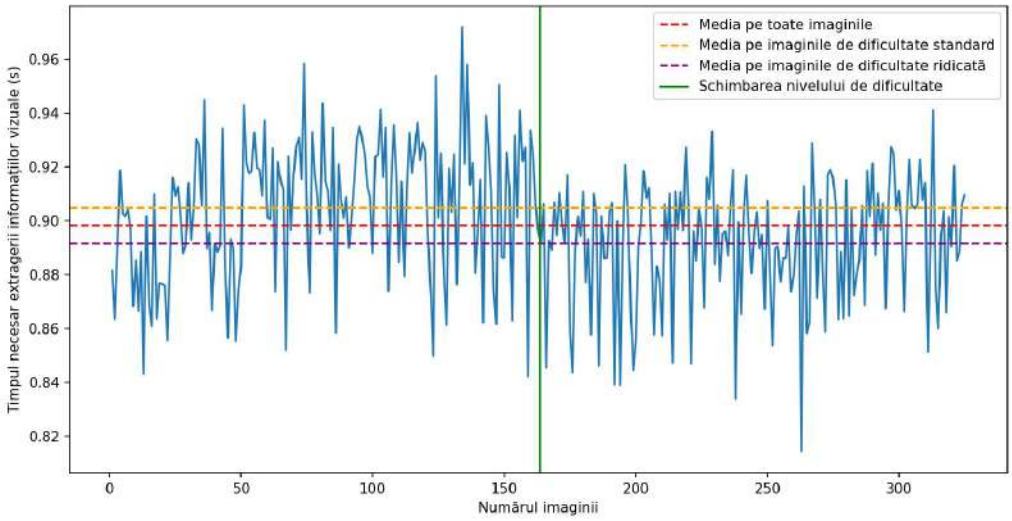
(a) (b) (c)

Figura 5.6: Imaginile fețelor drumului din **Figura** *5.5c*

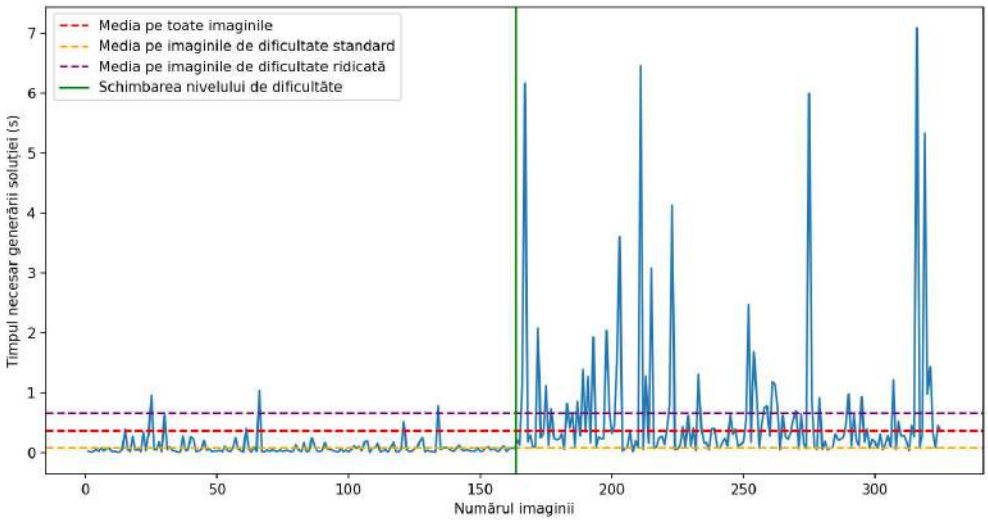
Pentru a rezolva această problemă, considerăm direcția drumului ca fiind pe orizontală dacă fața pentru care încercăm să asignăm o cifră se află pe axa Z și rotim restul fețelor la 90°. Dacă fața se află pe axa X, considerăm direcția ca fiind pe verticală și rotim restul fețelor la $-90°$.

5.3 Rezultate

Algoritmul descris reușește să genereze soluția pentru toate imaginile din baza de date, într-un timp mediu de $\approx 1.27s$, din care $\approx 0.9s$ pentru extragerea informațiilor vizuale și $\approx 0.37s$ pentru generarea soluției. Se observă, de asemenea, diferența între performanțele pe imaginile de dificultate standard și pe cele de dificultate mai ridicată (**Figura** *5.7b*). În medie, pe imaginile standard obținem ca timp de rulare $\approx 1s$, iar pentru cele de dificultate mai ridicată ≈ 1.55. Timpul maxim necesar rezolvării unei configurații de dificultate standard este de $1.95s$, iar pentru configurațiile de dificultate mai ridicată $7.98s$.

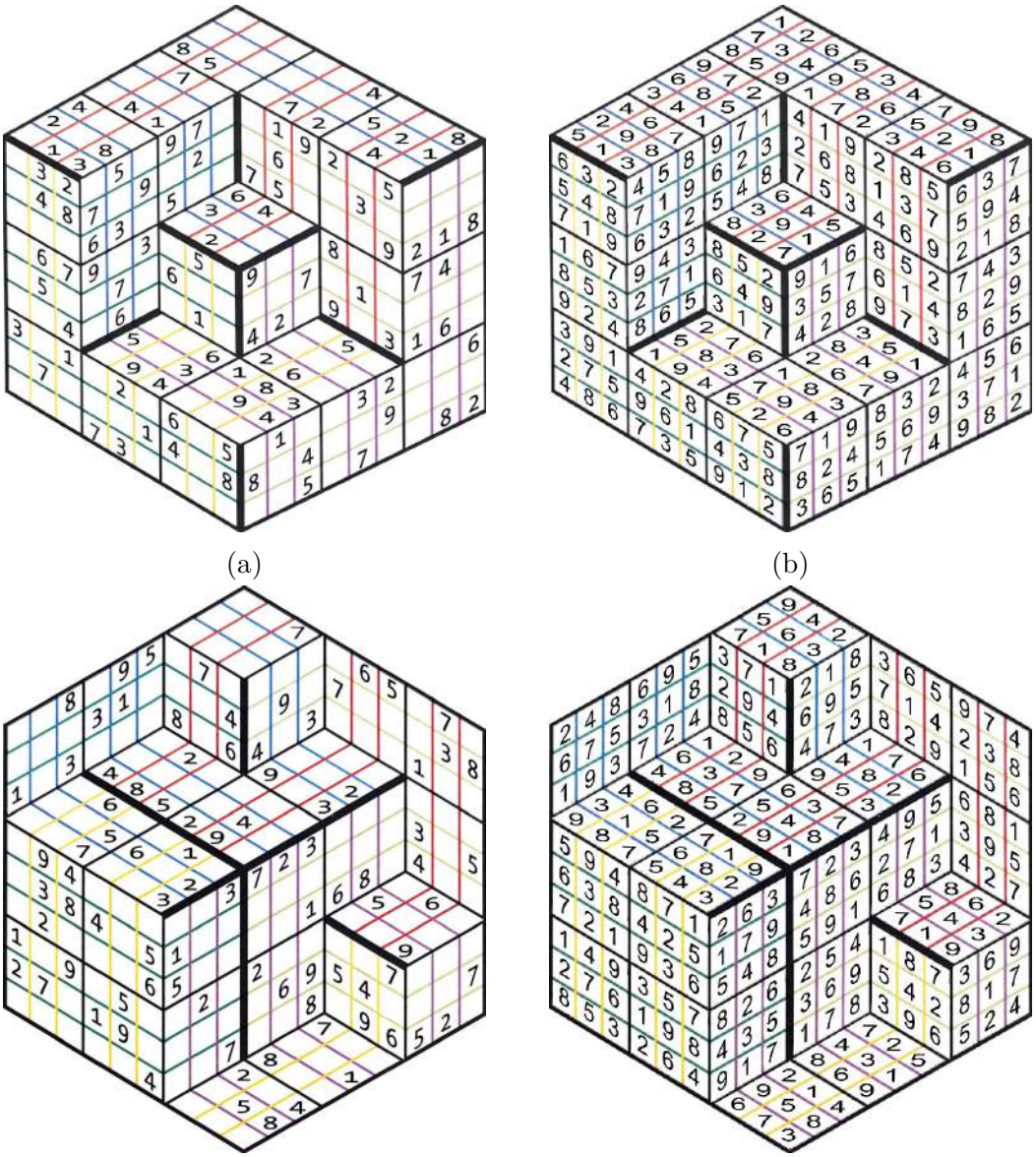


(a)



(b)

Figura 5.7: Vizualizările timpilor de execuție al algoritmilor; a) pentru extragerea informațiilor vizuale; b) pentru generarea soluției.



(a) (b) (c) (d)

Figura 5.8: Configurațiile pentru care soluția a fost generată cel mai rapid, respectiv cel mai lent: a) imaginea 13; b) soluția, generată în 0.85s; c) imaginea 316; d) soluția, generată în 7.98s.

Capitolul 6

Concluzie

În cadrul acestei lucrări am analizat procesul dezvoltării a doi algoritmi, unul care folosește tehnici din domeniul vederii artificiale pentru a extrage informații din imagini cu Sudoku 3D și unul pentru generarea soluției. Am plecat de la o bază de date cu imagini, pe care le-am preprocesat astfel încât să corectăm erorile de înclinare rezultate în urma scanării, am extras informații despre segmentele îngroșate și le-am folosit pentru a parcurge drumurile și a extrage informații despre configurații. Pentru rezolvarea acestora, am ales un algoritm de tip backtracking care implementează constrângerile impuse de regulile jocului. Fiecare etapă a venit cu provocări specifice.

O primă provocare a constat în găsirea unei modalități de a extrage imaginea unei fețe 3×3 fiind date patru puncte. Deoarece acestea au fost calculate relativ la o imagine cu conturul unui cub încadrat perfect iar imaginile din baza noastră de date prezentau mici erori (chiar și după aplicarea procesului de corectare a înclinării), o simplă transformare de perspectivă nu a fost suficientă, deoarece fețele extrase aveau atât un grad de rotație, cât și o oarecare deplasare față de centru, ceea ce îngreuna procesul de extragere a cifrelor.

Cea mai mare provocare a fost găsirea unei metode algoritmice de a putea parcurge drumurile și a extrage fețele. Deși în descrierea oficială a jocului este sugerată urmărirea fețelor folosind culorile din careuri, această idee ar fi fost mai dificilă ca implementare, mai puțin robustă și nu ar fi funcționat pe imagini în tonuri de gri (variațiunea mai dificilă a aceluiași puzzle).

În viitor, soluția ar putea fi îmbunătățită prin adaptarea algoritmului de vedere artificială astfel încât să funcționeze nu doar pe imagini scanate, ci și pe imagini fotografiate. De asemenea, deși algoritmul prin care generăm soluția rulează în medie destul de rapid, am observat că există configurații pentru care acesta poate fi chiar și de ≈ 10 ori mai lent. Astfel, acesta ar putea fi optimizat, sau înlocuit complet cu unul mai eficient. De asemenea, am putea dezvolta o aplicație pentru telefon care să permită fotografierea unei imagini și afișarea soluției asociate.

Bibliografie

[1] Ganesh Gorain. „Mathematics of Magic Squares". **in***JK Times*: 4 (**january** 2011), **pages** 48–55.

[2] https://www.amazon.co.uk/3D-Sudoku-Puzzle-Book/dp/1445407078. *ultima accesare 01.06.2022*.

[3] https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. *ultima accesare 01.06.2022*.

[4] https://pyimagesearch.com/2021/01/19/image-masking-with-opencv/. *ultima accesare 01.06.2022*.

[5] https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html. *ultima accesare 01.06.2022*.

[6] https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html. *ultima accesare 01.06.2022*.

[7] https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html. *ultima accesare 01.06.2022*.

[8] https://theailearner.com/tag/cv2-warpperspective. *ultima accesare 01.06.2022*.

[9] https://pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/. *ultima accesare 01.06.2022*.