

Interpretação e modelação de esquemas elétricos com recurso a algoritmos de visão computacional

HUGO MIGUEL RIBEIRO BARBOSA
novembro de 2022

POLITÉCNICO DO PORTO
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Interpretação e modelação de esquemas elétricos com recurso a algoritmos de visão computacional

Hugo Miguel Ribeiro Barbosa

Mestrado em Engenharia Electrotécnica e de Computadores
Área de Especialização em Automação e Sistemas



DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
Instituto Superior de Engenharia do Porto

Novembro, 2022

*Esta dissertação satisfaz, parcialmente, os requisitos que constam da Ficha de
Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em
Engenharia Electrotécnica e de Computadores, Área de Especialização em
Automação e Sistemas.*

Candidato: Hugo Miguel Ribeiro Barbosa, N.º 1110539,
1110539@isep.ipp.pt

Orientação Científica: Mário Jorge de Andrade Ferreira Alves,
mjf@isep.ipp.pt

Coorientação Científica: Lino Manuel Cerqueira Sousa, sss@isep.ipp.pt



DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Novembro, 2022

Agradecimentos

Gostaria de expressar os meus sinceros agradecimentos a todas as pessoas que, de alguma forma, contribuíram para a concretização deste projeto.

Agradeço aos meus orientadores, Professor Mário Alves e Professor Lino Sousa, pelo acompanhamento, aconselhamento e conhecimentos que me transmitiram, que foram fundamentais para a realização deste trabalho.

Uma palavra de agradecimento também dirigida ao Professor André Rocha, pela ajuda e sugestões dadas durante o desenvolvimento do projeto.

Finalmente, agradeço à minha família, pela motivação e apoio sempre concedidos para superar os desafios que vão surgindo ao longo da minha vida.

Resumo

A análise de circuitos elétricos/eletrônicos é uma componente fundamental na área da Engenharia Eletrotécnica. Geralmente, os estudantes iniciam a análise de um circuito desenhando-o em papel, e, posteriormente à sua resolução (analítica), implementam o circuito num simulador para verificar e validar os resultados obtidos analiticamente. No entanto, este processo faz com que os estudantes necessitem de voltar a desenhar o circuito, tarefa que consome tempo e durante a qual poderão ser introduzidos erros ao traçar o mesmo no simulador.

Este projeto consiste no desenvolvimento de um módulo de *software* capaz de interpretar a topologia de um esquema elétrico e de construir o seu modelo a partir de uma imagem, quer seja desenhado “à mão” ou a computador. Este módulo é baseado em algoritmos de visão computacional, propondo uma abordagem inovadora para a análise geral de um circuito elétrico através de uma imagem, com a deteção e extração dos seguintes elementos presentes no mesmo: componentes, nós, interligações entre os elementos, e *labels* (nomes, valores, etc) associadas aos respetivos elementos do circuito. O *software* deste projeto gera um mapa com o modelo do esquema elétrico analisado, resultante do processo de segmentação efetuado, bem como imagens com a *region of interest* (ROI) de cada componente e *label* extraída. Os ficheiros produzidos poderão ser utilizados por um outro módulo de *software* (não abrangido pela presente dissertação), responsável pela classificação dos componentes segmentados e também pelo reconhecimento dos caracteres alfanuméricos contidos nas *labels* do circuito. Assim, a combinação de ambos os módulos de *software* conceberá uma aplicação completa de reconhecimento de circuitos elétricos a partir de uma imagem. A finalidade futura desta aplicação é a sua integração na ferramenta U=RIsolve, *framework web* para ensino e auto-aprendizagem de metodologias para análise de circuitos, o que permitirá aos estudantes a obtenção e validação dos resultados do circuito através de uma imagem.

Palavras-Chave: Análise de circuitos, visão computacional, processamento de imagem, segmentação de circuito, aprendizagem.

Abstract

The analysis of electrical/electronic circuits is a fundamental component in the area of Electrical Engineering. Generally, students start the analysis of a circuit by drawing it on paper, and, after its (analytical) resolution, they implement the circuit in a simulator to verify and validate the results obtained analytically. However, this process makes students need to redraw the circuit, a time-consuming task during which errors may be introduced when tracing the circuit in the simulator.

This project consists in the development of a software module capable of interpreting the topology of an electrical schematic and of building its model from an image, whether drawn by hand or by computer. This module is based on computer vision algorithms, proposing an innovative approach for the general analysis of an electrical circuit through an image, with the detection and extraction of the following elements present in it: components, nodes, interconnections between elements, and labels (names, values, etc) associated with the respective circuit elements. The software of this project generates a map with the model of the electrical schematic analyzed, resulting from the segmentation process carried out, as well as images with the region of interest (ROI) of each component and label extracted. The files produced may be used by another software module (not covered by this dissertation), responsible for classifying the segmented components and also for recognizing the alphanumeric characters contained in the labels of the circuit. Thus, the combination of both software modules will create a complete application for recognizing electrical circuits from an image. The future purpose of this application is its integration into the U=RI solve tool, a web framework for teaching and self-learning methodologies for circuit analysis, which will allow students to obtain and validate the circuit results through an image.

Keywords: Circuit analysis, computer vision, image processing, circuit segmentation, learning.

Índice

Lista de Figuras	ix
Lista de Tabelas	xv
Listagens	xvii
Lista de Acrónimos	xix
1 Introdução	1
1.1 Contextualização	1
1.2 Objetivos	2
1.3 Contribuições	3
1.4 Plano de Trabalho	4
1.5 Organização da Dissertação	4
2 Estado da Arte	7
2.1 Visão computacional	7
2.1.1 Desafios	8
2.1.2 Evolução	9
2.1.3 Áreas de aplicação	12
2.2 Processamento de imagem	15
2.2.1 Imagem digital	16
2.2.2 Convolução	18
2.2.3 <i>Smoothing</i>	20
2.2.4 Operações de <i>threshold</i>	23
2.2.5 Transformações morfológicas	27
2.3 Reconhecimento de circuitos elétricos	31
2.3.1 Pré-processamento	31
2.3.2 Segmentação	33
2.3.3 Classificação	36
2.4 Ferramentas de <i>software</i> de visão computacional	37
2.4.1 OpenCV: <i>Open Source Computer Vision Library</i>	37
2.4.2 Simd	39
2.4.3 Boost GIL	39

2.4.4	Magick++	39
3	Arquitetura	41
3.1	Requisitos	41
3.2	Modelo JSON	42
3.3	Diagrama de blocos	45
4	Implementação	49
4.1	Organização do <i>software</i>	49
4.2	Funcionamento do <i>software</i>	50
4.2.1	Início do programa	52
4.2.2	Analisador da linha de comandos	53
4.2.3	<i>Logger</i>	54
4.2.4	<i>Wrapper</i> de OpenCV	56
4.2.5	Gestor de processamento de imagem	57
4.2.6	Receção de imagem	58
4.2.7	Pré-processamento	59
4.2.8	Segmentação	65
4.2.9	<i>Regions of interest</i>	79
4.2.10	Mapa de segmentação	82
4.3	Testes unitários	85
5	Casos de estudo	87
5.1	Circuito elétrico desenhado “à mão”	87
5.2	Circuito elétrico desenhado a computador	92
5.3	Circuito elétrico complexo	97
5.4	Limitações	101
6	Conclusões	105
6.1	Trabalho Futuro	107
	Referências	108
	Anexo A Exemplo de mapa de segmentação	117
	Anexo B Guia de utilização do <i>software</i>	125
B.1	Pré-requisitos	125
B.1.1	Instalação local	125
B.1.2	Utilização da plataforma Docker	127
B.2	Compilação e execução do <i>software</i>	128
	Anexo C Ficheiros de imagens de processamento	131

Lista de Figuras

1.1	Plano de desenvolvimento do trabalho.	6
2.1	Sistemas de visão humana e computacional [9].	8
2.2	Exemplo de imagem com diversos objetos. O círculo vermelho assinala uma região onde quase não há alteração de brilho, e o círculo verde indica uma zona cuja alteração de intensidade se deve a efeitos de luminosidade [13].	9
2.3	Evolução dos tópicos de investigação em visão computacional [10]. .	10
2.4	Deteção de objetos e pessoas em veículos autónomos [15].	12
2.5	Identificação de lugares de estacionamento [14].	13
2.6	Monitorização de tráfego rodoviário [14].	13
2.7	Exames pulmonares em combinação com o modelo COVID-Net [16].	14
2.8	Biópsia de gânglio linfático com auxílio da visão computacional [17].	15
2.9	Reconhecimento facial [18].	15
2.10	Cores disponíveis para cada píxel consoante a sua gama de <i>bits</i> . . .	16
2.11	Convenção do sistema de coordenadas usado numa imagem [19]. . .	16
2.12	Imagens binárias usadas em segmentação [19].	17
2.13	Imagem em escala de cinza e respetiva matriz [23].	17
2.14	Imagem <i>Red, Green and Blue</i> (RGB) e sua interpretação pelo computador [19].	18
2.15	Centro do <i>kernel</i> : (a) <i>kernel</i> 3 x 3 com centro válido e (b) <i>kernel</i> 2 x 2 sem centro válido [21].	19
2.16	Operação de convolução [24].	20
2.17	Convolução usando <i>zero-padding</i> [25].	20
2.18	Aplicação do <i>average filter</i> : imagens originais à esquerda; imagens de saída à direita [26].	21
2.19	Aplicação do filtro gaussiano: imagens originais à esquerda; imagens de saída à direita [26].	22
2.20	Aplicação do <i>median filter</i> : imagens originais à esquerda; imagens de saída à direita [26].	23
2.21	Aplicação do <i>bilateral filter</i> : imagens originais à esquerda; imagens de saída à direita [26].	23
2.22	Gráfico de intensidade dos píxeis da imagem para <i>threshold</i> [30]. . .	24

2.23	Resultados da aplicação do: (a) <i>threshold</i> binário e (b) <i>threshold</i> binário invertido [30].	25
2.24	Resultado da aplicação do <i>threshold</i> truncado [30].	25
2.25	Resultados da aplicação do: (a) <i>threshold</i> para zero e (b) <i>threshold</i> para zero invertido [30].	26
2.26	Imagem original para <i>thresholding</i> adaptativo [31].	27
2.27	Resultados da aplicação do: (a) <i>threshold</i> simples, (b) <i>threshold</i> baseado no algoritmo de Otsu, (c) <i>threshold</i> adaptativo com média aritmética e (d) <i>threshold</i> adaptativo com média gaussiana [30].	27
2.28	Transformações morfológicas básicas: (a) imagem de entrada, (b) aplicação da dilatação e (c) aplicação da erosão [33].	29
2.29	Operação morfológica de <i>opening</i> : imagem original à esquerda; resultado da aplicação à direita [34].	29
2.30	Operação morfológica de <i>closing</i> : imagem original à esquerda; resultado da aplicação à direita [34].	29
2.31	Operação morfológica de gradiente morfológico: imagem original à esquerda; resultado da aplicação à direita [34].	30
2.32	Operação morfológica de <i>top hat</i> : imagem original à esquerda; resultado da aplicação à direita [32].	30
2.33	Operação morfológica de <i>black hat</i> : imagem original à esquerda; resultado da aplicação à direita [32].	31
2.34	Diagrama geral para reconhecimento de circuitos elétricos a partir de uma imagem.	31
2.35	Exemplo de pré-processamento de imagem: (a) imagem de entrada, (b) imagem em escala de cinza, (c) filtro gaussiano, (d) imagem binária (<i>thresholding</i>), (e) dilatação e (f) esqueletização [37].	32
2.36	Metodologia de segmentação categorizando componentes [35].	33
2.37	Segmentação de componentes com formas fechadas: (a) imagem original, (b) imagem binária, (c) imagem após preenchimento de regiões, (d) detecção de componentes depois de <i>opening</i> , (e) localização de componentes na imagem original e (f) componentes segmentados [35].	34
2.38	Segmentação de componentes com linhas conectadas: (a) imagem após <i>closing</i> , (b) detecção de componentes depois de <i>opening</i> e (c) localização de componentes na imagem original [35].	34
2.39	Segmentação de componentes com linhas desconectadas: (a) imagem após espessamento e <i>thinning</i> , (b) remoção das restantes ligações do circuito e (c) localização de componentes na imagem original [35].	35
2.40	Segmentação das conexões do circuito: (a) identificação de fios de conexão e (b) detecção de terminais dos componentes [37].	36

2.41	Segmentação dos nós do circuito: (a) detecção de conexões através de <i>Hough transform</i> e (b) identificação de nós [41].	36
3.1	Diagrama de blocos da arquitetura geral do <i>software</i>	46
4.1	Fluxograma geral de funcionamento do programa.	52
4.2	Diagrama de sequência do método <i>Application::exec</i>	54
4.3	Diagrama de classes relativo ao analisador da linha de comandos. . .	55
4.4	Exemplo de <i>log</i> de mensagens do programa.	56
4.5	Encapsulamento da biblioteca OpenCV.	56
4.6	Fluxograma do processamento de imagem.	58
4.7	Diagrama de sequência do processo de recepção e leitura da imagem. .	59
4.8	Diagrama de sequência do processo de pré-processamento da imagem. .	60
4.9	Circuito-exemplo.	61
4.10	Conversão da imagem para escala de cinza durante o pré-processamento. .	61
4.11	Filtragem da imagem durante o pré-processamento.	61
4.12	Operação de <i>thresholding</i> adaptativo durante o pré-processamento. .	63
4.13	Transformação morfológica de dilatação durante o pré-processamento. .	64
4.14	Operação de esqueletização da imagem durante o pré-processamento. .	64
4.15	Fluxograma do processo de segmentação da imagem.	66
4.16	Diagrama de sequência do procedimento para detecção de conexões. .	68
4.17	Operações durante a detecção de conexões: (a) <i>closing</i> , (b) <i>opening</i> e (c) interseção.	69
4.18	Operações durante a detecção de conexões: (a) remoção de componentes e <i>labels</i> e (b) resultado da detecção de conexões (assinaladas a vermelho).	69
4.19	Diagrama de sequência do procedimento para detecção de componentes. .	70
4.20	Operações durante a detecção de componentes: (a) remoção de conexões, (b) <i>closing</i> e (c) resultado da detecção de componentes (assinalados a verde).	71
4.21	Diagrama de sequência do procedimento para atualização de conexões. .	71
4.22	Operações durante a atualização de conexões: (a) remoção de componentes e (b) resultado da atualização de conexões (assinaladas a vermelho).	72
4.23	Diagrama de sequência do procedimento para detecção de nós e atualização de conexões.	72
4.24	Resultado da detecção de nós e atualização de conexões (nós assinalados com pontos azuis e conexões com contornos vermelhos).	73
4.25	Diagrama de sequência do procedimento para detecção de <i>ports</i> de componentes.	74
4.26	Resultado da detecção de <i>ports</i> de componentes (assinalados a vermelho). .	74

4.27	Diagrama de sequência do procedimento para detecção de <i>labels</i>	76
4.28	Operações durante a detecção de <i>labels</i> : (a) remoção dos elementos, (b) <i>closing</i> , (c) <i>opening</i> e (d) resultado da detecção de <i>labels</i> (assinaladas com cor verde).	77
4.29	Diagrama de sequência do procedimento para associação de <i>labels</i> . .	78
4.30	Geração de <i>bounding boxes</i> para conexões e nós durante a associação de <i>labels</i> (assinaladas a verde).	79
4.31	Possíveis posições do elemento relativamente a uma <i>label</i> e respectivas distâncias consideradas.	79
4.32	Fluxograma do processo de geração de imagens <i>Region of Interest</i> (ROI).	80
4.33	Diagrama de sequência do procedimento para geração de ROI de componentes.	81
4.34	Imagens ROI de componentes: (a) fonte de tensão, (b) resistência, (c) condensador e (d) massa.	81
4.35	Diagrama de sequência do procedimento para geração de ROI de <i>labels</i> . 82	
4.36	Imagens ROI de <i>labels</i> associadas: (a) à fonte de tensão, (b) à resistência, e (c) ao condensador.	82
4.37	Relação entre classes para geração de <i>Universally Unique Identifier</i> (UUID).	83
4.38	Diagrama de sequência do procedimento para geração do mapa de segmentação.	84
4.39	Resultado da execução dos testes unitários.	86
4.40	Relatório gerado pelo LCOV.	86
5.1	Circuito elétrico do caso de estudo 1.	87
5.2	Pré-processamento do caso de estudo 1: (a) escala de cinza, (b) filtragem, (c) <i>thresholding</i> , (d) dilatação e (b) esqueletização.	88
5.3	Deteção de conexões do caso de estudo 1: (a) <i>closing</i> , (b) <i>opening</i> e (c) interseção.	89
5.4	Deteção de conexões do caso de estudo 1: (a) remoção de componentes e de <i>labels</i> , e (b) conexões detetadas.	89
5.5	Deteção de componentes do caso de estudo 1: (a) remoção de conexões, (b) <i>closing</i> e (c) componentes detetados.	89
5.6	Atualização de conexões detetadas do caso de estudo 1: (a) remoção de componentes e (b) conexões atualizadas.	90
5.7	Caso de estudo 1: (a) nós detetados e conexões atualizadas, e (b) <i>ports</i> de componentes detetados.	90
5.8	Deteção de <i>labels</i> do caso de estudo 1: (a) remoção dos elementos e (b) <i>closing</i>	90

5.9	Deteção de <i>labels</i> do caso de estudo 1: (a) <i>opening</i> e (b) <i>labels</i> detetadas.	90
5.10	Geração de ROI do caso de estudo 1: (a) de componentes e (b) de <i>labels</i> .	91
5.11	Circuito elétrico do caso de estudo 2.	93
5.12	Pré-processamento do caso de estudo 2: (a) escala de cinza e (b) filtragem.	93
5.13	Pré-processamento do caso de estudo 2: (a) <i>thresholding</i> , (b) dilatação e (c) esqueletização.	93
5.14	Deteção de conexões do caso de estudo 2: (a) <i>closing</i> , (b) <i>opening</i> e (c) interseção.	94
5.15	Deteção de conexões do caso de estudo 2: (a) remoção de componentes e de <i>labels</i> , e (b) conexões detetadas.	94
5.16	Deteção de componentes do caso de estudo 2: (a) remoção de conexões, (b) <i>closing</i> e (c) componentes detetados.	95
5.17	Atualização de conexões detetadas do caso de estudo 2: (a) remoção de componentes e (b) conexões atualizadas.	95
5.18	Caso de estudo 2: (a) nós detetados e conexões atualizadas, e (b) <i>ports</i> de componentes detetados.	95
5.19	Deteção de <i>labels</i> do caso de estudo 2: (a) remoção dos elementos e (b) <i>closing</i> .	96
5.20	Deteção de <i>labels</i> do caso de estudo 2: (a) <i>opening</i> e (b) <i>labels</i> detetadas.	96
5.21	Geração de ROI do caso de estudo 2: (a) de componentes e (b) de <i>labels</i> .	96
5.22	Circuito elétrico do caso de estudo 3.	98
5.23	Pré-processamento do caso de estudo 3.	99
5.24	Deteção de componentes do caso de estudo 3.	99
5.25	Deteção de nós e de conexões do caso de estudo 3.	100
5.26	Deteção de <i>ports</i> de componentes do caso de estudo 3.	100
5.27	Deteção de <i>labels</i> do caso de estudo 3.	101
5.28	Comparação da transformação de <i>closing</i> para a imagem do circuito com dimensões de (a) 600 x 382 e de (b) 1200 x 748 píxeis.	102
5.29	Comparação da deteção de componentes para a imagem do circuito com dimensões de (a) 600 x 382 e de (b) 1200 x 748 píxeis.	102
5.30	Junção de dois componentes em apenas um.	103
5.31	Deteção de <i>labels</i> como componentes do circuito.	103
5.32	<i>Label</i> de um nó associada a uma conexão do circuito.	103
D.1	Mensagens geradas quando a receção da imagem ocorre com (a) sucesso ou (b) insucesso.	135
D.2	Mensagens geradas durante o pré-processamento.	136

D.3	Mensagens geradas durante a detecção de conexões.	136
D.4	Mensagens geradas durante a detecção de componentes.	136
D.5	Mensagens geradas durante a atualização de conexões.	136
D.6	Mensagens geradas durante a detecção de nós e atualização de conexões.	136
D.7	Mensagens geradas durante a detecção de <i>ports</i> de componentes. . . .	137
D.8	Mensagens geradas durante a atualização de componentes detetados.	137
D.9	Mensagens geradas durante a detecção de <i>labels</i>	137
D.10	Mensagens geradas durante a associação de <i>labels</i>	137
D.11	Mensagens geradas durante a criação de imagens ROI.	137
D.12	Mensagens geradas durante a criação do mapa de segmentação. . . .	137

Lista de Tabelas

C.1	Imagens geradas durante a receção de imagem.	131
C.2	Imagens geradas durante o pré-processamento.	132
C.3	Imagens geradas durante a deteção de conexões.	132
C.4	Imagens geradas durante a deteção de componentes.	132
C.5	Imagens geradas durante a atualização de conexões.	132
C.6	Imagens geradas durante a deteção de nós e atualização de conexões.	132
C.7	Imagens geradas durante a deteção de <i>ports</i> de componentes.	132
C.8	Imagens geradas durante a deteção de <i>labels</i>	133
C.9	Imagens geradas durante a associação de <i>labels</i>	133

Listagens

3.1	Tipo básico <i>Position</i>	43
3.2	Tipo secundário <i>Port</i>	43
3.3	Tipo secundário <i>Label</i>	44
3.4	Tipo principal <i>Component</i>	44
3.5	Tipo principal <i>Connection</i>	45
3.6	Tipo principal <i>Node</i>	45
4.1	Função <i>main</i> do programa.	52
4.2	Exemplo de encapsulamentos de OpenCV.	57
4.3	Construtor da <i>struct Component</i>	83
4.4	Estrutura base do mapa de segmentação.	85
5.1	Excerto do mapa de segmentação do caso de estudo 1.	92
5.2	Excerto do mapa de segmentação do caso de estudo 2.	97
A.1	Mapa de segmentação resultante para o circuito-exemplo.	117
B.1	Comandos para compilação da imagem Docker.	127
B.2	Comando para execução do <i>container</i> Docker.	128
B.3	Comandos para configuração CMake.	129
B.4	Comando para compilação do <i>software</i> usando <i>cmake</i>	129
B.5	Comando para compilação do <i>software</i> usando <i>make</i>	129
B.6	Comando para execução do <i>software</i>	130
B.7	Comandos para compilação e execução dos testes unitários.	130

Lista de Acrónimos

API	<i>Application Programming Interface</i>
BMP	<i>Bitmap</i>
GCC	<i>GNU Compiler Collection</i>
GIL	<i>Generic Image Library</i>
GUI	<i>Graphical User Interface</i>
HOG	<i>Histogram of Oriented Gradients</i>
HTTP	<i>Hypertext Transfer Protocol</i>
I/O	<i>Input/Output</i>
JPEG	<i>Joint Photographic Experts Group</i>
JSON	<i>JavaScript Object Notation</i>
MIT	<i>Massachusetts Institute of Technology</i>
OCR	<i>Optical Character Recognition</i>
PNG	<i>Portable Network Graphics</i>
RGB	<i>Red, Green and Blue</i>
RGBA	<i>Red, Green, Blue and Alpha</i>
ROI	<i>Region of Interest</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SVM	<i>Support Vector Machine</i>
UUID	<i>Universally Unique Identifier</i>

Capítulo 1

Introdução

Neste capítulo é realizada uma introdução ao projeto, nomeadamente o contexto em que este trabalho se insere e os objetivos a serem cumpridos. Adicionalmente, são evidenciadas as contribuições deste projeto e o plano de trabalho adotado como base ao seu desenvolvimento, bem como a organização desta dissertação.

1.1 Contextualização

O desenho de diagramas é uma forma popular e de grande utilização nas áreas de Engenharia para representação de sistemas, visto que é uma metodologia simples e prática de demonstração de conceitos técnicos e científicos sem entrar em muitos detalhes. No campo da Engenharia Eletrotécnica, o diagrama de um circuito elétrico/eletrónico é uma representação gráfica de componentes elétricos e das interligações entre si. Os componentes são desenhados utilizando um conjunto de figuras denominadas de símbolos, havendo um único símbolo para cada componente.

Existem diversas aplicações computacionais para desenho de circuitos elétricos/eletrónicos, no entanto, na Engenharia é habitual começar-se por desenhar os circuitos “à mão”, em papel. Esta prática também é muito utilizada em ambiente académico, pelos professores para exposição de conteúdo pedagógico, mas sobretudo pelos estudantes na sua fase de aprendizagem. Frequentemente, na análise de circuitos elétricos, os estudantes iniciam o processo desenhando o circuito em papel. Após a sua resolução (analítica), o circuito é geralmente implementado com recurso a um simulador, de forma a comprovar os resultados obtidos analiticamente. No sentido

de minorar o tempo consumido pelos estudantes neste processo de redesenho, durante o qual podem ser introduzidos erros, revelar-se-ia vantajoso a existência de uma ferramenta capaz de interpretar o circuito previamente desenhado em papel. Com esse intuito, o diagrama do circuito teria de ser convertido para uma forma digital, para possibilitar a sua análise pela ferramenta.

Neste contexto, e numa época em que as aplicações didáticas assumem um papel cada vez mais essencial no ensino de Engenharia, este projeto pretende desenvolver um módulo de *software* capaz de interpretar e modelar esquemas elétricos a partir de uma imagem. Este módulo poderá ser futuramente integrado no contexto da ferramenta U=RI solve [1–7], *framework web* para ensino e auto-aprendizagem de metodologias para análise de circuitos. U=RI solve é uma aplicação que, além de fornecer os resultados, também providencia as equações e os cálculos matemáticos realizados para chegar a essas mesmas soluções, disponibilizando assim as etapas intermédias efetuadas que são de grande relevância para uma maior consolidação de conhecimentos por parte dos alunos. Do mesmo modo, esta aplicação também apresenta as características do circuito, tais como os ramos e os nós que este possui. O módulo deste projeto será futuramente uma mais-valia para a aplicação U=RI solve, visto que, em conjunto com um outro módulo de *software* dedicado à classificação de componentes (esta dissertação não abrange o desenvolvimento do módulo de classificação), possibilitará a obtenção dos resultados de um circuito a partir de uma imagem.

A segmentação de um circuito através de uma imagem envolve a aplicação de algoritmos de processamento de imagem e de visão computacional, de modo a detetar os componentes e as respetivas interligações. Diversos desafios podem ser enfrentados para esta finalidade, relacionados com as variadas formas com que o circuito pode ser desenhado, a localização dos componentes, a extração das ligações entre os elementos do circuito e a deteção de nós, bem como a qualidade do papel, cor e até a pressão efetuada durante a escrita.

1.2 Objetivos

Este projeto tem como principal propósito contribuir para o ensino e aprendizagem de metodologias de análise de circuitos elétricos/eletrónicos, através do desenvolvimento de um módulo de *software* baseado em algoritmos de visão computacional, capaz de interpretar a topologia de um esquema elétrico e de construir o seu modelo a partir de uma imagem, quer seja desenhado “à mão” ou a computador. Nesse sentido, os objetivos a serem alcançados neste trabalho são os seguintes:

- Estudo e aplicação de diferentes algoritmos de pré-processamento de imagem que permitam adaptar a filtragem das várias fontes de ruído;

- Estudo e implementação de métodos para detecção e extração de padrões do circuito, para segmentação dos seguintes elementos:
 - Componentes presentes no circuito;
 - Nós do circuito;
 - Interligações entre os elementos do circuito;
 - *Labels* (nomes, valores, etc) associadas aos elementos do circuito (por exemplo, nome R1 associado a uma resistência).
- Geração de um mapa/matriz que agrupe os elementos do circuito segmentados e que contenha as suas interligações;
- Geração de imagens com os componentes e as *labels* segmentadas (o módulo de *software* deste trabalho não deverá ter capacidade de classificação/identificação dos componentes nem de reconhecimento de caracteres alfanuméricos, produzindo estes ficheiros de imagem que deverão ser utilizados por um outro módulo responsável por estas tarefas).

A finalidade futura do módulo a desenvolver neste projeto é a sua integração na aplicação U=RIolve.

1.3 Contribuições

O módulo de *software* desenvolvido neste trabalho pode contribuir significativamente para o ensino em Engenharia Eletrotécnica, constituindo uma ferramenta auxiliar de apoio ao estudo de metodologias de análise de circuitos elétricos. As contribuições deste projeto são as seguintes:

1. É proposto um novo método para a segmentação de um circuito elétrico, desenhado “à mão” ou a computador, a partir de uma imagem. Esta inovadora abordagem analisa o circuito na generalidade, detetando e extraindo os elementos presentes no mesmo: componentes, nós, conexões e *labels* associadas aos respetivos elementos do circuito;
2. O módulo implementado gera um mapa resultante do processo de segmentação, que representa o modelo do esquema elétrico analisado. Adicionalmente, são criadas imagens para cada componente e *label* detetada. Os ficheiros produzidos por este módulo são suficientes para a identificação e classificação dos elementos contidos no circuito elétrico, podendo ser utilizados por um outro módulo de *software* dedicado a este fim;
3. A integração futura deste módulo na aplicação U=RIolve adicionará a funcionalidade de análise e modelação de um circuito através de uma imagem.

Conjuntamente com um módulo de *software* de classificação, permitirá aos alunos a obtenção e validação dos resultados de um circuito com maior rapidez;

4. O código-fonte do módulo de *software* desenvolvido encontra-se disponível publicamente na plataforma GitHub [8], partilhando-se assim a metodologia aplicada neste projeto para segmentação de circuitos elétricos;
5. A escrita de um artigo científico relacionado com a presente dissertação está a ser delineada.

1.4 Plano de Trabalho

Este projeto seguiu a sequência de tarefas do plano de trabalho representado no diagrama de Gantt da Figura 1.1. O projeto teve o seu início com uma definição clara de objetivos e uma análise de requisitos, de forma a verificar a melhor abordagem para o seu desenvolvimento, bem como os resultados esperados. Posteriormente, procedeu-se ao estudo do Estado da Arte, tarefa que foi sendo realizada em paralelo com o desenvolvimento do trabalho, que incidiu maioritariamente na aquisição de conhecimentos base e de algoritmos da área da visão computacional. De seguida, estabeleceu-se a arquitetura a ser seguida pelo *software* e iniciou-se o seu desenvolvimento. Finalmente, realizaram-se testes ao módulo implementado e elaborou-se a presente dissertação.

1.5 Organização da Dissertação

A presente dissertação encontra-se organizada em seis capítulos. O Capítulo 1 contextualiza o trabalho e apresenta os objetivos a serem concretizados, exibindo ainda as contribuições e o plano e calendarização das tarefas deste projeto. No Capítulo 2, é exposto o estudo efetuado do Estado da Arte, descrevendo noções de visão computacional, algoritmos de processamento de imagem, técnicas de reconhecimento de circuitos elétricos, e ferramentas de *software* disponíveis de visão computacional. O Capítulo 3 retrata a arquitetura do sistema, evidenciando os requisitos e os diferentes componentes necessários para a sua implementação. No Capítulo 4, é abordada a implementação do trabalho proposto, contendo uma descrição técnica exaustiva acerca da organização e do funcionamento do módulo de *software* desenvolvido. O Capítulo 5 contém casos de estudo delineados com o intuito de avaliar o desempenho deste *software*, com utilização de imagens de circuitos de diferentes níveis de complexidade, frisando também as limitações do módulo implementado. Por último, no Capítulo 6, são mencionadas as conclusões do trabalho desenvolvido, incluindo

os objetivos alcançados, as dificuldades sentidas ao longo da implementação, assim como possíveis futuros desenvolvimentos para melhoria deste trabalho.

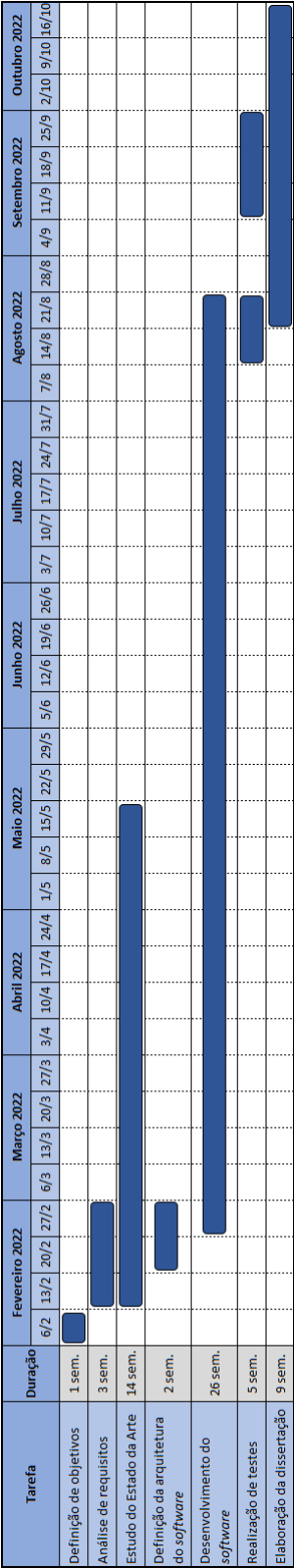


Figura 1.1: Plano de desenvolvimento do trabalho.

Capítulo 2

Estado da Arte

Este capítulo apresenta noções básicas, métodos e algoritmos da área da visão computacional, fundamentais para o desenvolvimento do módulo de *software* deste projeto. Conjuntamente, expõe diferentes técnicas de processamento de imagem aplicadas para reconhecimento de circuitos elétricos, descrevendo também algumas ferramentas de *software* existentes de visão computacional.

2.1 Visão computacional

A visão computacional consiste num vasto campo científico multidisciplinar. O seu foco reside em capacitar computadores e sistemas digitais para processamento, análise e compreensão da informação proveniente de imagens, vídeos ou outras fontes visuais, interpretando esses dados para concretizar um determinado propósito [9].

O ser humano possui um sistema visual capaz de interpretar o mundo a três dimensões com bastante facilidade. Características como a forma e cor de um objeto, padrões de luz e sombra nas superfícies, noção de distâncias entre objetos, reconhecimento de pessoas e emoções numa imagem através das suas expressões faciais, são perceções da visão humana com aparente pouco esforço e com respostas quase imediatas e normalmente disponíveis num décimo de segundo [10,11]. A tecnologia da visão computacional tenta “imitar” este incrível funcionamento do sistema visual humano, tal como demonstra a Figura 2.1, na qual cada sistema possui um “sensor” (olho para o humano, câmara para o computacional) e um “interpretador” (cérebro para o humano, computador para o computacional). Desta forma, a imagem da

fruteira, neste caso, é capturada pelo “sensor” e posteriormente analisada e compreendida pelo “interpretador”, resultando na detecção e identificação de cada fruto presente na imagem.

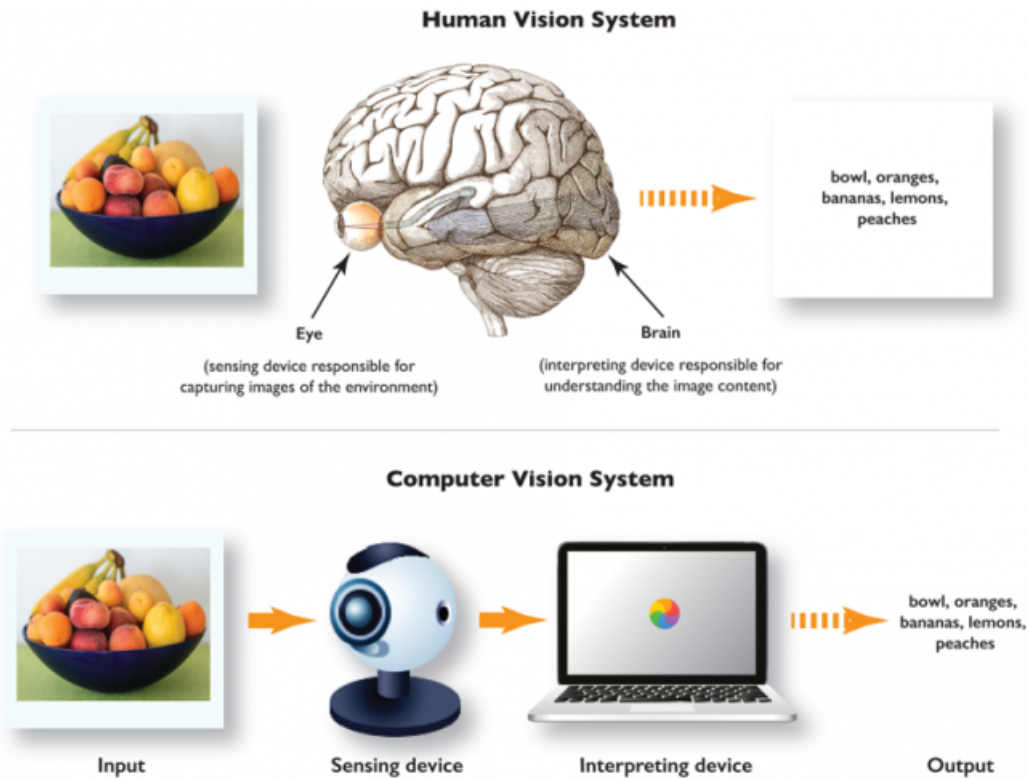


Figura 2.1: Sistemas de visão humana e computacional [9].

Ao contrário do sistema visual humano que faz uso da retina, nervos óticos e córtex visual, o de visão computacional utiliza câmaras, dados e diversos algoritmos para identificação de objetos, baseando-se fundamentalmente em reconhecimento de padrões. Assim, um sistema computacional treinado para, por exemplo, inspecionar determinados produtos, poderá analisar milhares destes por minuto, percebendo os que apresentam defeitos ou problemas imperceptíveis ao humano, conseguindo efetuar este processo de forma mais rápida que uma pessoa [12].

2.1.1 Desafios

Tal como já mencionado, o principal objetivo da visão computacional é a extração de informação útil a partir de imagens. No entanto, esta tarefa revelou-se ser profundamente desafiante, pois ocupou milhares de investigadores nas últimas décadas, e ainda hoje está longe a conceção de uma máquina capaz de “ver” de uso generalizado [13].

A complexidade dos dados visuais é parte do problema, exemplificada pela Figura 2.2, que ilustra uma imagem com múltiplos objetos. Além da variedade de

objetos existentes, quase todos se encontram parcialmente obstruídos, o que dificulta o estabelecimento, para um algoritmo de visão computacional, de quando um objeto termina e outro se inicia. Outra dificuldade está relacionada com a pouca alteração de brilho na fronteira entre o céu e o edifício, identificada na região destacada com um círculo vermelho. Por outro lado, o círculo verde assinala uma zona em que se verifica facilmente uma alteração na intensidade na janela do veículo devido a efeitos de luminosidade, contudo esta variação não corresponde a um limite de objeto ou alteração de material.



Figura 2.2: Exemplo de imagem com diversos objetos. O círculo vermelho assinala uma região onde quase não há alteração de brilho, e o círculo verde indica uma zona cuja alteração de intensidade se deve a efeitos de luminosidade [13].

O sistema visual humano comprova que estas complexidades podem ser superadas, o que demonstrou aos primeiros investigadores que a visão computacional poderia ser possível, embora extremamente desafiante.

2.1.2 Evolução

Durante várias décadas, cientistas e engenheiros ambicionavam a criação de máquinas com características da inteligência humana, que fossem capazes de “pensar” e atuar como os humanos. Uma das ideias mais fascinantes era providenciar ao computador a capacidade de “ver” e de interpretar o mundo à sua volta. Diversos estudos e desenvolvimentos nesta área foram realizados nos últimos 50 anos, alguns destacados na Figura 2.3.

O conceito de visão computacional surgiu pela primeira vez em finais dos anos de 1960, em universidades pioneiras da inteligência artificial. A sua finalidade era a imitação do sistema visual humano, com o objetivo de providenciar comportamento

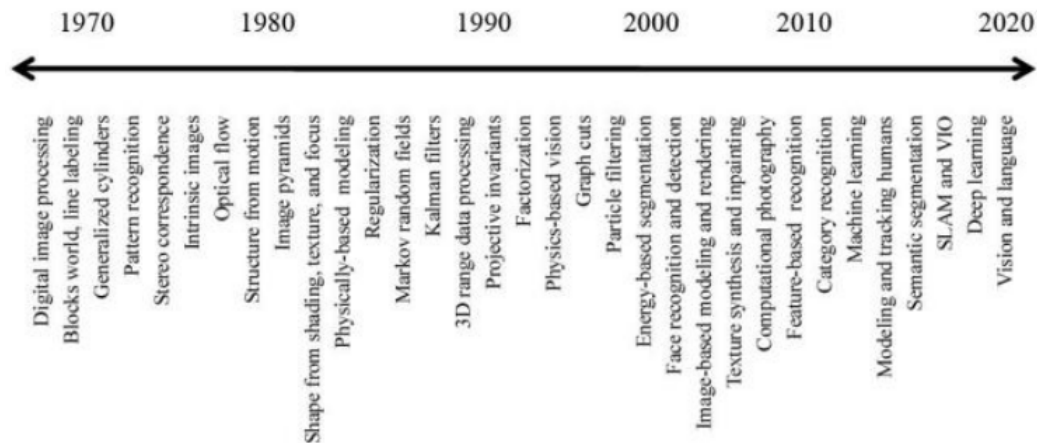


Figura 2.3: Evolução dos tópicos de investigação em visão computacional [10].

inteligente a robôs. Em 1966, acreditava-se que isso poderia ser alcançado através de um projeto de Verão, proposto por Marvin Minsky ao estudante Gerald Jay Sussman no *Massachusetts Institute of Technology* (MIT), cujo propósito era ligar uma câmara a um computador e conseguir que este descrevesse o que “viu” [10].

Na década de 1970, o campo de processamento digital de imagem já existia, e o que distinguia este da visão computacional era a obtenção da estrutura a três dimensões através de imagens, com a finalidade de compreensão do que esta representava. Os estudos efetuados durante esta década estabeleceram as primeiras bases para vários algoritmos de visão computacional que atualmente existem. Estes incluem a extração de curvas e deteção de linhas a partir de imagens, a modelação a três dimensões de objetos não-poliédricos e poliédricos, o reconhecimento de padrões, o entendimento de variações de intensidades e sombreados para compreensão dos seus efeitos em termos de formação de imagem (tópico designado por “imagens intrínsecas”), bem como a perceção de movimento.

Nos anos de 1980, os estudos incidiram sobretudo em técnicas matemáticas mais sofisticadas para análise quantitativa de imagens. *Image pyramids*, dedução de formas a partir de sombreamento, textura e foco, conceito de *scale-space* (manipulação de estruturas da imagem em escalas diferentes), e ainda a busca de um melhor método para deteção de bordas (fronteiras ou extremidades de objetos) e de contornos foram algumas das investigações levadas a cabo neste período. Os investigadores constataram que muitos dos algoritmos matemáticos podiam ser unificados, ou pelo menos descritos, usando a mesma *framework* matemática como a regularização. Simultaneamente, verificaram que estes também podiam igualmente ser formulados utilizando os modelos discretos de *Markov random fields*, que possibilitavam o uso de melhores algoritmos para pesquisa e otimização. De referir também que, em 1982, o neurocientista David Marr introduziu a noção de três níveis para formulação de um sistema de processamento de informação visual, filosofia que ainda hoje é

aplicável:

- **Teoria computacional:** qual o objetivo da tarefa de computação e quais as limitações que são conhecidas ou que podem ser trazidas para suportar o problema;
- **Representações e algoritmos:** como são representadas a entrada, a saída e a informação intermediária, e quais os algoritmos usados para calcular o resultado pretendido;
- **Implementação de *hardware*:** como são as representações e algoritmos mapeados para o *hardware* real, e como as limitações de *hardware* podem influenciar na escolha das representações e algoritmos.

Os anos de 1990 viram os tópicos anteriormente mencionados continuarem a ser explorados, sendo que alguns deles tornaram-se significativamente mais ativos. Realçam-se as atividades em reconstruções projetivas 3D e em segmentação de imagem, processo que reside na detecção e extração de padrões da imagem, permitindo a sua representação em algo mais fácil de analisar e com mais significado. Além disso, é nesta década que são usadas na prática pela primeira vez, técnicas de aprendizagem estatística para reconhecimento facial em imagens. Provavelmente, o maior desenvolvimento na visão computacional durante este intervalo de tempo foi o aumento da interação com o campo da computação gráfica, especialmente nas áreas de renderização e transformação de imagem, e interpolação da visualização.

Na década de 2000, a interação entre os campos da visão e da computação gráfica continuou a ser aprofundada. Métodos para síntese de textura, acolchoamento (*quilting*) e pintura interna (*inpainting*) emergiram nesta altura, assim como técnicas *feature-based* para reconhecimento de objetos.

Finalmente, a partir de 2010, o desenvolvimento de algoritmos de reconhecimento de imagem foi totalmente revolucionado com a utilização de algoritmos de *machine learning*, devido à construção de conjuntos de dados de larga escala que providenciavam informação rotulada suficiente para a implementação de soluções completas baseadas em *machine learning*. Do mesmo modo, evoluíram também consideravelmente técnicas de *deep learning*, de tal forma que surgiram rapidamente diversos modelos de redes neuronais e um maior número de investigadores a trabalhar nesta área.

Graças aos avanços na área da inteligência artificial e no poder computacional, a tecnologia da visão computacional evoluiu muito e o seu futuro é promissor, sendo expectável que a visão artificial se torne predominante na próxima década [13].

2.1.3 Áreas de aplicação

A visão computacional encontra-se presente atualmente numa enorme variedade de aplicações reais, estando integrada em diversas áreas do quotidiano. Seguidamente, são enumerados alguns dos cenários onde se pode encontrar aplicações para esta tecnologia.

Transporte

A indústria do transporte tem vindo a receber cada vez mais desenvolvimentos tecnológicos para responder às exigências modernas para este sector, com a visão computacional a assumir um papel central.

Um exemplo muito atual é a automação da condução de veículos. A visão computacional é aqui usada para deteção e classificação de objetos (como sinais rodoviários, semáforos e outros veículos), criação de mapas 3D, estimação de movimento, ou ainda para deteção de pessoas, de vias rodoviárias e de espaço livre entre veículos. A Figura 2.4 representa um exemplo da aplicação da visão computacional em veículos autónomos [14, 15].



Figura 2.4: Deteção de objetos e pessoas em veículos autónomos [15].

Outra situação em que já existe uma grande utilização é na deteção de ocupação de parques de estacionamento, como demonstrado na Figura 2.5, servindo como uma alternativa a sistemas baseados em sensores de maior custo e sujeitos a manutenção regular. A tecnologia da visão computacional oferece alta precisão, podendo a identificação de zonas de estacionamento ser combinada com o reconhecimento de matrículas de veículos, possibilitando assim o conhecimento de qual viatura ocupa um determinado lugar num dado período [14].

A monitorização de fluxo de tráfego rodoviário é mais uma finalidade da visão computacional no sector do transporte, cujos algoritmos permitem rastrear e analisar a densidade de tráfego, auxiliando na projeção de melhores sistemas para gestão de

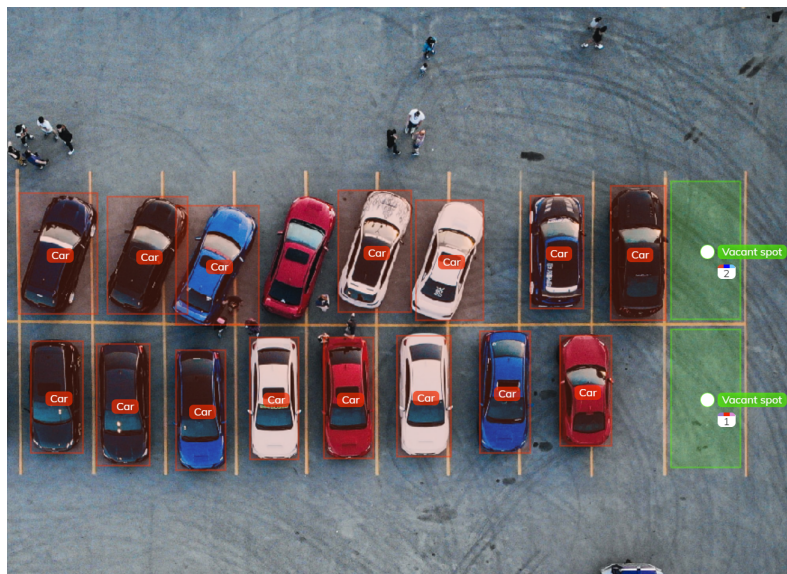


Figura 2.5: Identificação de lugares de estacionamento [14].

tráfego e na melhoria da segurança nas estradas. Exemplo desta aplicação encontra-se ilustrado na Figura 2.6, cujos dados visuais podem ser obtidos através de uma câmara ou de um *drone* [14].

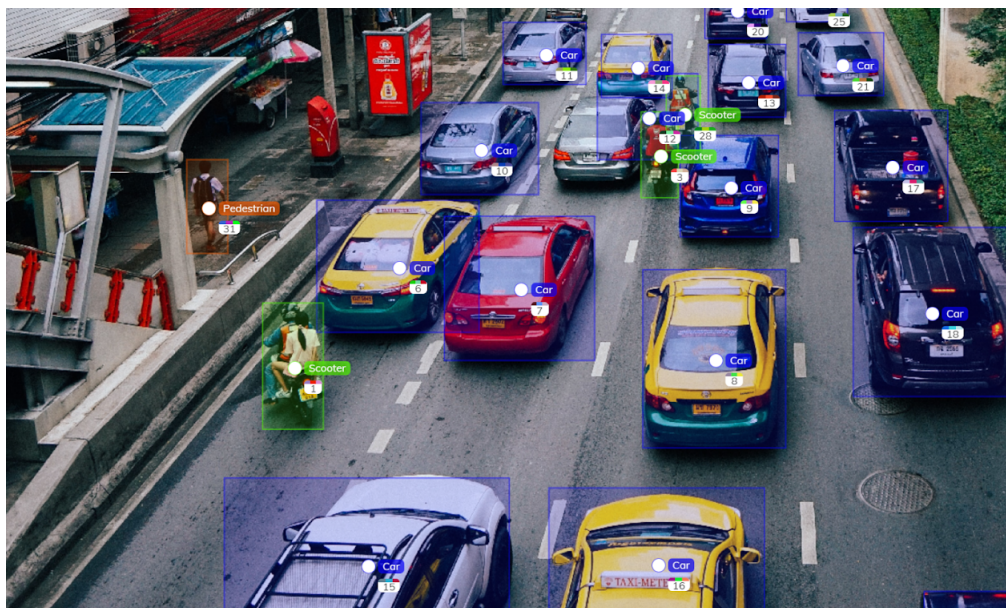


Figura 2.6: Monitorização de tráfego rodoviário [14].

Saúde

A visão computacional tem sido utilizada em diversas aplicações na área da medicina, auxiliando os profissionais médicos na tomada de decisões em relação ao tratamento dos seus pacientes.

A análise de imagens médicas com a tecnologia da visão computacional é um método que providencia um diagnóstico mais preciso através da visualização específica de órgãos e tecidos, o que permite aos médicos e cirurgiões identificar mais facilmente quaisquer problemas ou anormalidades nos seus pacientes. Alguns exemplos incluem imagens obtidas através de radiografia, ecografia e ressonância magnética. Recentemente, aplicações da visão computacional têm contribuído significativamente para o combate à pandemia da Covid-19, não só no diagnóstico e tratamento da doença, mas também no seu controlo e prevenção. A Figura 2.7 apresenta exemplos de análises a exames pulmonares de pacientes, com recurso ao modelo COVID-Net. O protótipo desta aplicação obteve resultados de cerca de 92.4% de precisão no diagnóstico de Covid-19 [16].

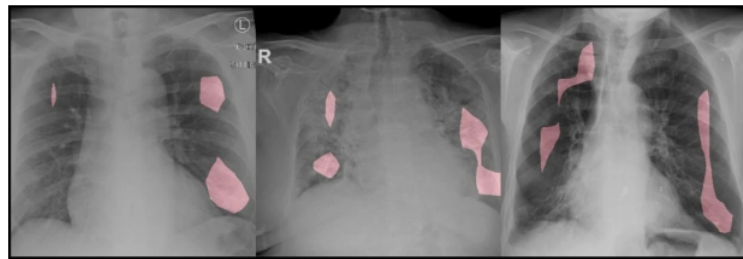


Figura 2.7: Exames pulmonares em combinação com o modelo COVID-Net [16].

Similarmente, a deteção de cancro é mais um exemplo notável da contribuição da visão computacional na área da saúde, assistindo os clínicos na sua identificação com maior rapidez e precisão. Este recurso encontra-se demonstrado na Figura 2.8, que expõe uma biópsia a um gânglio linfático com a localização de uma metástase de cancro da mama. O tecido, além da metástase, contém também macrófagos, que, à primeira vista, parecem tumores mas são zonas normais benignas. O algoritmo aqui empregue consegue detetar com sucesso a região do tumor (identificado a verde brilhante), não sendo “confundido” pelos macrófagos [9,17].

Deteção e reconhecimento facial

Uma das aplicações da visão computacional mais usada é a deteção e reconhecimento facial. Esta tecnologia descobre e assinala uma face humana em imagens digitais. Há diversos casos de uso, entre eles incluem-se os seguintes [18]:

- **Fotografia:** câmaras digitais recentes usam este método para a funcionalidade de autofocus;
- **Videovigilância:** a deteção de rosto é utilizada neste tipo de sistemas para detetar e analisar multidões;

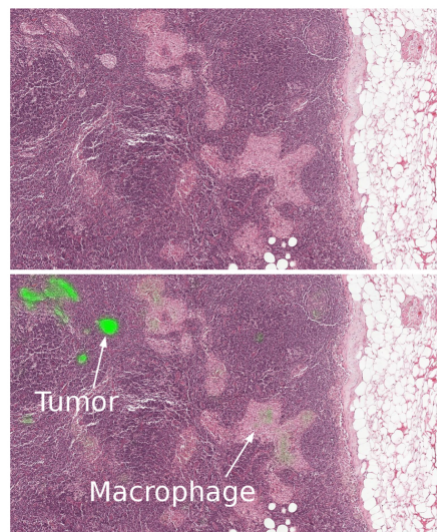


Figura 2.8: Biópsia de gânglio linfático com auxílio da visão computacional [17].

- **Presença:** o reconhecimento facial é utilizado para detetar a presença humana. Na gestão de acessos, é frequentemente combinado com a deteção biométrica;
- **Marketing:** a deteção facial está a tornar-se cada vez mais importante nesta área, tanto para análise do comportamento do cliente, bem como para direcionamento de publicidade.

A Figura 2.9 exhibe um exemplo de aplicação da tecnologia de reconhecimento facial, no qual se pode constatar a deteção inclusive de rostos ocultos e com distintas orientações.

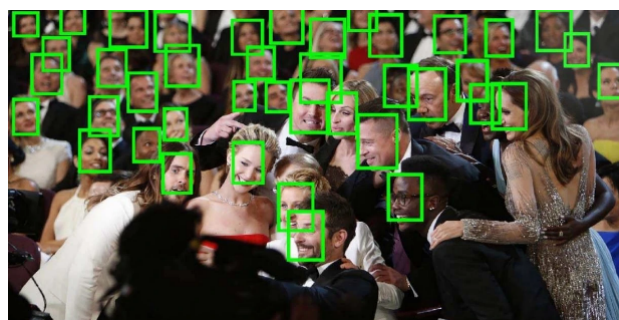


Figura 2.9: Reconhecimento facial [18].

2.2 Processamento de imagem

O processamento de imagem consiste no processo de manipulação de imagens digitais através de algoritmos computacionais, com o objetivo de a transformar ou obter

informação útil a partir da mesma. Este processamento é um passo relevante em diversas aplicações da visão computacional, sendo exemplo sistemas de *deep learning*, onde o pré-processamento pode aumentar drasticamente a *performance* do modelo [19].

2.2.1 Imagem digital

Uma imagem digital é interpretada por um computador como uma matriz multi-dimensional, composta por *picture elements*, também denominados por píxeis. É representada pelas suas dimensões baseadas no número de píxeis que a constitui: largura (número de colunas da matriz) e altura (número de linhas da matriz). Por exemplo, se as dimensões de uma imagem são 400 x 300 (largura x altura), significa que o número total de píxeis da imagem é de 120000 [20].

Um píxel constitui o menor elemento endereçável numa imagem, e é designado por um valor numérico relacionado com a intensidade do brilho/cor visualizável. O número de tons/cores distintas que cada píxel pode representar numa imagem depende do número de *bits* usado pelo pixel, em que cada *bit* adicional duplica a quantidade de cores disponíveis, tal como se verifica na Figura 2.10 [21, 22].



Figura 2.10: Cores disponíveis para cada píxel consoante a sua gama de *bits*.

Um computador “vê” assim uma imagem digital como uma função $I(x, y)$, onde I representa a intensidade do píxel e (x, y) as coordenadas de cada píxel. A Figura 2.11 apresenta o sistema de coordenadas padrão utilizado numa imagem [19].



Figura 2.11: Convenção do sistema de coordenadas usado numa imagem [19].

Imagens binárias

Imagens binárias são aquelas que apenas possuem dois possíveis valores para a intensidade de píxel: 0 (cor preta), e 1 (cor branca). Estas imagens são habitualmente usadas para destacar uma determinada zona de uma imagem colorida. Por exemplo, são utilizadas em processos de segmentação de imagem como demonstra a Figura 2.12.

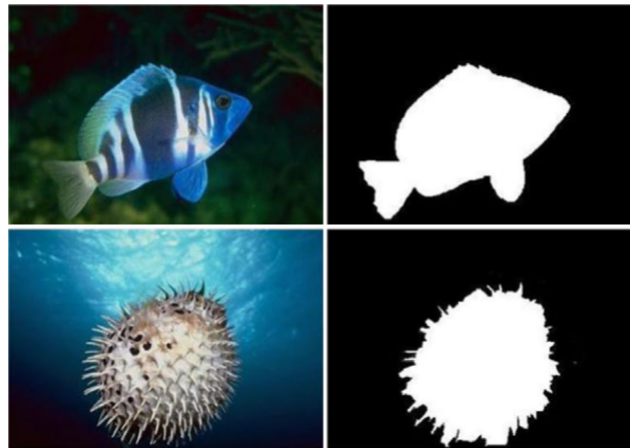


Figura 2.12: Imagens binárias usadas em segmentação [19].

Imagens em escala de cinza

As imagens constituídas por unicamente 256 cores, onde as intensidades de píxel de valores 0 e 255 representam preto e branco respetivamente, são denominadas por imagens de escala de cinza (em inglês, *grayscale images*) ou imagens de 8 *bits*. Os outros 254 valores abrangem os diferentes tons de cinza. Um exemplo deste tipo de imagem e respetiva matriz encontram-se exibidos na Figura 2.13.

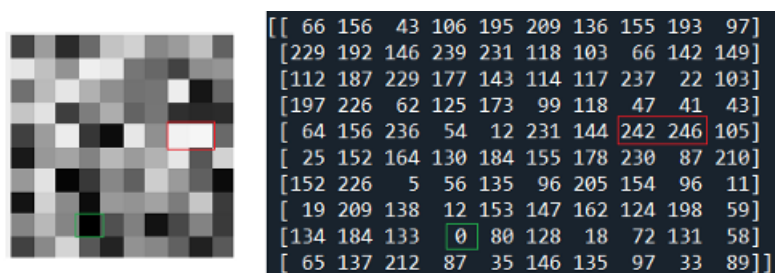


Figura 2.13: Imagem em escala de cinza e respetiva matriz [23].

Imagens de cor RGB

As imagens atualmente mais usadas são as imagens que utilizam o modelo de cor *Red, Green and Blue* (RGB), que possui três canais representativos das cores vermelha,

verde e azul de uma imagem. Cada píxel engloba três matrizes de valores inteiros de 8 *bits* correspondentes aos três canais de cores, o que possibilita deste modo a exibição de 16777216 cores para cada píxel. Assim, neste tipo de imagem, um píxel de cor preta tem como valor (0, 0, 0) e um de cor branca tem (255, 255, 255). A Figura 2.14 revela uma imagem RGB com a demonstração de como o computador a interpreta [22].

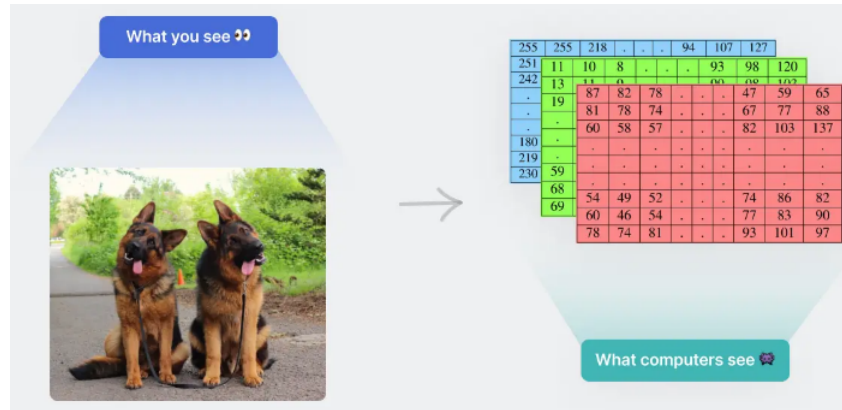


Figura 2.14: Imagem RGB e sua interpretação pelo computador [19].

Imagens de cor RGBA

Imagens *Red, Green, Blue and Alpha* (RGBA) são imagens RGB com um canal extra nomeado de “*alpha*”. Este canal retrata a propriedade de opacidade/transparência da imagem, variando o seu valor entre 0% e 100% [19].

2.2.2 Convolução

A convolução é uma das operações mais críticas e fundamentais na visão computacional e no processamento de imagem, sendo a base de múltiplos algoritmos, tais como de *smoothing/blurring*, nitidez, detecção de bordas, entre outros [21].

O processo de convolução reside na adição de cada elemento da imagem aos seus elementos vizinhos, ponderados por uma matriz de convolução denominada por *kernel* ou por máscara. O *kernel* é uma matriz de dimensão inferior à da imagem, e pode ter um tamanho arbitrário de $M \times N$ píxeis, sendo obrigatoriamente M e N valores inteiros ímpares, para garantir um valor no centro do *kernel*. A Figura 2.15 ilustra um *kernel* com valor ao centro válido e outro com centro inválido. Constatase que a matriz em (a) apresenta o centro localizado em $(x = 1, y = 1)$, considerando que o canto superior esquerdo da matriz é usado como origem e as coordenadas são indexadas com início a zero. Por outro lado, a matriz em (b) terá o centro situado em $(x = 0.5, y = 0.5)$, correspondendo a coordenadas inválidas uma vez que estas

têm de ser valores inteiros. Esta é a razão pela qual as dimensões do *kernel* são definidas com inteiros ímpares.

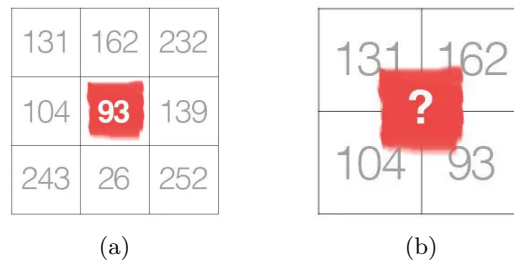


Figura 2.15: Centro do *kernel*: (a) *kernel* 3 x 3 com centro válido e (b) *kernel* 2 x 2 sem centro válido [21].

Em processamento de imagem, a convolução requer três componentes:

- Uma imagem de entrada;
- Uma matriz de *kernel* que será aplicada à imagem de entrada;
- Uma imagem de saída que armazena o resultado da convolução da imagem de entrada com o *kernel*.

As etapas no processo de convolução são as seguintes:

1. Inverter as linhas e colunas da matriz do *kernel*;
2. Colocar o **centro** do *kernel* em cada coordenada da matriz da imagem de entrada;
3. Multiplicar elemento a elemento da região da imagem com o *kernel*, e somar os valores resultantes das operações de multiplicação;
4. Inserir o resultado na mesma coordenada na matriz da imagem de saída.

Um exemplo visual é mostrado na Figura 2.16 para melhor entendimento da operação de convolução, descrevendo o cálculo mencionado anteriormente para um píxel da matriz.

Para o caso dos píxeis dispostos nos limites da matriz da imagem de entrada, alguns elementos do *kernel* ficam situados fora da imagem, visto que o centro do *kernel* deverá estar alinhado com estes píxeis. Nestas situações, é comum a adição de uma camada de píxeis em redor da matriz da imagem para cálculo da convolução, normalmente com valores nulos, utilizando a técnica conhecida como “*zero-padding*”, demonstrada na Figura 2.17. Desta forma, a imagem de saída mantém o mesmo tamanho da imagem de entrada, não havendo redução da informação.

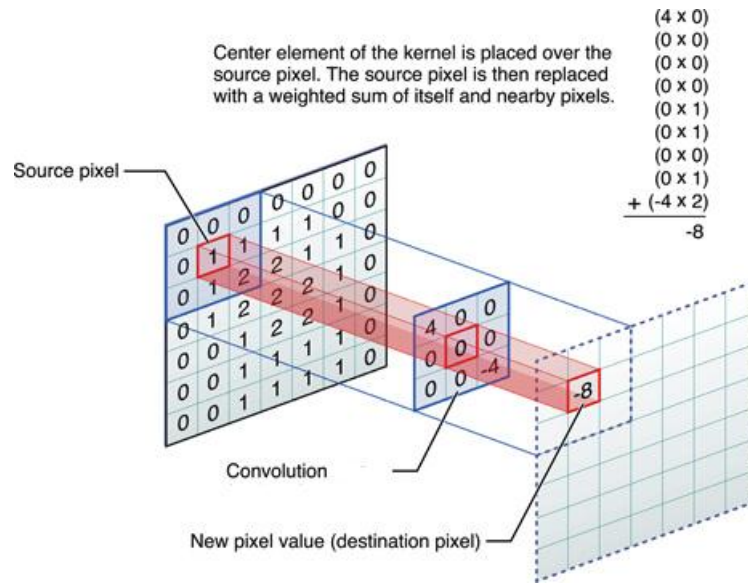


Figura 2.16: Operação de convolução [24].

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel		
0	-1	0
-1	5	-1
0	-1	0

114	328	-26	470	158
53	266	-61	-30	344
403	116	-47	295	244
108	-135	256	-128	344
314	346	279	153	421

Figura 2.17: Convolução usando *zero-padding* [25].

2.2.3 Smoothing

A operação de *smoothing*, vulgarmente intitulada também como *blurring*, provoca visualmente uma desfocagem e perda de nitidez numa imagem. Apesar de este efeito ser geralmente indesejado em fotografias, é bastante útil em tarefas de processamento de imagem. Na realidade, a suavização/desfoque é uma das fases de pré-processamento mais frequentes em visão computacional e processamento de imagem.

Existem várias razões para a sua utilização, mas a mais habitual é para redução de conteúdo de alta-frequência, como ruído e extremidades. Deste modo, o “detalhe” de uma imagem é diminuído, o que auxilia na procura de objetos de interesse, permitindo ainda o foco nos objetos estruturais maiores de uma imagem [26, 27].

Os principais métodos para *smoothing* são seguidamente explorados.

Average filter

O *average filter* ou *normalized box filter* é o filtro mais simples de todos. Este filtro gera a média numa área de píxeis ao redor de um píxel central, e substitui o píxel central pelo valor médio, atenuando a sua amplitude [28].

Um *kernel* representativo deste filtro é o descrito na Equação 2.1, no qual cada entrada está uniformemente ponderada. À medida que a dimensão da matriz do *kernel* aumenta, a imagem torna-se progressivamente mais desfocada. Isto pode levar a um ponto em que se perdem as extremidades de importantes objetos estruturais na imagem. Assim, no desenvolvimento de aplicações de visão computacional, é fundamental a seleção da quantidade de *smoothing* pretendida.

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (2.1)$$

A Figura 2.18 apresenta exemplos de aplicação deste filtro.

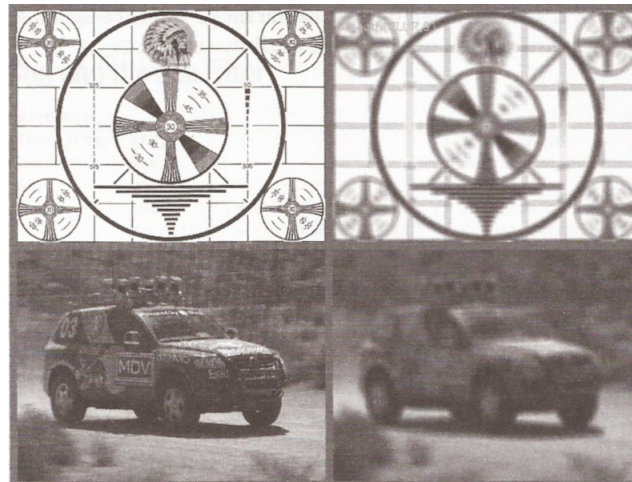


Figura 2.18: Aplicação do *average filter*: imagens originais à esquerda; imagens de saída à direita [26].

Filtro gaussiano

O filtro gaussiano tem como finalidade remover ruído seguindo aproximadamente uma distribuição gaussiana. Com este método, a imagem torna-se menos *blurred* do que usando o *average filter*, possibilitando uma maior preservação de bordas existentes na imagem conforme a ponderação usada no filtro [27].

Este filtro envolve a convolução de cada ponto da imagem com um *kernel* gaussiano, onde os píxeis vizinhos que estão mais próximos ao píxel central possuem maior contribuição, como se depreende do exemplo de um *kernel* deste filtro, com dimensão retangular, descrito na Equação 2.2.

$$K = \frac{1}{141} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 7 & 26 & 41 & 26 & 7 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \quad (2.2)$$

O resultado da aplicação do filtro gaussiano encontra-se exemplificado na Figura 2.19.

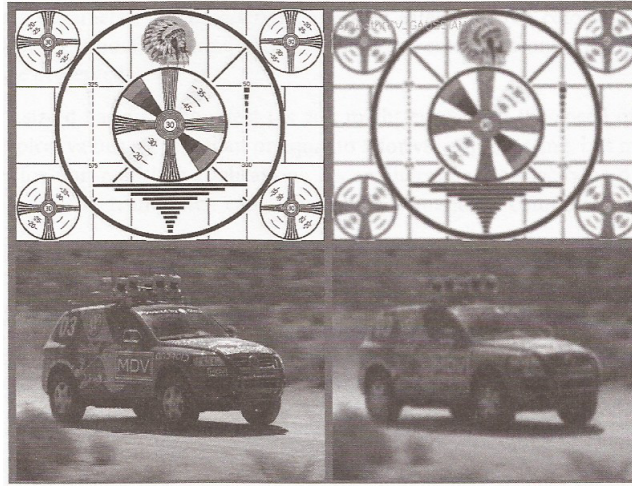


Figura 2.19: Aplicação do filtro gaussiano: imagens originais à esquerda; imagens de saída à direita [26].

Median filter

O *median filter* permite substituir cada píxel pela mediana dos elementos da sua vizinhança, ao contrário do *average filter*, que substitui pela média. Desta forma, os píxeis têm sempre algum valor já existente na imagem, e como a mediana é robusta para valores discrepantes, o ruído é menos influente do que em outro método estatístico, como a média. A Figura 2.20 ilustra exemplos de resultados usando este tipo de filtro [27].

Bilateral filter

Outra forma de *smoothing* é denominada por *bilateral filtering*. Conhecida também como *edge-preserving smoothing*, esta operação permite a redução de ruído com manutenção das extremidades dos objetos na imagem, introduzindo duas distribuições gaussianas. A primeira distribuição considera apenas vizinhos espaciais, isto é, píxeis que surgem próximos no espaço de coordenadas (x, y) da imagem. A segunda função gaussiana modela a intensidade do píxel da vizinhança, garantindo que apenas píxeis com amplitude semelhante sejam incluídos no cálculo real da filtragem. Por um lado, se píxeis na mesma vizinhança apresentarem um valor similar, representam hipoteticamente o mesmo objeto. Todavia, se dois píxeis na mesma vizinhança

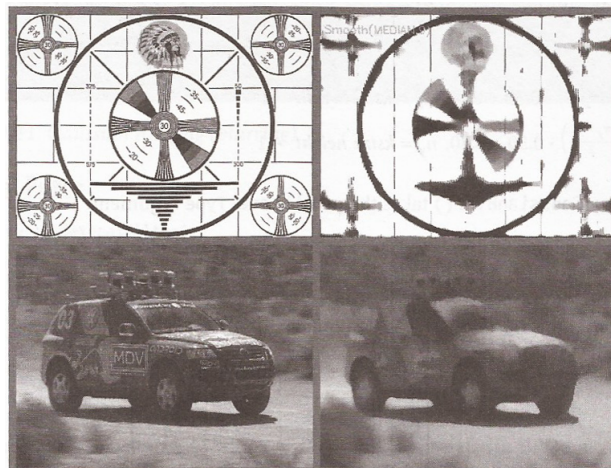


Figura 2.20: Aplicação do *median filter*: imagens originais à esquerda; imagens de saída à direita [26].

possuem valores distintos, pode significar a presença de uma borda ou limite de um objeto. Assim, este filtro é capaz de preservar as extremidades numa imagem, enquanto simultaneamente reduz ruído. Contudo, apresenta a desvantagem de um maior tempo de processamento comparando com outros métodos de filtragem [27].

Exemplos de utilização do *bilateral filtering* encontram-se demonstrados na Figura 2.21.

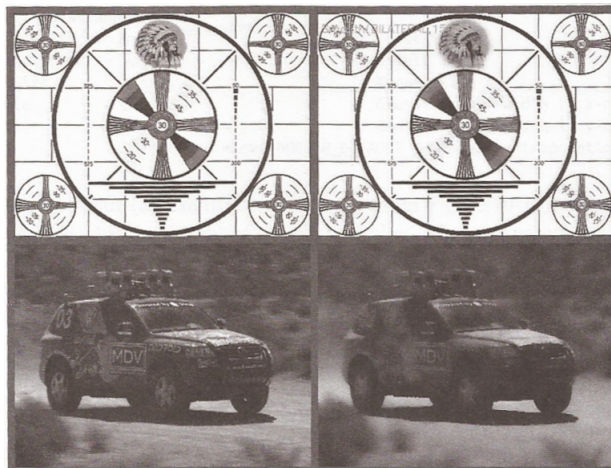


Figura 2.21: Aplicação do *bilateral filter*: imagens originais à esquerda; imagens de saída à direita [26].

2.2.4 Operações de *threshold*

Uma operação de *threshold* consiste na binarização de uma imagem, ou seja, na conversão de uma imagem em escala de cinza para binária. Pode ser considerada como um método simples de segmentação, em que separa os objetos ou áreas particulares

de interesse numa imagem e elimina os pontos que não são relevantes, emergindo em diferentes formas: *thresholding* simples, dinâmico e adaptativo [26, 29].

Thresholding simples

Em *thresholding* simples, o parâmetro de *threshold* é manualmente fornecido para segmentação da imagem. Este método funciona bem em condições de luminosidade controladas, onde se assegura alto contraste entre os objetos de interesse e o *background* da imagem. Esta categoria contém cinco diferentes métodos de *thresholding*: binário, binário invertido, truncado, *threshold* para zero, e *threshold* para zero invertido. Para ilustração de como estes processos funcionam, considera-se uma imagem de entrada com a intensidade de píxeis definida como $src(x, y)$, representada no gráfico da Figura 2.22, no qual a linha a azul fixa o valor de *threshold*.

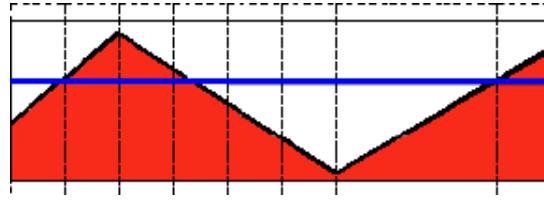


Figura 2.22: Gráfico de intensidade dos píxeis da imagem para *threshold* [30].

O método de *threshold* binário pode ser definido através da Equação 2.3, em que $dst(x, y)$ é a imagem de saída e max é o valor selecionado como máximo. Deduz-se portanto que quando a amplitude de um píxel da imagem é maior que o *threshold*, o seu valor é alterado para max . Caso contrário, é mudado para zero.

$$dst(x, y) = \begin{cases} max, & \text{se } src(x, y) > threshold \\ 0, & \text{senão} \end{cases} \quad (2.3)$$

O *threshold* binário invertido corresponde exatamente ao processo inverso do *threshold* binário, sendo expresso pela Equação 2.4. Assim um píxel da imagem é definido como zero se a sua intensidade for superior ao *threshold*, senão é alterado para max .

A Figura 2.23 exhibe os resultados da utilização do *threshold* binário e do binário invertido.

$$dst(x, y) = \begin{cases} 0, & \text{se } src(x, y) > threshold \\ max, & \text{senão} \end{cases} \quad (2.4)$$

O processo de *threshold* truncado permite a limitação das intensidades dos píxeis da imagem cujas amplitudes são maiores que o *threshold*, mantendo as intensidades

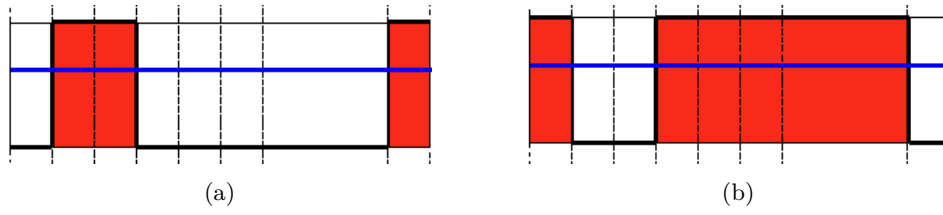


Figura 2.23: Resultados da aplicação do: (a) *threshold* binário e (b) *threshold* binário invertido [30].

dos outros píxeis. A Equação 2.5 expressa este comportamento, comprovado pelo resultado contido na Figura 2.24.

$$dst(x, y) = \begin{cases} threshold, & \text{se } src(x, y) > threshold \\ src(x, y), & \text{senão} \end{cases} \quad (2.5)$$

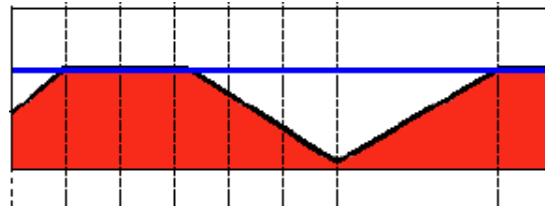


Figura 2.24: Resultado da aplicação do *threshold* truncado [30].

O método de *threshold* para zero pode ser declarado pela Equação 2.6, em que se infere que quando a amplitude de um píxel da imagem é menor que o *threshold*, o seu valor é alterado para zero, não modificando os outros píxeis.

$$dst(x, y) = \begin{cases} src(x, y), & \text{se } src(x, y) > threshold \\ 0, & \text{senão} \end{cases} \quad (2.6)$$

Comportamento inverso apresenta o *threshold* para zero invertido, expresso pela Equação 2.7, no qual um píxel é definido como zero quando este possui intensidade superior ao *threshold*.

Resultados da aplicação dos processos do *threshold* para zero e do seu inverso estão expostos na Figura 2.25.

$$dst(x, y) = \begin{cases} 0, & \text{se } src(x, y) > threshold \\ src(x, y), & \text{senão} \end{cases} \quad (2.7)$$

Thresholding dinâmico

O processo de *thresholding* dinâmico procura de forma dinâmica o parâmetro de *threshold*, sendo o seu valor ótimo automaticamente calculado baseado na imagem.

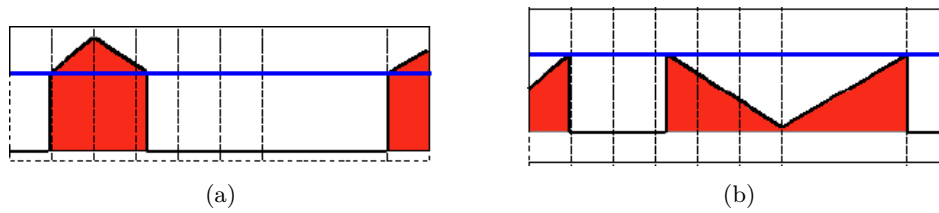


Figura 2.25: Resultados da aplicação do: (a) *threshold* para zero e (b) *threshold* para zero invertido [30].

Para imagens simples com condições de luminosidade controladas, pode ser suficiente a definição de um valor *hardcoded*, porém, em inúmeras aplicações reais, não é possível o conhecimento a priori dessas mesmas condições.

O algoritmo de Otsu é um método bastante popular que se encaixa nesta categoria de *thresholding*. Resumidamente, este algoritmo considera todos os valores possíveis para o *threshold*, e calcula a variância de cada uma de duas classes de píxeis: classe com píxeis abaixo do *threshold* e classe com píxeis acima do mesmo. O valor ótimo de *threshold* equivale àquele para o qual a variância entre classes é mínima. Devido à exaustiva pesquisa pelo espaço de possíveis *thresholds*, este algoritmo não é particularmente rápido [26, 29].

***Thresholding* adaptativo**

Os métodos referidos anteriormente são técnicas globais para *thresholding*, implicando que o mesmo valor de *threshold* seja usado para testar todos os píxeis da imagem de entrada. No entanto, um único *threshold* poderá funcionar numa determinada zona da imagem, e falhar totalmente noutra região da mesma.

O *thresholding* adaptativo, muitas vezes designado como *thresholding* local, pretende solucionar este tipo de problemas. O parâmetro de *threshold* é aqui calculado em pequenas partes, isto é, a imagem é “partida” em reduzidas regiões em que cada uma é sujeita a *thresholding* separadamente. Esta técnica destaca-se quando há diferentes níveis de iluminação em toda a imagem, distribuindo-se de forma não-uniforme. Assim, o valor ótimo de *threshold* poderá divergir para as várias partes da imagem. Este método baseia-se na análise estatística dessas regiões locais da imagem para determinar o *threshold* ótimo, sendo prática comum o uso da média aritmética ou da média gaussiana como estatística para este objetivo. No caso da média aritmética, cada píxel na vizinhança contribui igualmente para a computação do *threshold*, enquanto que na média gaussiana, os píxeis mais afastados da coordenada do centro da região têm uma menor contribuição para o seu cálculo [26, 31].

Seguidamente, é efetuada uma exposição de resultados gerados pelos diferentes tipos de *thresholding*, utilizando a imagem de entrada representada na Figura 2.26, com a finalidade de segmentação do logótipo e do texto. Os resultados estão manifestados na Figura 2.27, a partir dos quais se pode concluir que, para esta imagem:

o *thresholding* simples extrai o logótipo e texto, mas vem acompanhado de bastante ruído; o algoritmo de Otsu segmenta apenas parte do logótipo, perdendo-se também o texto; e o *thresholding* adaptativo produz melhor segmentação da imagem, obtendo-se o texto e o logótipo quase completo.



Figura 2.26: Imagem original para *thresholding* adaptativo [31].

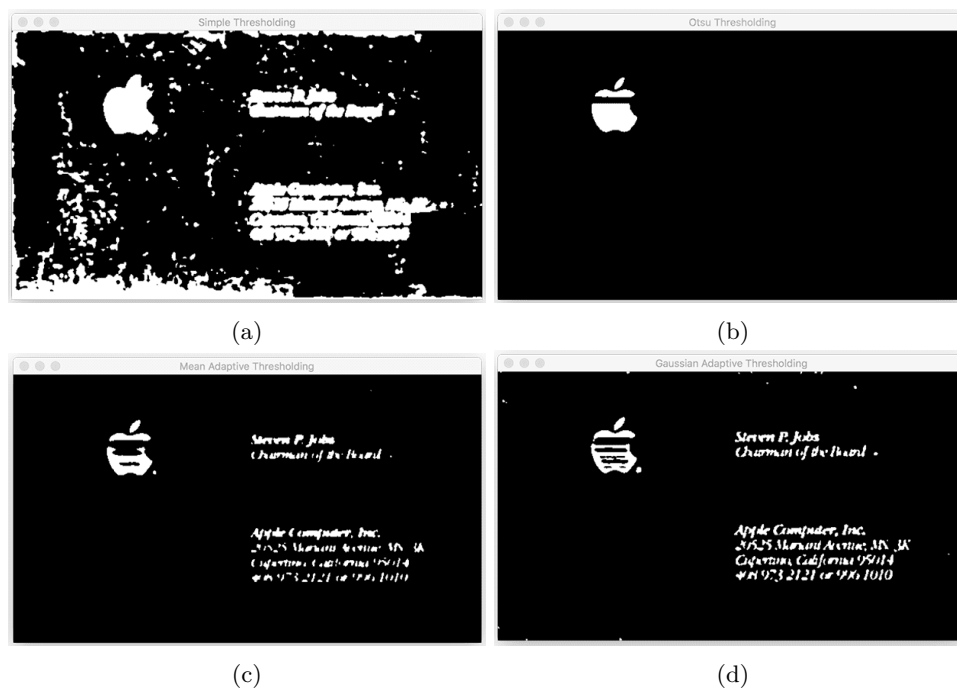


Figura 2.27: Resultados da aplicação do: (a) *threshold* simples, (b) *threshold* baseado no algoritmo de Otsu, (c) *threshold* adaptativo com média aritmética e (d) *threshold* adaptativo com média gaussiana [30].

2.2.5 Transformações morfológicas

As transformações morfológicas são operações que foram desenvolvidas ao longo dos anos, a maior parte para um propósito específico, mas algumas dessas foram

encontrando uma ampla coleção de utilidades. Estas operações de processamento de imagem são aplicadas em imagens em escala de cinza ou binárias, e são usadas em algoritmos de *Optical Character Recognition* (OCR), detecção de códigos de barras, detecção de matrículas, entre muitas outras aplicações. Em algumas situações, o uso inteligente das transformações morfológicas poderá evitar a utilização de algoritmos de *machine learning* e de *deep learning* mais complicados e computacionalmente dispendiosos.

Essencialmente, todas as transformações morfológicas são baseadas em apenas duas operações primitivas, a dilatação e a erosão, gerando operações mais sofisticadas, como *opening*, *closing*, gradiente morfológico, *top hat* e *black hat*. Estas operações examinam uma imagem com um *kernel*, também designado neste contexto como *structuring element* [26,32].

Dilatação e erosão

A dilatação e a erosão constituem as operações morfológicas mais básicas, e são aplicadas em diversos contextos como remoção de ruído, isolamento individual de elementos, e junção de elementos díspares.

A dilatação consiste na convolução de uma imagem com um *kernel*, de qualquer forma ou tamanho, sendo usualmente quadrado ou circular. À medida que o *kernel* “desliza” sobre a imagem, verifica-se qual o valor máximo de píxel sobreposto pelo *kernel* e substitui-se o píxel da imagem que se encontra na posição de âncora (normalmente, ao centro do *kernel*), por esse máximo. Desta forma, esta operação de maximização faz com que as regiões brilhantes numa imagem aumentem, sendo expressa pela Equação 2.8.

$$dst(x, y) = \max(src(x + i, y + j)), (i, j) \in kernel \quad (2.8)$$

A erosão equivale à operação inversa da dilatação, em que substitui a intensidade de píxel da imagem no ponto de âncora pelo valor mínimo encontrado na área do *kernel*, tal como define a Equação 2.9. Assim, as regiões brilhantes numa imagem tendem a diminuir numa erosão, aumentando as zonas mais escuras.

$$dst(x, y) = \min(src(x + i, y + j)), (i, j) \in kernel \quad (2.9)$$

A Figura 2.28 ilustra um exemplo de resultado aplicando a dilatação e a erosão.

Opening e closing

As operação de *opening* e *closing* são combinações simples dos operadores de dilatação e de erosão. No caso de *opening*, é gerada pela aplicação da erosão seguida por uma dilatação, sendo útil para remoção de pequenas regiões brilhantes ou para

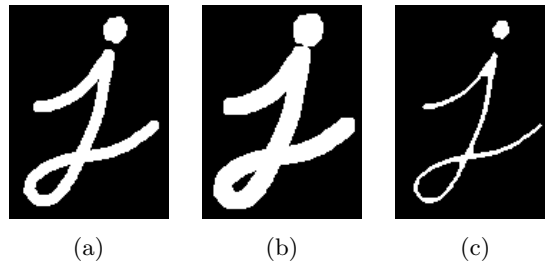


Figura 2.28: Transformações morfológicas básicas: (a) imagem de entrada, (b) aplicação da dilatação e (c) aplicação da erosão [33].

contagem de elementos. Um exemplo de aplicação desta transformação morfológica encontra-se exibido na Figura 2.29.



Figura 2.29: Operação morfológica de *opening*: imagem original à esquerda; resultado da aplicação à direita [34].

A operação de *closing* é obtida pela dilatação da imagem, com uma posterior erosão, permitindo a conexão de elementos e redução de pequenas regiões escuras, tal como exemplifica a Figura 2.30.



Figura 2.30: Operação morfológica de *closing*: imagem original à esquerda; resultado da aplicação à direita [34].

Gradiente morfológico

O operador de gradiente morfológico corresponde à subtração da dilatação com a erosão de uma imagem, tal como expressa a Equação 2.10. O efeito produzido por esta transformação é deixar a borda de um objeto na imagem, ilustrado no exemplo da Figura 2.31.

$$\text{gradiente}(\text{src}) = \text{dilate}(\text{src}) - \text{erode}(\text{src}) \quad (2.10)$$



Figura 2.31: Operação morfológica de gradiente morfológico: imagem original à esquerda; resultado da aplicação à direita [34].

Top hat e black hat

As transformações morfológicas de *top hat* e de *black hat* são usadas para revelar, respetivamente, regiões brilhantes sobre um fundo escuro ou regiões escuras sobre um fundo brilhante, a partir de uma imagem. Em termos matemáticos, a operação de *top hat* pode ser expressa pela diferença entre a imagem de entrada e a sua operação de *opening* (Equação 2.11), enquanto a de *black hat* é retratada pela subtração da operação de *closing* da imagem original e ela própria (Equação 2.12).

$$tophat(src) = src(src) - open(src) \quad (2.11)$$

$$blackhat(src) = close(src) - src \quad (2.12)$$

Para exemplificar uma aplicação destas transformações, considera-se uma situação de deteção de matrícula de um veículo, que contém letras e dígitos escuros sobre uma região clara. Aplicando as operações de *top hat* e de *black hat*, resultam, respetivamente, as imagens contidas na Figura 2.32 e na Figura 2.33. De referir que a imagem de entrada neste exemplo foi, antes da aplicação dos operadores, convertida para a escala de cinza, e que os resultados foram obtidos utilizando um *kernel* de dimensão adequada para o problema em questão, neste caso de 13 píxeis de largura e 5 de altura.

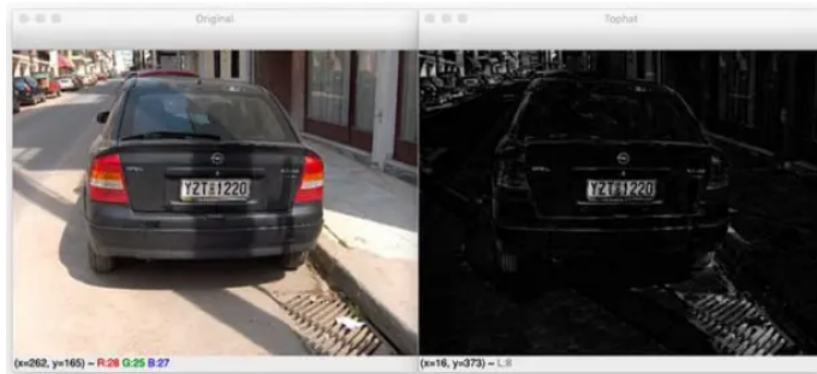


Figura 2.32: Operação morfológica de *top hat*: imagem original à esquerda; resultado da aplicação à direita [32].

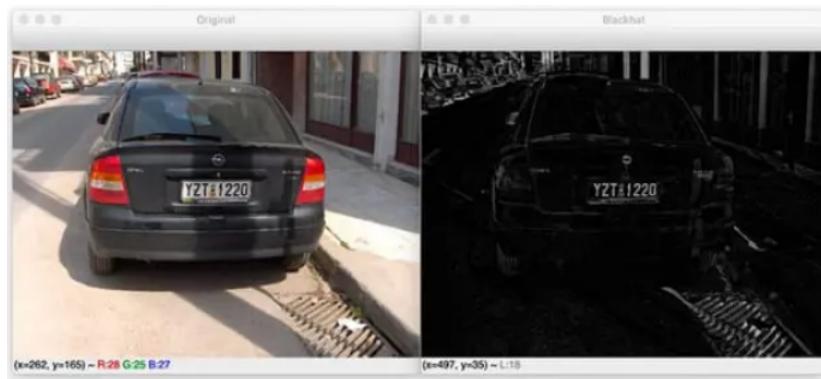


Figura 2.33: Operação morfológica de *black hat*: imagem original à esquerda; resultado da aplicação à direita [32].

2.3 Reconhecimento de circuitos elétricos

A visão computacional é um recurso de grande utilidade no reconhecimento de circuitos elétricos a partir de imagens. Devido às inúmeras possibilidades de manipulação dos diferentes algoritmos oferecidos pelo campo da visão computacional, existem várias metodologias consideradas no desenvolvimento e implementação de uma aplicação capaz de processar, segmentar e identificar os elementos presentes num esquema elétrico.

Geralmente, as etapas seguidas num processo de reconhecimento de circuitos elétricos são as incluídas no diagrama da Figura 2.34: pré-processamento da imagem do circuito, segmentação do circuito elétrico, e classificação dos diferentes elementos contidos no circuito.



Figura 2.34: Diagrama geral para reconhecimento de circuitos elétricos a partir de uma imagem.

2.3.1 Pré-processamento

Para uma segmentação do circuito mais eficiente, é necessário que a imagem seja pré-processada, removendo possíveis fontes de ruído e preparando a imagem para posterior análise. Esta etapa é de extrema importância, visto que afeta significativamente a eficiência e o sucesso da segmentação.

O pré-processamento inclui as tarefas principais descritas no seguinte procedimento [35]:

1. Conversão do tipo de imagem, isto é, imagens de cor são transformadas em imagens de escala de cinza;

2. Filtragem da imagem, utilizando filtros *smoothing*, para eliminação de ruído;
3. Conversão da imagem de escala de cinza para binária, através de uma operação de *threshold*.

O tipo de filtro aplicado na imagem é variável consoante a aplicação. Enquanto alguns estudos científicos propõem a utilização de um *median filter* [36], outros adotam o filtro gaussiano [37, 38] para supressão de ruído. Quanto à conversão da imagem para binária, é igualmente diversificado o tipo de operação de *threshold* aplicado: *thresholding* global, no qual o mesmo valor de *threshold* é usado em toda a imagem [35]; *thresholding* dinâmico recorrendo ao algoritmo de Otsu, onde se calcula automaticamente o valor ótimo de *threshold* baseado na imagem [39]; e *thresholding* adaptativo, permitindo a utilização de distintos *thresholds* em diferentes regiões da imagem [37, 38, 40, 41]. Esta variedade é justificada pela diversidade de funcionalidades das aplicações para reconhecimento de circuitos elétricos, influenciando assim nos algoritmos a serem aplicados para pré-processamento da imagem.

Em adição às tarefas referidas para pré-processamento, é comum a execução de outras durante esta fase, nomeadamente da transformação morfológica de dilatação e de um processo de esqueletização. A dilatação permite eliminar descontinuidades e irregularidades, juntando assim fragmentos que possam existir na imagem. A operação de esqueletização possibilita a redução de cada linha na imagem para um píxel, sendo útil para a contagem de píxeis vizinhos brilhantes para cada píxel, o que auxilia na descoberta de pontos terminais de objetos na imagem [37]. A Figura 2.35 exemplifica imagens intermédias geradas durante o pré-processamento de um circuito.

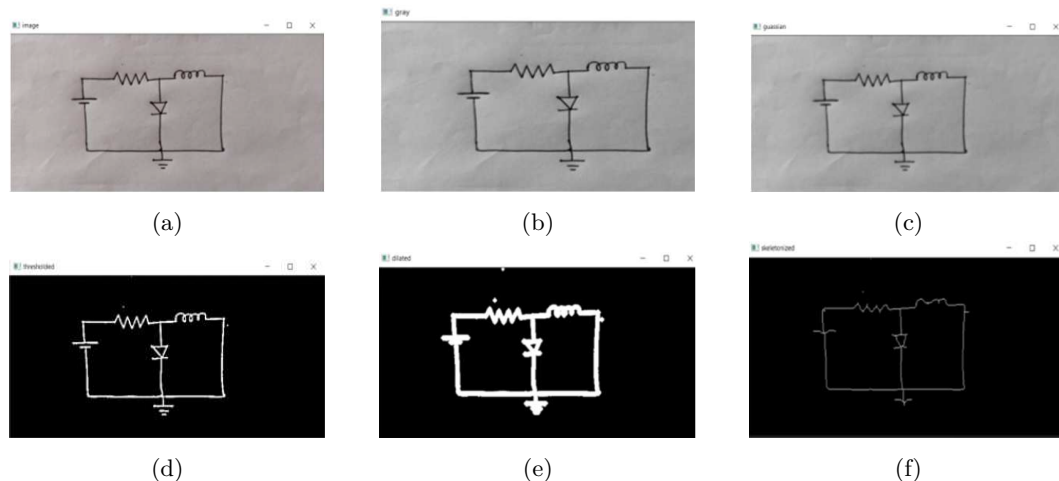


Figura 2.35: Exemplo de pré-processamento de imagem: (a) imagem de entrada, (b) imagem em escala de cinza, (c) filtro gaussiano, (d) imagem binária (*thresholding*), (e) dilatação e (f) esqueletização [37].

2.3.2 Segmentação

Após o pré-processamento da imagem, a etapa que se segue é a segmentação do circuito. A segmentação consiste no processo de extração dos elementos presentes no circuito a partir da imagem, englobando componentes, nós e conexões entre estes.

Uma metodologia considerada para a segmentação de componentes de um circuito elétrico é baseada na forma dos seus símbolos. De acordo com este método, os componentes eletrônicos podem ser categorizados em três grupos diferentes tendo em conta a sua estrutura geométrica: símbolos com formas fechadas, símbolos com linhas conectadas, e símbolos com linhas desconectadas. A Figura 2.36 apresenta exemplos de componentes pertencentes a cada uma das categorias [35].

Component Category	Example Components	Symbols
Closed Shape Components	AC Voltage	
	DC Voltage	
	Lamp	
	Diode	
Components with Connected Lines	Resistor	
	Inductor	
Components with Disconnected Lines	Battery	
	Switch	
	Ground	
	Capacitor	

Figura 2.36: Metodologia de segmentação categorizando componentes [35].

Cada grupo de componentes requer técnicas diferentes para segmentação, dependendo das suas características. Os componentes com formas fechadas têm a particularidade de estarem rodeados por um segmento de linha à sua volta, logo este atributo pode ser usado para extrair este tipo de geometria. O preenchimento de regiões fechadas destaca estes símbolos, podendo ser aplicada uma transformação morfológica de *opening*, com um *structuring element* adequado, para eliminação das conexões e dos outros componentes. Assim, os elementos restantes podem ser considerados como pertencendo a esta categoria, tal como demonstra a Figura 2.37.

Como os componentes com formas fechadas já foram detetados, estes podem ser removidos (regiões que ocupavam são definidas com píxeis nulos) da imagem de

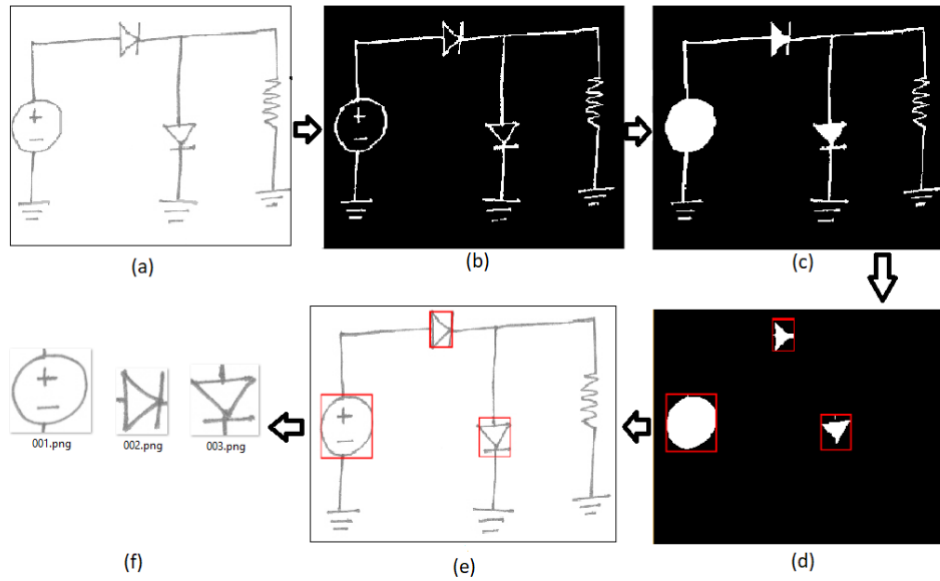


Figura 2.37: Segmentação de componentes com formas fechadas: (a) imagem original, (b) imagem binária, (c) imagem após preenchimento de regiões, (d) detecção de componentes depois de *opening*, (e) localização de componentes na imagem original e (f) componentes segmentados [35].

processamento. Uma operação morfológica de *closing* é aplicada ao circuito para dilatação dos componentes com linhas conectadas, formando as vulgarmente apelidadas de bolhas. A transformação de *opening* é novamente utilizada para remover o resto do circuito, o que deixa apenas os símbolos dilatados anteriormente. A Figura 2.38 expõe o resultado deste processo.

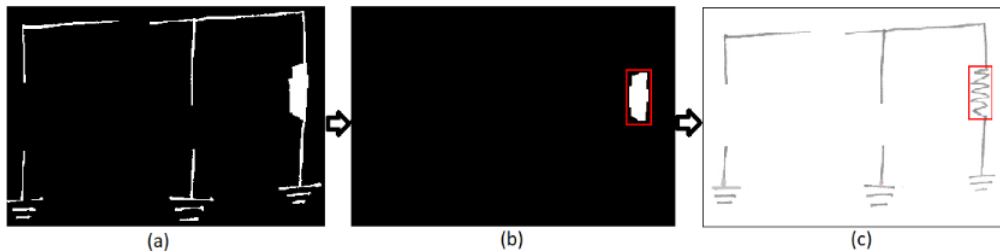


Figura 2.38: Segmentação de componentes com linhas conectadas: (a) imagem após *closing*, (b) detecção de componentes depois de *opening* e (c) localização de componentes na imagem original [35].

O último grupo de componentes, cujas linhas estão desconectadas, detém mais complexidade, já que circuitos desenhados sobretudo à “mão” podem variar a distância entre as linhas destes elementos, dificultando a definição do *structuring element* para transformações morfológicas. No caso do exemplo da metodologia aqui em estudo, é aplicada uma operação de espessamento seguida de *thinning* (Figura 2.39), resultando na conexão das linhas desconectadas dos componentes, e posteriormente

são removidas as restantes ligações do circuito.

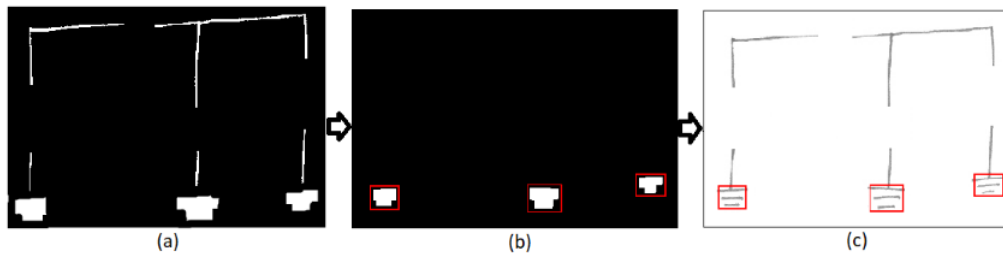


Figura 2.39: Segmentação de componentes com linhas desconectadas: (a) imagem após espessamento e *thinning*, (b) remoção das restantes ligações do circuito e (c) localização de componentes na imagem original [35].

Nem todos os métodos para deteção de componentes os divide em diferentes categorias. Há técnicas em que aplica apenas uma transformação morfológica de *closing*, o que necessita de uma seleção da dimensão de *structuring element* suficientemente grande para que “caibam” os componentes, formando bolhas nos elementos. Nesta metodologia, uma operação de esqueletização deverá ser realizada durante o pré-processamento, para que, após a operação de *closing*, os fios das ligações do circuito possam ser removidos eficientemente [42].

Além da utilização de algoritmos simples de visão computacional para a segmentação de componentes, também é possível a aplicação de metodologias de *deep learning* com recurso a redes neuronais, detetando e localizando os diferentes componentes presentes no circuito [41].

Diversas abordagens podem também ser seguidas na extração dos nós e das interligações entre os elementos do circuito. Uma das técnicas possíveis é a segmentação de nós e ligações após a deteção dos componentes do circuito. Os componentes detetados são eliminados da imagem original do circuito, deixando em aberto as suas ligações. Utilizando um método de busca de contornos, os fios podem ser localizados, e, uma vez que os componentes segmentados estavam conectados a esses fios, ao fazer a interseção entre os fios e os componentes, é possível obter os pontos de conexão (terminais) dos componentes bem como a que conexão cada componente está ligado. Um exemplo deste processo está ilustrado na Figura 2.40 [37].

A metodologia mencionada tem a capacidade de extrair as conexões, porém não é possível saber a localização exata dos nós do circuito. Este detalhe pode ser obtido utilizando um outro método baseado na operação de *Hough transform*. Esta transformação tem a capacidade de detetar linhas retas, existindo também uma variante para descobrir círculos [43, 44]. Assim, posteriormente à remoção dos componentes segmentados do circuito, procede-se à deteção das linhas horizontais e verticais existentes no mesmo, correspondentes às conexões do circuito. Os nós equivalem à interseção entre as retas horizontais e verticais, como mostra a Figura 2.41 [41].

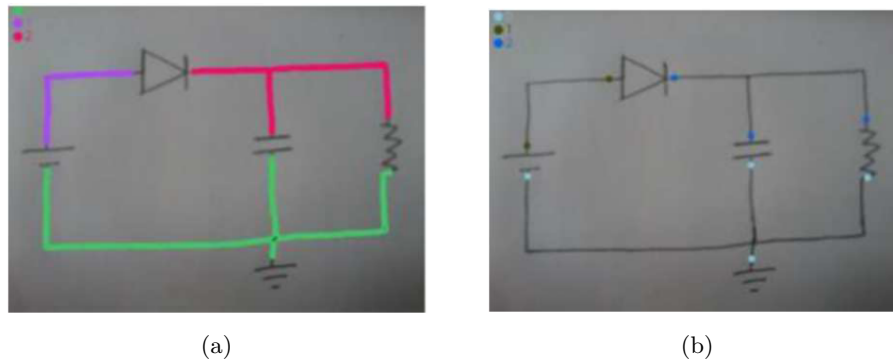


Figura 2.40: Segmentação das conexões do circuito: (a) identificação de fios de conexão e (b) detecção de terminais dos componentes [37].

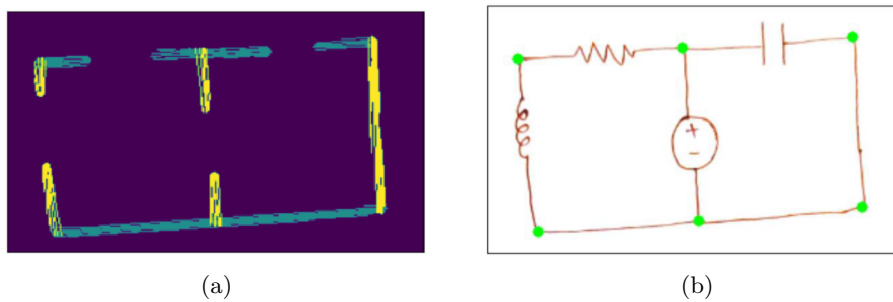


Figura 2.41: Segmentação dos nós do circuito: (a) detecção de conexões através de *Hough transform* e (b) identificação de nós [41].

2.3.3 Classificação

A classificação dos componentes do circuito é a última etapa numa aplicação para reconhecimento de esquemas elétricos. Frequentemente, é usada uma combinação do método de *Histogram of Oriented Gradients* (HOG) com o de *Support Vector Machine* (SVM). O conceito de HOG refere-se a um histograma construído usando os gradientes e a orientação de píxel, que usa vetores para representação de direção calculados para pequenas secções da imagem, gerando um histograma único para cada região da mesma. Assim, as *features* são extraídas e alimentam o modelo de *machine learning* SVM. Este modelo é previamente treinado para análise das características fornecidas pelo histograma e, baseando-se nelas, consegue prever os elementos [37,38].

Outra metodologia de classificação de componentes bastante utilizada é baseada em inteligência artificial, nomeadamente em redes neuronais, utilizando modelos treinados com uma grande quantidade de dados, fornecendo excelentes resultados [39,41].

2.4 Ferramentas de *software* de visão computacional

A adoção de ferramentas de *software* de visão computacional tem vindo a sofrer um aumento cada vez mais acentuado por parte da indústria tecnológica, em sectores como saúde, segurança, agricultura, automóvel, industrial, entre muitos outros. Esta amplificação tem sido acompanhada pelo desenvolvimento de numerosas plataformas, *toolkits* e bibliotecas de *software* relacionadas com o campo da visão computacional [45].

Existem diversas ferramentas de *software* que facultam algoritmos para processamento de imagem e para *machine learning*, contudo, no contexto do presente projeto, é mais relevante o acesso ao primeiro. Algumas bibliotecas são mais completas do que outras em termos de variedade de metodologias que oferecem para processamento de imagem, assim é fundamental a verificação das capacidades de cada uma. Além disso, as linguagens de programação que suporta (no caso deste projeto, deverá ter suporte para C++), bem como a disponibilização de uma boa documentação da sua *Application Programming Interface* (API), são outros aspetos a ter em consideração no momento da escolha da mesma. Seguidamente, são apresentadas de forma breve algumas bibliotecas C++ que providenciam algoritmos para processamento de imagem.

2.4.1 OpenCV: *Open Source Computer Vision Library*

OpenCV é uma biblioteca de *software open source* de visão computacional e de *machine learning*. Foi lançada por Gary Bradski, em 1999, enquanto trabalhava na Intel Corporation, com a esperança de acelerar a evolução da visão computacional e da inteligência artificial, providenciando uma sólida e comum infraestrutura para quem exercia funções nestes campos. Esta ferramenta foi desenhada tendo em mente a eficiência computacional, e com um foco forte em aplicações de tempo real, tirando vantagem de processadores *multicore* [26].

Esta ferramenta possui mais de 2500 algoritmos otimizados, incluindo um conjunto abrangente tanto de algoritmos clássicos como de algoritmos de última geração. Estes podem ser utilizados para identificação de objetos, deteção e reconhecimento facial, classificação de ações humanas em vídeos, extração de movimentos e de modelos 3D de objetos, etc. OpenCV é uma biblioteca imensamente popular e de uso gratuito [46], tendo mais de 47 mil pessoas na comunidade de utilizadores e um número estimado de *downloads* que excede os 18 milhões, sendo usada extensivamente por empresas como a Microsoft, Google, Intel, IBM, Sony, Honda, Toyota ou Yahoo, por grupos de investigação e também por órgãos governamentais [47].

A biblioteca OpenCV está escrita nativamente em C++, mas fornece múltiplas interfaces para outras linguagens de programação, designadamente para Python, Java e MATLAB, e suporta os sistemas operativos Windows, Linux, Android e

MacOS. Apresenta uma estrutura modular que contém uma série de bibliotecas *shared* e *static*, com os seguintes módulos principais [26, 46, 48]:

- **Core functionality (módulo “core”)**: define as estruturas básicas de dados, como o denso *array* multidimensional “Mat”, e funções básicas utilizadas pelos outros módulos;
- **Image processing (módulo “imgproc”)**: permite transformações básicas sobre imagens, como filtragem linear e não-linear, transformações geométricas, conversão de espaços de cor, histogramas, entre outros;
- **Image Input/Output (I/O) (módulo “imgcodecs”)**: funções para leitura e escrita de imagens;
- **Camera calibration and 3D reconstruction (módulo “calib3d”)**: contém algoritmos de geometria de visualização múltipla, calibração de câmara *single* e *stereo*, estimativa de pose de objeto, correspondência *stereo*, e elementos de reconstrução 3D;
- **2D features framework (módulo “features2d”)**: possui algoritmos para detecção, descrição e correspondência de pontos-chave de *features*;
- **Object detection (módulo “objdetect”)**: possibilita a detecção de objetos e instâncias de classes pré-definidas (por exemplo, rostos, olhos, pessoas e veículos);
- **High-level Graphical User Interface (GUI) (módulo “highgui”)**: API simples de utilizar que oferece funções para interfaces gráficas, nomeadamente para visualização de imagens ou para receção de simples ações do utilizador (por exemplo, clicar num botão);
- **Video analysis (módulo “video”)**: tem funções para leitura e escrita de vídeos, e inclui estimação de movimento, subtração de *background* e algoritmos de rastreamento de objetos;
- **Video I/O (módulo “videoio”)**: interface para captura de vídeo e *video codecs*;
- **Machine Learning (módulo “ml”)**: é na realidade uma biblioteca própria, que contém um vasto conjunto de algoritmos de *machine learning* implementados de forma a serem compatíveis com as estruturas de dados de OpenCV;
- **Deep neural network (módulo “dnn”)**: providencia o carregamento de modelos de redes neurais a partir de diferentes *frameworks*;
- **Computational photography (módulo “photo”)**: faculta métodos úteis para fotografia computacional.

2.4.2 Simd

Simd é uma biblioteca gratuita *open source* de processamento de imagem e de *machine learning*, implementada com algoritmos otimizados utilizando processamento paralelo do tipo *Single Instruction, Multiple Data* (SIMD). Foi desenhada para programadores C e C++, e providencia várias técnicas de elevada eficiência para processamento de imagem, disponibilizando funcionalidades como [49]:

- Filtragem de imagem;
- Redimensionamento de imagem;
- Extração de informações estatísticas a partir de imagens;
- Detecção de movimento;
- Detecção e classificação de objetos;
- Redes neurais.

2.4.3 Boost GIL

Boost *Generic Image Library* (GIL) é uma biblioteca C++ *header-only* que abstrai a representação de imagens dos seus algoritmos, e está incorporada no conjunto de bibliotecas de *software* do popular projeto Boost. Algumas das suas capacidades são [50]:

- Convolução 2D;
- Filtragem de imagem;
- Operações de *threshold* em imagem;
- Criação de histogramas de imagem;
- Transformações morfológicas;
- Redimensionamento de imagem.

2.4.4 Magick++

Magick++ é uma API em C++ para a biblioteca de *software* gratuita de processamento de imagem ImageMagick [51].

Esta interface faculta diversas funcionalidades amplamente úteis para processamento de imagem, entre elas fazem parte as seguintes [52]:

- Filtragem de imagem;

- Operações de *threshold* em imagem;
- Detecção de bordas;
- Desenho de formas e inserção de texto numa imagem;
- Conversão de formatos de imagem;
- Extração de características e reconhecimento de padrões numa imagem;
- Redimensionamento, rotação, recorte e desbaste de imagem.

Capítulo 3

Arquitetura

Neste capítulo exibem-se a lista de requisitos do módulo de *software* a desenvolver e a arquitetura geral que este deverá seguir. Além disso, são aqui também abordados os diferentes blocos de *software* necessários e suas interligações para a implementação deste projeto.

3.1 Requisitos

O módulo de *software* a desenvolver neste trabalho deverá ter a capacidade de interpretar e modelar circuitos elétricos a partir de uma imagem, baseando-se em algoritmos de visão computacional. A sua implementação deverá cumprir o seguinte conjunto de requisitos funcionais:

1. **Receção e leitura da imagem do circuito elétrico** a segmentar através de um ficheiro com formato de imagem, devendo suportar vários formatos como, por exemplo, *Portable Network Graphics* (PNG), *Joint Photographic Experts Group* (JPEG) e *Bitmap* (BMP);
2. **Segmentação do circuito elétrico** da imagem, utilizando métodos de visão computacional que permitam a deteção e extração dos elementos presentes no mesmo, nomeadamente:
 - Componentes;
 - Nós;

- Conexões entre os elementos;
 - *Labels* associadas a cada elemento.
3. **Geração de um ficheiro de tipo *JavaScript Object Notation* (JSON)** com o modelo do circuito, isto é, contendo o mapa/matriz resultante da segmentação do esquema elétrico, agrupando os elementos do circuito detetados e suas interligações. A sua estrutura deve respeitar o modelo descrito na Secção 3.2;
 4. **Geração de ficheiros de imagem** com a ROI de cada componente e *label* segmentada, que serão posteriormente utilizados por um outro módulo de *software* responsável pela classificação dos componentes segmentados e pelo reconhecimento de caracteres alfanuméricos presentes nas *labels*.

3.2 Modelo JSON

O módulo de *software* proposto para este projeto deve gerar um mapa de segmentação, que contenha todos os elementos detetados e segmentados a partir da imagem de entrada do circuito. A estrutura deste mapa tem de seguir a de um modelo JSON compatível com a aplicação de edição de circuitos da ferramenta U=RI solve [5, 53], para permitir que este mapa possa ser importado e usado por este editor, facilitando assim a possível futura integração na aplicação U=RI solve do módulo desenvolvido neste trabalho. Nesta secção, é apresentada de forma breve a organização deste modelo de esquema elétrico, que se encontra estruturado em diferentes tipos de elementos: básicos, secundários e principais.

Os tipos básicos são tipos simples auxiliares criados para remover a redundância dos tipos mais complexos, incluindo-se dentro deste grupo os tipos seguintes:

- *ID*: corresponde a uma *string* de identificação de cada elemento do circuito elétrico, devendo ser única para que o elemento possa ser facilmente identificado no esquema elétrico. Um exemplo de ID válido equivale a "*8c0168ec-07fe-4b2c-96ee-4292ab34cfb2*";
- *Position*: é composto pelos campos definidos na Listagem 3.1 (descritos em TypeScript [54] para facilitar a sua representação), e refere-se à posição em que se encontra localizado cada elemento no esquema elétrico, existindo duas formas para a especificar:
 - *Global*: a posição é relativa à origem da imagem do circuito (canto superior esquerdo corresponde às coordenadas $(x = 0, y = 0)$);
 - *Relative*: a posição é relativa à origem do elemento “pai”, variando o valor das suas coordenadas entre 0 e 1.

```
1 interface Position {
2     x: number;
3     y: number;
4     angle?: number;
5 }
6 type GlobalPosition = Position;
7 type RelativePosition = Position;
```

Listagem 3.1: Tipo básico *Position*.

Os tipos secundários do modelo descrevem pequenas estruturas utilizadas pelos tipos principais, reduzindo assim a sua complexidade, e fazem uso dos tipos básicos. Nesta categoria fazem parte:

- *Port*: consiste no ponto de conexão de um *Component* (tipo principal), permitindo ao seu *owner* estar conectado ao circuito elétrico. Os campos pertencentes a este tipo estão declarados na Listagem 3.2;

```
1 interface Port {
2     id: ID;
3     type?: 'hybrid' | 'input' | 'output';
4     position: RelativePosition;
5     owner?: ID;
6     connection?: ID;
7 }
```

Listagem 3.2: Tipo secundário *Port*.

- *Label*: é responsável pelo armazenamento do texto que diferencia visualmente cada elemento, cuja estrutura está apresentada na Listagem 3.3. Note-se que o módulo de *software* do presente projeto deve detetar e associar *labels* (se existirem) a cada elemento do circuito, no entanto, como não é responsável pela decodificação do texto que esta contém, os campos *name*, *value* e *unit* deverão ser sempre definidos com *strings* vazias.

```
1 interface Label {
2     id: ID;
3     name: string;
4     value: string;
5     unit: string;
6     position: Position;
7     isNameHidden?: boolean;
8     isValueHidden?: boolean;
9     owner?: ID;
10 }
```

Listagem 3.3: Tipo secundário *Label*.

Relativamente aos tipos principais, estes constituem os tipos mais complexos do modelo e contêm a maior parte dos dados. Este grupo abrange os seguintes tipos:

- *Component*: representa qualquer espécie de componente elétrico (uma resistência, uma bobine, um condensador, etc). A sua estrutura encontra-se demonstrada na Listagem 3.4;

```
1 interface Component {
2     id: ID;
3     type: string;
4     fullName: string;
5     position: GlobalPosition;
6     label: Label;
7     ports: Port[];
8 }
```

Listagem 3.4: Tipo principal
Component.

- *Connection*: simboliza uma conexão física entre os *ports* de componentes (do tipo *Component*) e os nós do circuito (do tipo *Node*), através dos campos *start* e *end* incluídos na Listagem 3.5;

```

1 interface Connection {
2     id: ID;
3     start: ID;
4     end: ID;
5     label: Label;
6 }

```

Listagem 3.5: Tipo principal *Connection*.

- *Node*: representa um ponto no circuito que acumula múltiplas conexões (ao contrário do *Port* que apenas pode conter uma conexão), sendo referenciada a sua estrutura na Listagem 3.6.

```

1 interface Node {
2     id: ID;
3     type?: 'real' | 'virtual';
4     position: GlobalPosition;
5     label: Label;
6     connections?: ID[];
7 }

```

Listagem 3.6: Tipo principal *Node*.

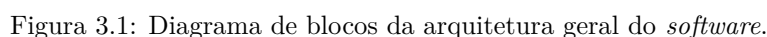
3.3 Diagrama de blocos

A arquitetura geral do *software* encontra-se representada no diagrama de blocos da Figura 3.1, sendo os blocos assinalados a verde os que se pretendem implementar no módulo deste projeto:

- Receção da imagem;
- Pré-processamento;
- Segmentação;
- *Regions of interest*;
- Mapa de segmentação.

A aplicação, no seu conjunto final, deverá funcionar da seguinte forma:

1. O utilizador fornece um ficheiro de imagem do circuito elétrico a segmentar, que deverá ser recebido e lido pela aplicação;



- ¹A presente dissertação não abrange os procedimentos descritos neste ponto. No entanto, a sua descrição é relevante para a contextualização do trabalho desenvolvido.

7. O ficheiro com o mapa completo descritivo do modelo do circuito elétrico é importado pelo editor de circuitos da *framework* U=RIolve, visualizando-se graficamente o circuito elétrico presente na imagem inicial. Posteriormente, o circuito poderá ser analisado pela aplicação U=RIolve¹.

Capítulo 4

Implementação

Este capítulo apresenta o desenvolvimento realizado para a implementação do módulo de *software* deste projeto. A estrutura de organização e o funcionamento do *software* são aqui detalhadamente explanados.

4.1 Organização do *software*

O módulo de *software* deste projeto foi desenvolvido utilizando a linguagem de programação C++, devido não só à familiarização com esta linguagem, mas também por ser uma linguagem multi-paradigma que proporciona uma programação eficiente e de grande desempenho. Este *software* encontra-se estruturado em vários diretórios:

- *cmake*: contém um ficheiro CMake útil para disponibilização de bibliotecas externas que são necessárias ao projeto;
- *docker*: contém ficheiros para a geração de uma imagem Docker [55], para possibilitar a compilação e a execução do *software* usando esta plataforma;
- *docs*: contém documentação relacionada com o mapa de segmentação gerado por este *software*, bem como resultados para alguns exemplos de circuitos;
- *doxygen*: contém a configuração para geração de documentação a partir do código-fonte usando a ferramenta doxygen [56];
- *scripts*: contém *shell scripts* para automatizar e facilitar o uso de diversas ferramentas utilizadas durante o desenvolvimento do *software*, relacionadas

em particular com a cobertura, documentação, formatação e análise estática de código;

- *src*: contém o código-fonte do módulo de *software* deste projeto;
- *tests*: contém ficheiros relacionados com testes unitários realizados no desenvolvimento do *software*.

Para utilização deste módulo, é necessário a instalação na máquina de algumas ferramentas externas, bem como a execução de determinados comandos para a sua compilação. A enumeração dessas ferramentas e as instruções a realizar pelo utilizador estão descritas no guia disponibilizado no Anexo B.

4.2 Funcionamento do *software*

O código-fonte do *software* desenvolvido encontra-se no diretório *src* do projeto. Está distribuído por diferentes pastas dentro de *src*, permitindo a sua divisão consoante a sua função, cujo nome da pasta onde um ficheiro se encontra é igual ao *namespace* do código desse ficheiro, contido num *namespace* global denominado por *circuitSegmentation*. Os diretórios/*namespaces* e classes/módulos pertencentes a este *software* são listados seguidamente:

- *main*: contém a função *main* que inicia o programa;
- *application*: *namespace* que inclui os seguintes módulos dedicados à configuração do programa:
 - *Application*: classe responsável pela execução do programa, contendo também algumas informações como nome, versão e data da aplicação;
 - *CommandLineParser*: classe responsável pela verificação das opções passadas pela linha de comandos quando o programa é iniciado;
 - *Config*: configuração da opção de abertura de janelas com as imagens obtidas durante o processamento do programa, útil na fase de desenvolvimento e *debug* do *software*, uma vez que permitiu ajustar e verificar a influência de cada algoritmo de visão computacional na imagem do circuito (opção desativada por *default*).
- *circuit*: *namespace* que engloba os seguintes módulos relativos aos tipos do modelo JSON:
 - *Component*: estrutura representativa do tipo *Component* do modelo;
 - *Connection*: estrutura representativa do tipo *Connection* do modelo;
 - *Id*: módulo representativo do tipo *ID* do modelo;

- *Label*: estrutura representativa do tipo *Label* do modelo;
- *Node*: estrutura representativa do tipo *Node* do modelo;
- *Port*: estrutura representativa do tipo *Port* do modelo;
- *Position*: estrutura base e suas derivadas *GlobalPosition* e *RelativePosition* representativas do tipo *Position* do modelo.
- *cmdLineParser*: *namespace* que contém:
 - *CmdLineParser*: classe responsável pela análise dos argumentos inseridos pelo utilizador na linha de comandos para execução do *software*.
- *common*: *namespace* que incorpora:
 - *UuidGen*: classe que permite gerar UUID para cada elemento no modelo JSON do circuito.
- *computerVision*: *namespace* que inclui:
 - *OpenCvWrapper*: classe que interage com a biblioteca OpenCV.
- *imageProcessing*: *namespace* que engloba as seguintes classes referentes ao processamento de imagem:
 - *ImagePreprocessing*: classe responsável pelo pré-processamento de imagem;
 - *ImageProcManager*: classe responsável pela gestão de todo o processamento de imagem;
 - *ImageReceiver*: classe responsável pela receção da imagem de entrada do programa;
 - *ImageSegmentation*: classe responsável pela segmentação da imagem.
- *logging*: *namespace* que contém:
 - *Logger*: classe responsável pelo registo de mensagens geradas durante a execução do programa.
- *schematicSegmentation*: *namespace* que abarca os seguintes módulos usados para a segmentação do circuito elétrico:
 - *ComponentDetection*: classe responsável pela deteção de componentes presentes no circuito;
 - *ConnectionDetection*: classe responsável pela deteção de conexões entre os componentes e de nós do circuito;
 - *LabelDetection*: classe responsável pela deteção de *labels* do circuito;

- *RoiSegmentation*: classe responsável pela geração de imagens com ROI para cada componente e *label* segmentada;
- *SchematicSegmentation*: classe responsável pela detecção de *ports* dos componentes detetados, associação de *labels* aos elementos do circuito, e pelo armazenamento dos elementos segmentados;
- *SegmentationMap*: classe responsável pela geração e escrita do mapa de segmentação para um ficheiro JSON;
- *SegmentationUtils*: contém diversas funções úteis usadas durante o processo de segmentação do circuito.

4.2.1 Início do programa

O fluxograma da Figura 4.1 apresenta o procedimento geral executado pelo programa, cujo início é realizado através da função *main*. O programa começa por verificar as opções passadas pela linha de comandos, e efetua posteriormente o processamento de imagem. Contudo, estas operações são externas à função *main*, visto que esta contém o código descrito na Listagem 4.1. Como se pode constatar, nesta função é apenas criada uma instância da classe *Application* (linha 3) e chamado o método *exec* dessa instância (linha 4), delegando a responsabilidade de execução do programa para o objeto *app* criado.

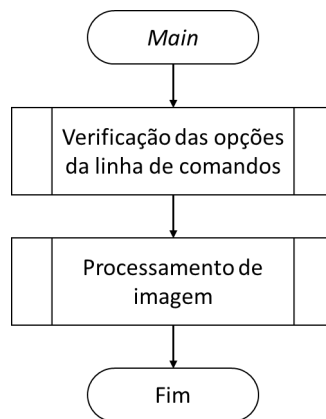


Figura 4.1: Fluxograma geral de funcionamento do programa.

```

1 int main(int argc, char const* argv[])
2 {
3     circuitSegmentation::application::Application app;
4     return app.exec(argc, argv);
5 }
  
```

Listagem 4.1: Função *main* do programa.

A Figura 4.2 ilustra um diagrama de sequência do procedimento efetuado pelo método *exec*, com a indicação das interações entre as classes utilizadas neste método. De acordo com o diagrama, este processo está dividido nas duas fases anteriormente mencionadas: verificação das opções passadas pela linha de comandos, e processamento de imagem. Na primeira fase, os argumentos da linha de comandos são recolhidos e analisados pelo *CommandLineParser* (classe detalhada na Secção 4.2.2), sendo posteriormente verificados quais argumentos foram passados de forma a definir o comportamento posterior do programa. As opções da linha de comandos que são verificadas por este *software* são as seguintes:

- “-V” ou “--verbose”: ativa mensagens de *log*;
- “-h” ou “--help”: apresenta uma mensagem de ajuda;
- “-i” ou “--image”: caminho do ficheiro da imagem do circuito a processar;
- “-s” ou “--save-proc”: guarda imagens obtidas durante o processamento no diretório atual (ficheiros gerados descritos no Anexo C). Salienta-se que as imagens com ROI são sempre guardadas independentemente da passagem deste parâmetro;
- “-v” ou “--version”: apresenta a versão da aplicação.

Todas as opções mencionadas são opcionais, excetuando a opção “-i” ou “--image” que é a única obrigatória a ser passada para informar ao programa da localização do ficheiro de imagem do circuito. Caso esta não esteja presente, não faz sentido o programa continuar e por isso termina, acontecimento que também ocorre quando são passadas as opções “-h” ou “--help”, e “-v” ou “--version”.

Na segunda fase do método *exec*, é realizado o processamento da imagem de entrada do circuito, utilizando a classe *ImageProcManager* (classe abordada na Secção 4.2.5). No momento em que o processamento de imagem termina, o programa chega ao seu fim.

4.2.2 Analisador da linha de comandos

Neste projeto foi desenvolvido um analisador de opções da linha de comandos, implementado na classe *CmdLineParser*. É um analisador genérico que, além da verificação se determinados argumentos foram passados, também permite a impressão na linha de comandos de uma mensagem de ajuda e da versão da aplicação, informações que devem ser passadas através de métodos próprios desta classe.

O *CommandLineParser* utiliza a classe *CmdLineParser*, tal como demonstra o diagrama da Figura 4.3, para verificar as opções específicas para este programa durante a respetiva fase mencionada anteriormente pertencente ao método *exec*. Os

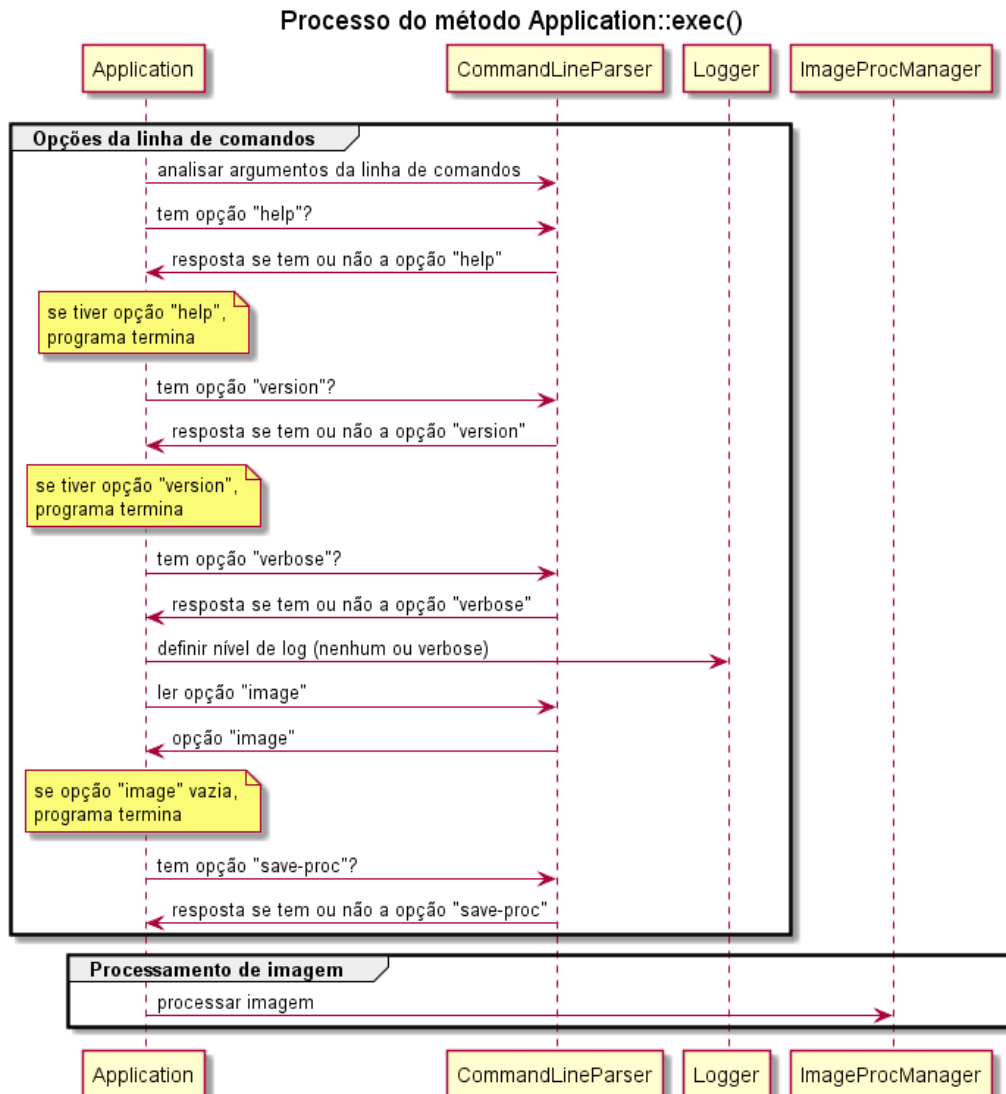


Figura 4.2: Diagrama de sequência do método `Application::exec`.

parâmetros da linha de comandos são armazenados nesta classe, e podem posteriormente ser verificados recorrendo aos métodos `hasOption` (constatar se está presente a opção) ou `getOption` (ler o valor da opção).

4.2.3 *Logger*

O *software* contém a implementação de um *logger* simples que permite o registo de mensagens ao longo da execução do programa. Este registo é efetuado numa *stream* de saída do tipo C++ `std::ostream`, sendo utilizado neste projeto a *stream* de saída para a consola, `std::cout`.

Este *logger* apresenta os seguintes níveis para o registo de mensagens:

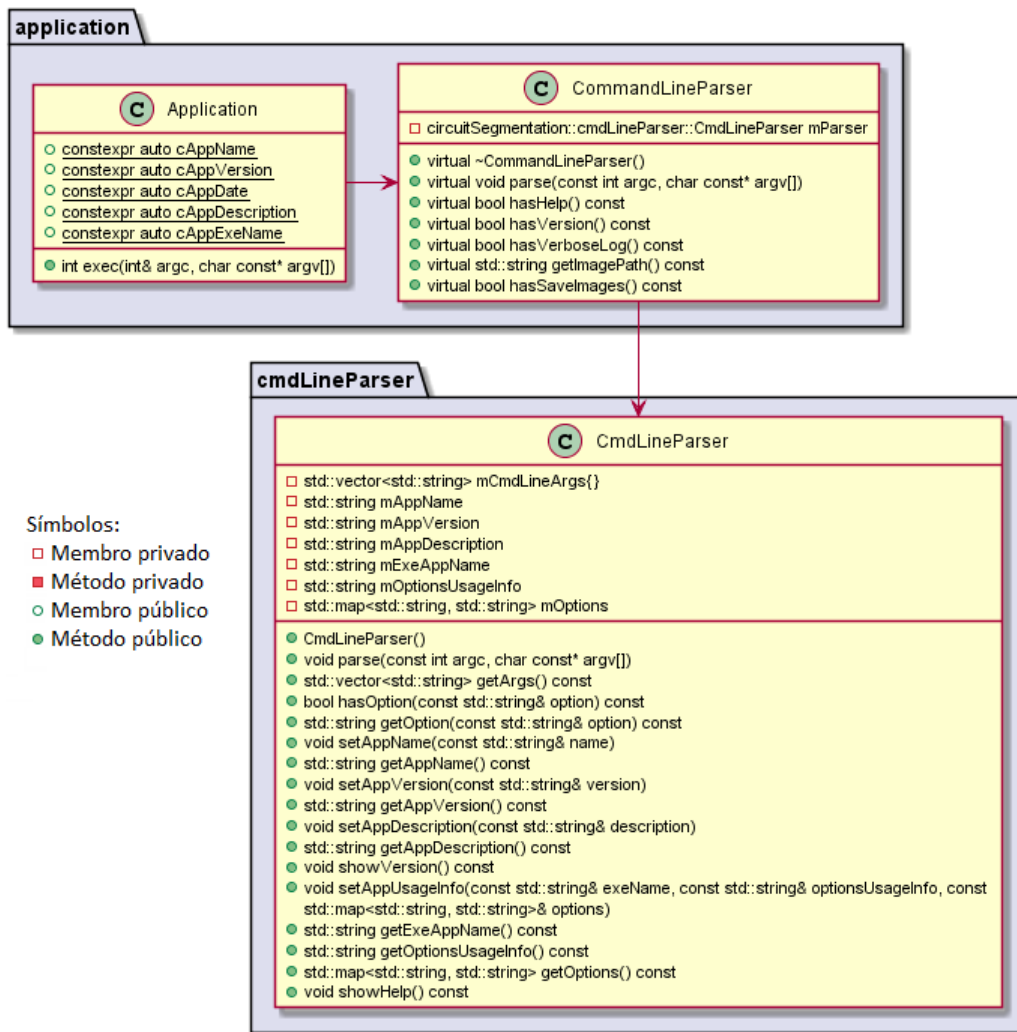


Figura 4.3: Diagrama de classes relativo ao analisador da linha de comandos.

- *None*: não há qualquer registo de mensagens (chamando qualquer método para registo não produz efeito);
- *Fatal*: regista apenas mensagens críticas;
- *Error*: regista mensagens de erro e de nível anterior (por exemplo, definindo o *logger* com este nível, as mensagens críticas são também registadas mas as de *warning* não são);
- *Warning*: regista mensagens de aviso e de níveis anteriores;
- *Info*: regista mensagens de informação e de níveis anteriores;
- *Debug*: regista mensagens de *debug* e de níveis anteriores;
- *Verbose*: regista mensagens de *verbose* e de níveis anteriores.

Neste módulo de *software*, o registo de mensagens é utilizado em diversas classes, sendo acionado pelo utilizador passando a opção “-V” ou “--verbose” pela linha de comandos para ativar o nível de *verbose* do *logger*. Caso esta opção não seja passada, o nível de registo é definido como *none*, não imprimindo qualquer mensagem para a consola. A Figura 4.4 demonstra um exemplo de registo de mensagens pelo programa quando este é ativado pelo utilizador (o Anexo D inclui o registo completo das mensagens efetuado durante o processamento da imagem de um circuito elétrico).

```
[2022-10-01 22:24:15][INFO] Starting Circuit Segmentation and Modeling: version 1.0.0  
[2022-10-01 22:24:15][INFO] Starting image processing  
[2022-10-01 22:24:15][INFO] Image file path: circuit-1.jpg  
[2022-10-01 22:24:15][INFO] Image received successfully  
[2022-10-01 22:24:15][INFO] Starting image preprocessing
```

Figura 4.4: Exemplo de *log* de mensagens do programa.

4.2.4 *Wrapper* de OpenCV

A biblioteca OpenCV, descrita na Secção 2.4.1, foi a ferramenta selecionada para acesso a algoritmos de visão computacional neste projeto, uma vez que esta fornece as funcionalidades necessárias, providencia documentação completa da sua API, além de ser uma biblioteca bastante utilizada e com um desenvolvimento ativo, possibilitando assim uma eventual atualização futura deste *software* com possíveis novas metodologias e correções de *bugs* introduzidas na ferramenta OpenCV.

Neste projeto optou-se por criar uma classe *wrapper* da biblioteca OpenCV, nomeada de *OpenCvWrapper* e pertencente ao *namespace computerVision* (encapsulamento ilustrado na Figura 4.5). Este *wrapper* consiste num módulo que encapsula a funcionalidade de OpenCV, sendo bastante útil pois providencia um nível de abstração da implementação desta biblioteca, o que permite assim simplificar a sua utilização. Além disso, a sua conceção também facilita o desenvolvimento de testes unitários, uma vez que assim é possível criar um *mock* para as funcionalidades de OpenCV, extremamente útil para testar o comportamento do *software* para diferentes resultados retornados pelos métodos desta biblioteca (testes unitários realizados ao *software* abordados na Secção 4.3).

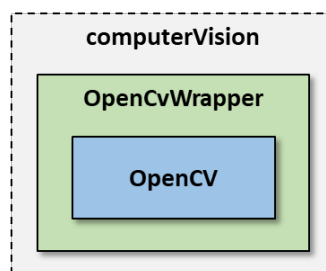


Figura 4.5: Encapsulamento da biblioteca OpenCV.

OpenCvWrapper encapsula vários tipos de dados usando *alias*es (facilitando a utilização de tipos providenciados pela biblioteca OpenCV), enumerações e métodos. A Listagem 4.2 contém alguns exemplos de encapsulamentos implementados pelo *wrapper* de OpenCV, nomeadamente do *alias* de *cv::Mat* (linha 2) e do método para leitura de ficheiro de imagem (linha 5).

```
1 // OpenCvWrapper.h
2 using ImageMat = cv::Mat;
3
4 // OpenCvWrapper.cpp
5 ImageMat OpenCvWrapper::readImage(const std::string& fileName)
6 {
7     // Read image
8     ImageMat image = cv::imread(fileName, cv::IMREAD_COLOR);
9     return image;
10 }
```

Listagem 4.2: Exemplo de encapsulamentos de OpenCV.

4.2.5 Gestor de processamento de imagem

O processamento de imagem é iniciado na segunda fase de execução do método *exec*, tal como se demonstrou na Figura 4.2. Nesta fase, é criada uma instância da classe *ImageProcManager*, que é a classe responsável deste *software* por toda a gestão do processamento da imagem de entrada do circuito. Este processo é despoletado pela chamada ao método *processImage* do *ImageProcManager*, e engloba as seguintes operações, também expostas no fluxograma da Figura 4.6:

1. Receção e leitura da imagem do circuito para processamento;
2. Pré-processamento da imagem;
3. Segmentação do circuito;
4. Geração das imagens com a ROI de cada componente e *label* segmentada;
5. Geração do mapa de segmentação.

O método *processImage* retorna um valor booleano representando o sucesso do processamento (valor “true”) ou o insucesso (valor “false”), sendo que para retornar “true”, todas as operações terão de ocorrer com sucesso. As operações mencionadas são detalhadas nas próximas secções deste capítulo.

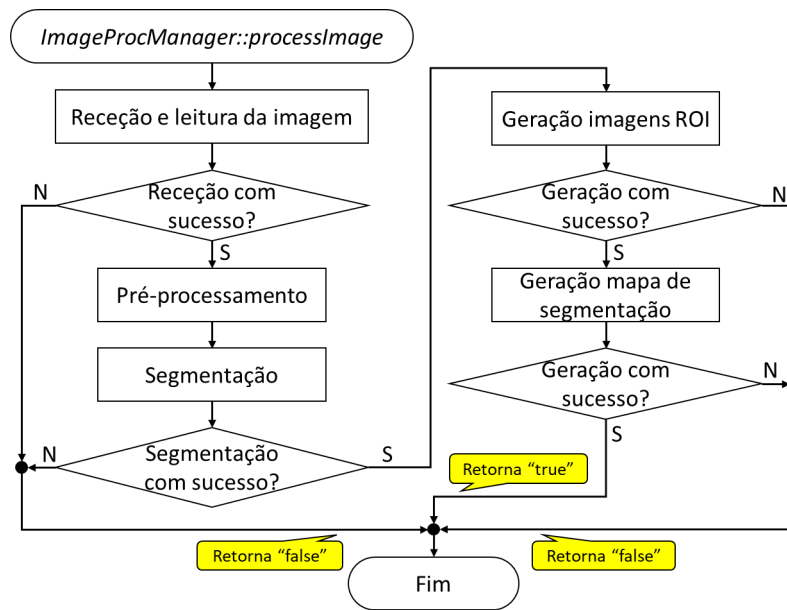


Figura 4.6: Fluxograma do processamento de imagem.

4.2.6 Receção de imagem

O processamento de imagem realizado pelo gestor (*ImageProcManager*) inicia com a receção e leitura da imagem do circuito elétrico. Os passos efetuados para a receção da imagem são os seguintes, encontrando-se também representados graficamente no diagrama de sequência da Figura 4.7:

1. Definição do caminho do ficheiro de imagem (passado pelo utilizador através da opção “-i” ou “--image”) num objeto do tipo da classe *ImageReceiver*, responsável neste *software* pela receção e leitura da imagem de entrada do circuito;
2. Receção da imagem do circuito com a chamada do método *receiveImage* de *ImageReceiver*;
3. Leitura da imagem através do *wrapper* de OpenCV, a partir do ficheiro especificado. Se a imagem não pôde ser lida (devido a ficheiro não encontrado, falta de permissões ou formato de ficheiro não suportado ou inválido), o *wrapper* retorna uma matriz de imagem vazia;
4. Verificação se a imagem lida está vazia;
5. Se a imagem estiver vazia, *ImageReceiver* informa o gestor que a imagem não foi recebida com sucesso. Caso contrário, retorna sucesso;
6. Se a imagem foi recebida com sucesso, o gestor armazena-a num membro para poder ser posteriormente passada para os objetos encarregues das próximas operações de processamento.

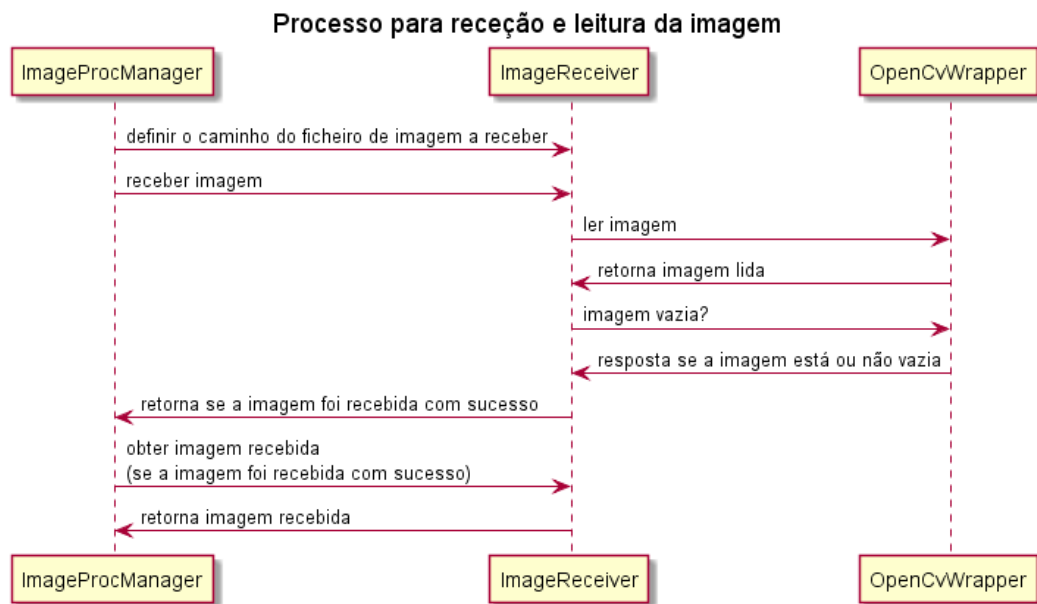


Figura 4.7: Diagrama de sequência do processo de receção e leitura da imagem.

4.2.7 Pré-processamento

Após a receção de imagem, a etapa que se segue é o pré-processamento da mesma. Esta operação é de extrema importância, visto que o sucesso da segmentação do circuito elétrico está dependente da eficiência do pré-processamento da imagem.

O pré-processamento é também uma operação iniciada pelo *ImageProcManager*, invocando o método da sua classe designado por *preprocessImage*. A Figura 4.8 ilustra o procedimento realizado para este processo.

Nas secções seguintes são aprofundadas as etapas realizadas durante o pré-processamento da imagem. Para uma explicação mais clara das operações realizadas, será utilizada a imagem de um circuito-exemplo desenhado “à mão” representada na Figura 4.9, cujas dimensões originais são de 600 x 382 (largura x altura) píxeis. Os resultados aqui expostos para este circuito podem ser acedidos no diretório de *software* do projeto, em *docs/results/circuit-2*.

Clonagem da imagem inicial

Este passo é executado pelo gestor para criar uma cópia da imagem recebida, chamando o método *cloneImage* da API do *OpenCvWrapper*. Esta cópia serve depois como argumento para o método *preprocessImage* da classe *ImagePreprocessing*, sendo posteriormente devolvida ao gestor atualizada após o pré-processamento. A classe *ImagePreprocessing* é responsável por todas as operações de pré-processamento a realizar na imagem do circuito.

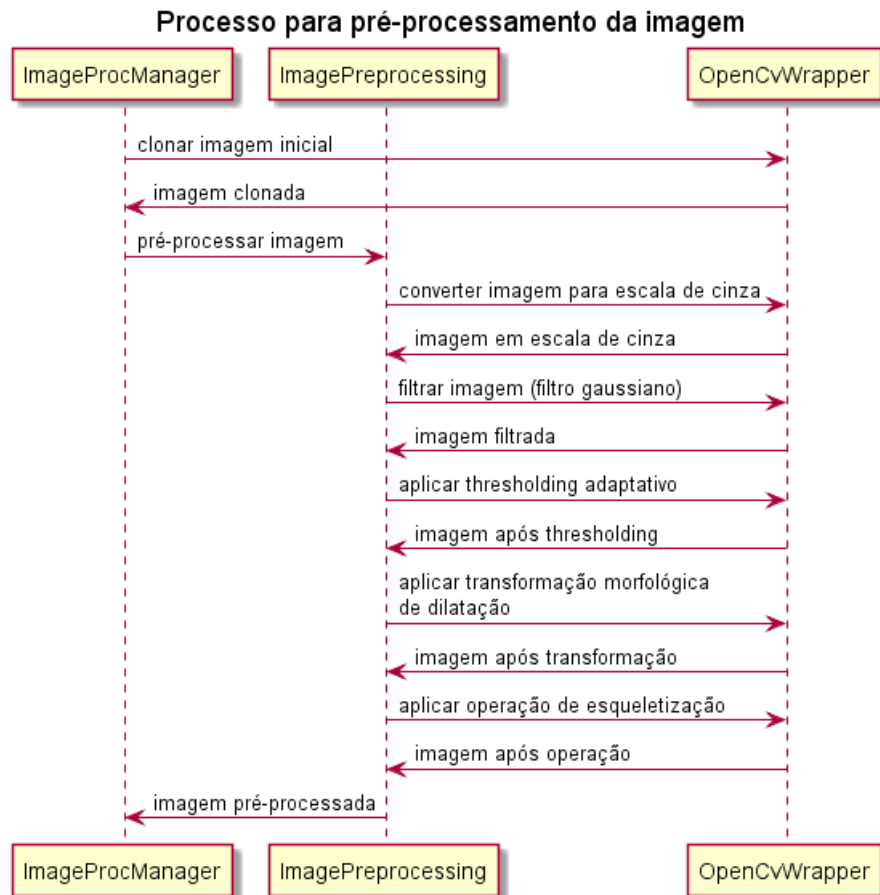


Figura 4.8: Diagrama de sequência do processo de pré-processamento da imagem.

Conversão da imagem para escala de cinza

O pré-processamento da imagem principia com a conversão da imagem para escala de cinza. Como nesta aplicação não é relevante a informação da cor da imagem, esta pode ser convertida para a escala de cinza, o que permite reduzir a complexidade dos processos seguintes.

Esta operação é executada com a chamada do método *convertImageToGray* do *OpenCvWrapper*, tendo o efeito na imagem do circuito-exemplo visualizado na Figura 4.10.

Filtragem da imagem

O passo posterior no pré-processamento da imagem é a aplicação de um filtro para redução de ruído. Entre os vários tipos de filtros existentes (enumerados na Secção 2.2.3), foi seleccionado o filtro gaussiano, uma vez que, além de reduzir o ruído na imagem, também preserva as bordas/contornos presentes, sendo assim adequado para uma aplicação de segmentação de circuitos eléctricos.

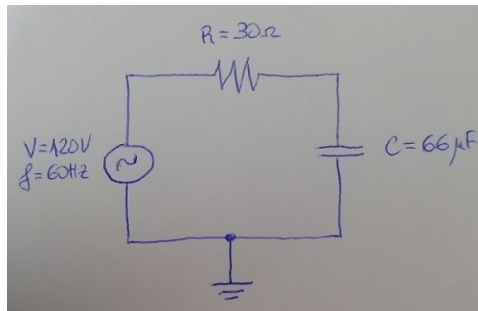


Figura 4.9: Circuito-exemplo.

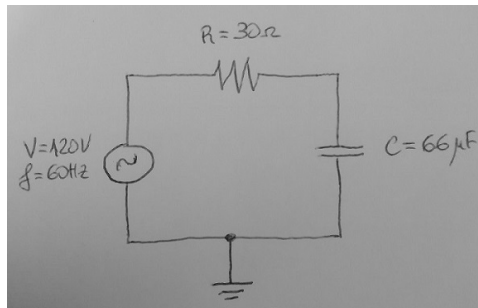


Figura 4.10: Conversão da imagem para escala de cinza durante o pré-processamento.

O método disponível no *OpenCvWrapper* para filtragem gaussiana é o *gaussian-BlurImage*, que contém um parâmetro relativo ao tamanho do *kernel* a ser aplicado nesta operação, que deve ser positivo e ímpar. A Figura 4.11 demonstra a imagem do circuito-exemplo após a sua filtragem, verificando-se o aparecimento do efeito de desfocagem devido ao filtro, mas com os contornos do circuito bem delineados.

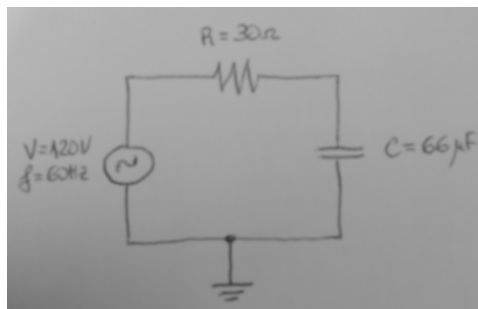


Figura 4.11: Filtragem da imagem durante o pré-processamento.

***Thresholding* adaptativo**

Após a filtragem da imagem, esta é sujeita a uma operação de *threshold* para a transformar numa imagem binária, permitindo a separação clara do circuito elétrico do *background* da imagem. A forma de *thresholding* aplicada no *software* deste projeto é o adaptativo (apresentado na Secção 2.2.4), pois esta técnica é muito útil

em imagens com condições de luminosidade variáveis, facto que pode ocorrer, por exemplo, ao tirar uma fotografia ao circuito elétrico.

O *wrapper* de OpenCV contém o método *adaptiveThresholdImage* que aplica uma operação de *threshold* adaptativo a uma imagem, cujos parâmetros são os seguintes:

- *srcImg* e *dstImg*: imagens de 8 *bits* para entrada e saída da operação, respetivamente;
- *maxValue*: valor a ser definido para os píxeis em que a condição de *thresholding* é satisfeita. Foi definido para este parâmetro o valor de 255, o que leva a que os píxeis que cumpram a condição apresentem a intensidade de cor branca;
- *adaptiveMethod*: algoritmo para *thresholding* adaptativo a usar, baseando-se na média aritmética ou gaussiana. Selecionou-se a média gaussiana para o algoritmo, apresentando assim uma menor contribuição por parte dos píxeis mais afastados da coordenada do centro de cada região da imagem, para o cálculo do *threshold* ótimo;
- *thresholdOp*: tipo de *thresholding* a utilizar, podendo ser utilizado o tipo binário (define *maxValue* para os píxeis que estão acima do *threshold*, colocando a zero os restantes) ou binário invertido (contrário ao tipo binário). A opção recaiu sobre o *thresholding* binário invertido, uma vez que, desta forma, os contornos existentes na imagem ficam destacados com cor branca sobre um fundo escuro (importante para o passo seguinte de pré-processamento);
- *blockSize*: tamanho da vizinhança de píxeis utilizado para computação do valor de *threshold* para o píxel, devendo obrigatoriamente ser um valor inteiro e ímpar (3, 5, 7, etc);
- *subConst*: constante subtraída à média ou à média ponderada. Normalmente, é um valor positivo mas poderá ser utilizado também zero ou um valor negativo.

Na Figura 4.12 está contida a imagem resultante da aplicação de *thresholding* adaptativo. É possível constatar a transformação da imagem filtrada para uma imagem binária, onde os contornos do circuito elétrico, bem como as *labels* dos seus elementos, encontram-se representados a branco.

Transformação morfológica de dilatação

A dilatação, tal como mencionado na Secção 2.2.5, é uma transformação morfológica que possibilita a amplificação das regiões brilhantes numa imagem. Após a operação de *thresholding*, o circuito elétrico e as *labels* presentes estão representadas com cor branca sobre um fundo escuro, constituindo assim as regiões brilhantes da imagem.

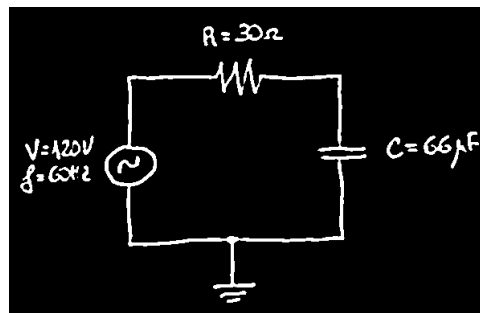


Figura 4.12: Operação de *thresholding* adaptativo durante o pré-processamento.

A aplicação da dilatação numa imagem como esta leva ao aumento da espessura dos contornos contidos na mesma, o que permite “ligar” contornos que se encontrem muito próximos. Esta é a razão pela utilização de uma transformação de dilatação durante o pré-processamento, visto que possibilita “fechar” o circuito caso existam pequenos espaços vazios nas suas conexões deixados durante o desenho do mesmo, evitando assim conexões quebradas.

Esta transformação pode ser aplicada através do método *morphologyEx* disponível na API do *OpenCvWrapper*. Este método não é específico para a dilatação, podendo ser configurado para a execução de outras transformações morfológicas, e inclui os seguintes parâmetros:

- *srcImg* e *dstImg*: imagens de entrada e saída da transformação respetivamente;
- *op*: tipo de transformação morfológica a realizar, como dilatação, erosão, *opening*, *closing*, gradiente morfológico, *top hat* e *black hat*. Evidentemente, para este parâmetro foi selecionada a dilatação nesta etapa de pré-processamento;
- *kernel*: *structuring element* a usar para análise da imagem. Este pode ser criado utilizando o método *getStructuringElement* também fornecido no *wrapper* de OpenCV, que faculta a configuração de um *kernel* com forma (retangular, em cruz ou elíptica) e dimensão desejadas. Uma vez que a finalidade da aplicação da dilatação neste caso era apenas unir contornos muito próximos, foi utilizado um *kernel* de forma retangular e com um tamanho reduzido (com valor de 3 píxeis);
- *iterations*: número de vezes que a erosão e a dilatação são aplicadas. Note-se que, por exemplo, uma operação de *opening* com duas iterações equivale a aplicar sucessivamente, erosão -> erosão -> dilatação -> dilatação, e não erosão -> dilatação -> erosão -> dilatação. Pela mesma razão referida no parâmetro anterior, optou-se apenas por uma iteração da transformação da dilatação nesta etapa.

A concretização da dilatação na imagem do circuito-exemplo resulta na Figura 4.13, na qual se observa uma pequena amplificação dos contornos da imagem.

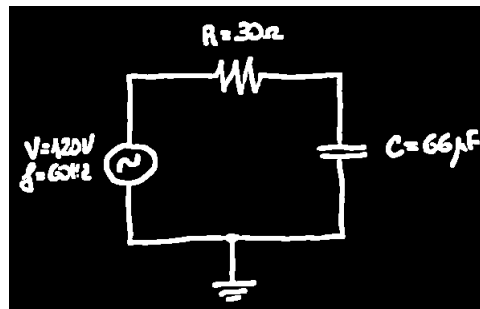


Figura 4.13: Transformação morfológica de dilatação durante o pré-processamento.

Esqueletização da imagem

A última etapa do pré-processamento da imagem engloba a realização de uma operação de esqueletização. Esta operação consiste numa transformação da imagem digital para uma imagem mais simplificada mas topologicamente equivalente, permitindo a obtenção do “esqueleto” do conteúdo da imagem através da redução de cada contorno para a densidade de um píxel. Assim, a esqueletização é especialmente útil nesta aplicação, para permitir que o posterior processo de segmentação seja independente da espessura com que os contornos do circuito elétrico foram desenhados, uma vez que estes são sempre antes atenuados para a densidade de um píxel.

A esqueletização é executada com a chamada ao método *thinning* oferecido pelo *OpenCvWrapper*, originando a imagem do circuito elétrico utilizado como exemplo exibida na Figura 4.14. O circuito apresenta agora os contornos estreitos, estando assim preparado para o processo de segmentação.

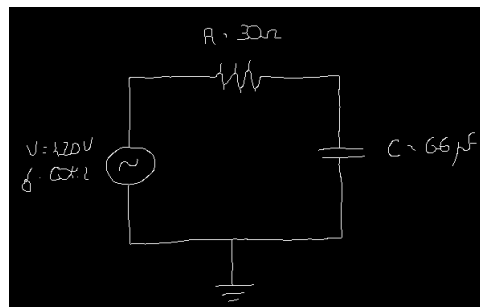


Figura 4.14: Operação de esqueletização da imagem durante o pré-processamento.

4.2.8 Segmentação

Posteriormente ao pré-processamento da imagem, o processo de segmentação do circuito elétrico pode ser realizado, tendo como objetivo a detecção e a segmentação dos seguintes elementos do circuito:

- Componentes elétricos/eletrônicos;
- Nós do circuito;
- Interligações/Conexões entre os elementos;
- *Labels* associadas a cada elemento.

A classe *ImageProcManager* solicita o início do processo de segmentação do circuito através do método *segmentImage*, pertencente à classe *ImageSegmentation*. Este processo respeita o fluxo representado na Figura 4.15, que inclui as seguintes operações (detalhadas nas próximas secções):

1. Detecção de conexões entre os elementos do circuito (caso o programa não detete a existência de interligações, é considerado que o processo de segmentação falhou);
2. Detecção de componentes presentes no circuito (caso o programa não detete a existência de componentes, é considerado que o processo de segmentação falhou);
3. Atualização da lista de conexões detetadas anteriormente, tendo em conta a detecção efetuada dos componentes (caso o programa não detete conexões após a atualização, é considerado que o processo de segmentação falhou);
4. Detecção de nós e nova atualização da lista de conexões detetadas (caso o programa não detete conexões após nova atualização, é considerado que o processo de segmentação falhou);
5. Detecção de *ports* (pontos de conexão) dos componentes previamente detetados, utilizando as conexões extraídas;
6. Atualização da lista de componentes detetados, descartando os componentes que não possuem *ports* (caso o programa não detete componentes após a atualização, é considerado que o processo de segmentação falhou);
7. Detecção de *labels* presentes no circuito (não é requerido que os elementos do circuito tenham obrigatoriamente de apresentar *labels*, portanto a segmentação é executada com sucesso mesmo que estas não sejam detetadas);

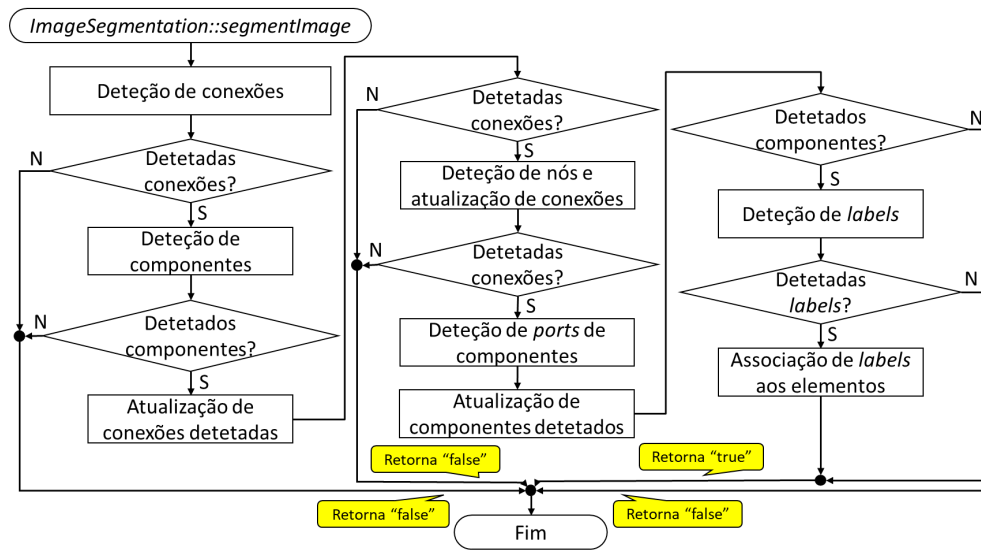


Figura 4.15: Fluxograma do processo de segmentação da imagem.

- Se forem detetadas *labels*, estas são associadas aos respetivos elementos do circuito.

O circuito-exemplo utilizado nas secções relativas ao pré-processamento da imagem, é também usado nas próximas secções do processo de segmentação do circuito.

Detecção de conexões

Na Figura 4.16 encontra-se um diagrama que demonstra a sequência de passos realizados mencionados seguidamente, para a deteção das conexões presentes no circuito:

- O processo de deteção principia com a geração de uma imagem apenas com as conexões do circuito, a partir da imagem pré-processada, sendo necessário remover todos os outros elementos presentes no circuito;
- Para isso, é aplicada uma transformação morfológica de *closing* (referenciada na Secção 2.2.5), com um *kernel* de forma retangular e de dimensão de 11 píxeis, para permitir a dilatação (seguida de erosão) dos elementos do circuito. Esta operação tem mais impacto nos componentes e *labels*, não dilatando as conexões de forma tão acentuada, facto importante para o passo seguinte;
- A transformação morfológica de *opening* (abordada na Secção 2.2.5) é também usada, cujo *kernel* apresenta um tamanho reduzido para apenas remover as conexões do circuito, restando assim na imagem somente os elementos do mesmo;

4. Esta imagem sem conexões é “interseçada” com a imagem pré-processada que inclui as conexões do circuito, levando à criação de uma imagem apenas com os componentes e as *labels*;
5. Neste ponto do processo, encontram-se apenas os elementos do circuito sem conexões na imagem de processamento, sendo necessário localizá-los através do método *findContours* disponível na API de *OpenCvWrapper* (método capaz de encontrar contornos existentes numa imagem binária). Desta forma, os contornos que representam os elementos do circuito são localizados na imagem, sendo criada, para cada contorno, uma *bounding box* (caixa envolvente);
6. Retângulos com as dimensões correspondentes às *bounding boxes* são desenhados na imagem de pré-processamento contendo todos os píxeis nulos, removendo assim todos os componentes e *labels*, restando apenas as conexões do circuito;
7. É aplicado um algoritmo similar agora para encontrar os contornos relativos às conexões, sendo verificada a dimensão de cada um, uma vez que só são considerados os contornos com um tamanho mínimo de 20 píxeis como conexões reais;
8. Cumprido o requisito, esse contorno é guardado como conexão e adicionado ao vetor de conexões detetadas pela classe *ConnectionDetection*, que armazena objetos de tipo *circuit::Connection*, sendo utilizado posteriormente durante a segmentação.

As imagens para o circuito-exemplo resultantes das operações de *closing*, de *opening* e da interseção, e da remoção dos elementos do circuito e detecção dos contornos considerados como conexões, estão respetivamente incluídas na Figura 4.17 e na Figura 4.18.

Detecção de componentes

O diagrama de sequência da Figura 4.19 contém o procedimento efetuado para a detecção de componentes presentes no circuito, constituído pelos seguintes passos:

1. As conexões detetadas na etapa anterior são removidas da imagem utilizada durante este procedimento, definindo os seus contornos com píxeis nulos, o que deixa na imagem somente os componentes e as *labels*;
2. É aplicada uma transformação morfológica de *closing*, utilizando um *kernel* de tamanho de 7 píxeis, para que os elementos na imagem sejam amplificados, sendo especialmente útil para união de linhas desconectadas que simbolizam componentes (por exemplo, símbolos do condensador ou da massa);

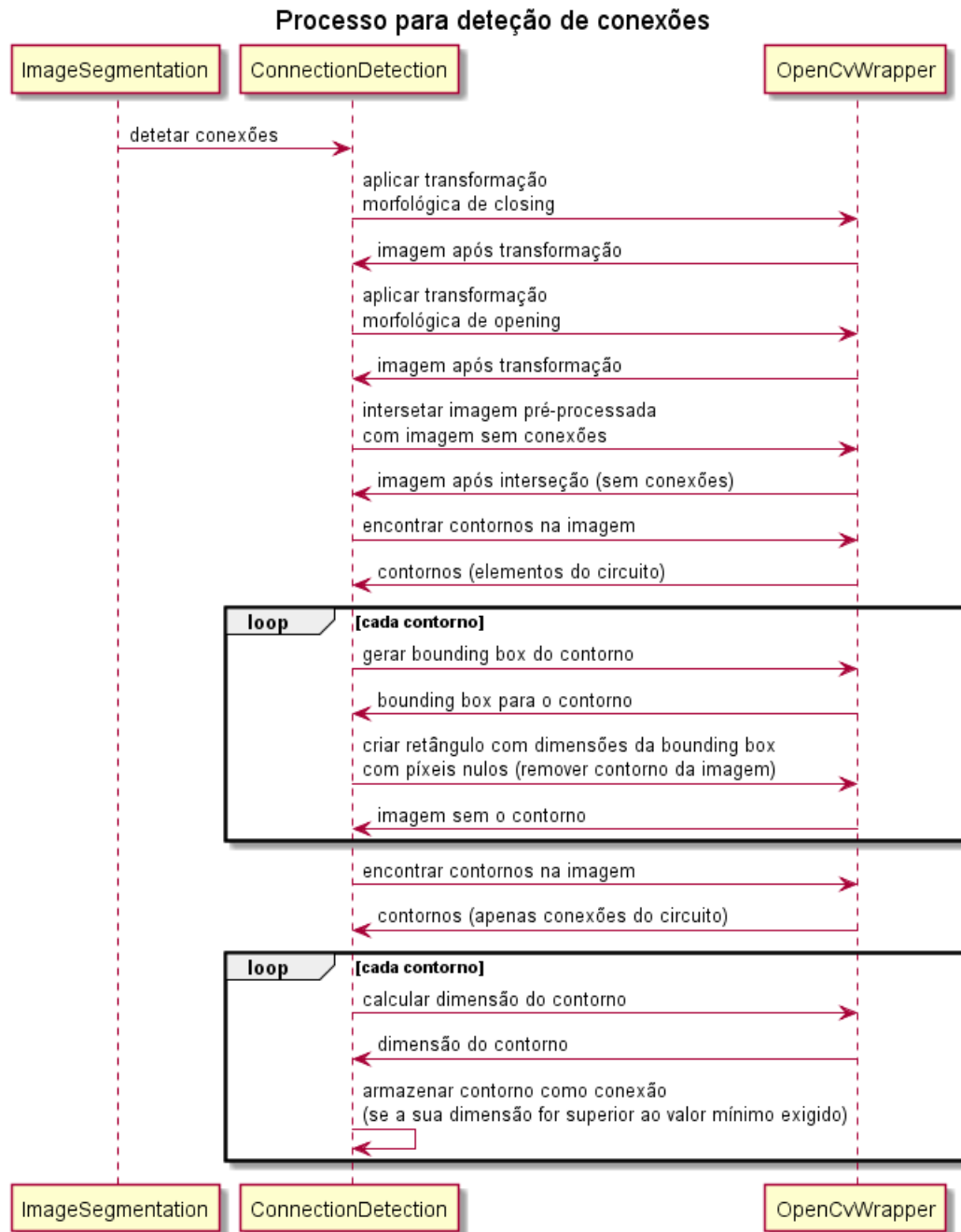


Figura 4.16: Diagrama de sequência do procedimento para detecção de conexões.

3. Procede-se à procura dos contornos existentes na imagem, sendo encontrados os contornos não só dos componentes, mas também das *labels*;
4. Para cada contorno, é gerada uma *bounding box* e é calculada a sua área, devendo o seu valor ser igual ou maior que 300 unidades de área (u.a.) para poder ser considerado um componente;

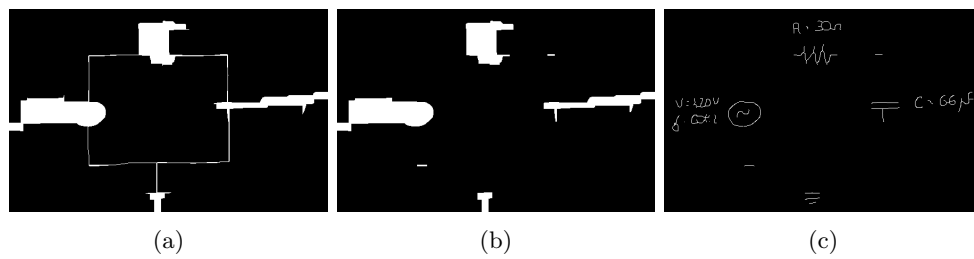


Figura 4.17: Operações durante a detecção de conexões: (a) *closing*, (b) *opening* e (c) interseção.

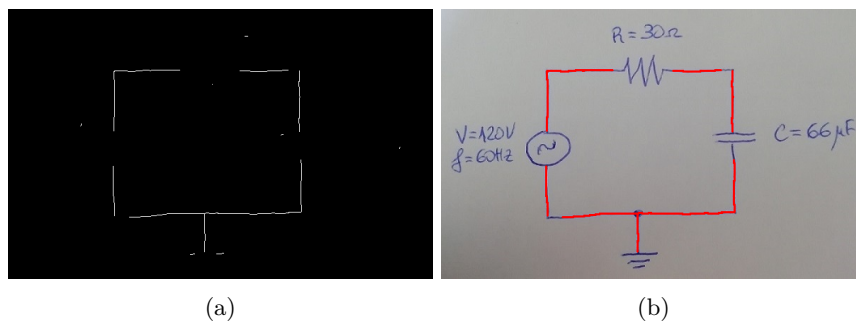


Figura 4.18: Operações durante a detecção de conexões: (a) remoção de componentes e *labels* e (b) resultado da detecção de conexões (as-sinaladas a vermelho).

5. É verificada a existência de pontos de cada conexão detetada contidos na *bounding box*. Esta verificação é necessária para diferenciar contornos representativos de componentes e de *labels*, visto que apenas os de componentes terão pontos em comum com as conexões;
6. Cada *bounding box* que passe nas condições anteriores é guardada e associada a um objeto de tipo *circuit::Component* que é aqui criado, sendo adicionado ao vetor de componentes detetados pela classe *ComponentDetection*.

Os resultados dos passos da remoção das conexões da imagem, da transformação morfológica de *closing*, e da detecção dos componentes estão expostos na Figura 4.20.

Atualização de conexões detetadas

A Figura 4.21 ilustra graficamente os seguintes passos do processo de atualização das conexões detetadas anteriormente:

1. Os componentes detetados são removidos da imagem através do desenho das suas *bounding boxes* com píxeis nulos, o que deixa a imagem neste ponto apenas com as conexões e as *labels*;

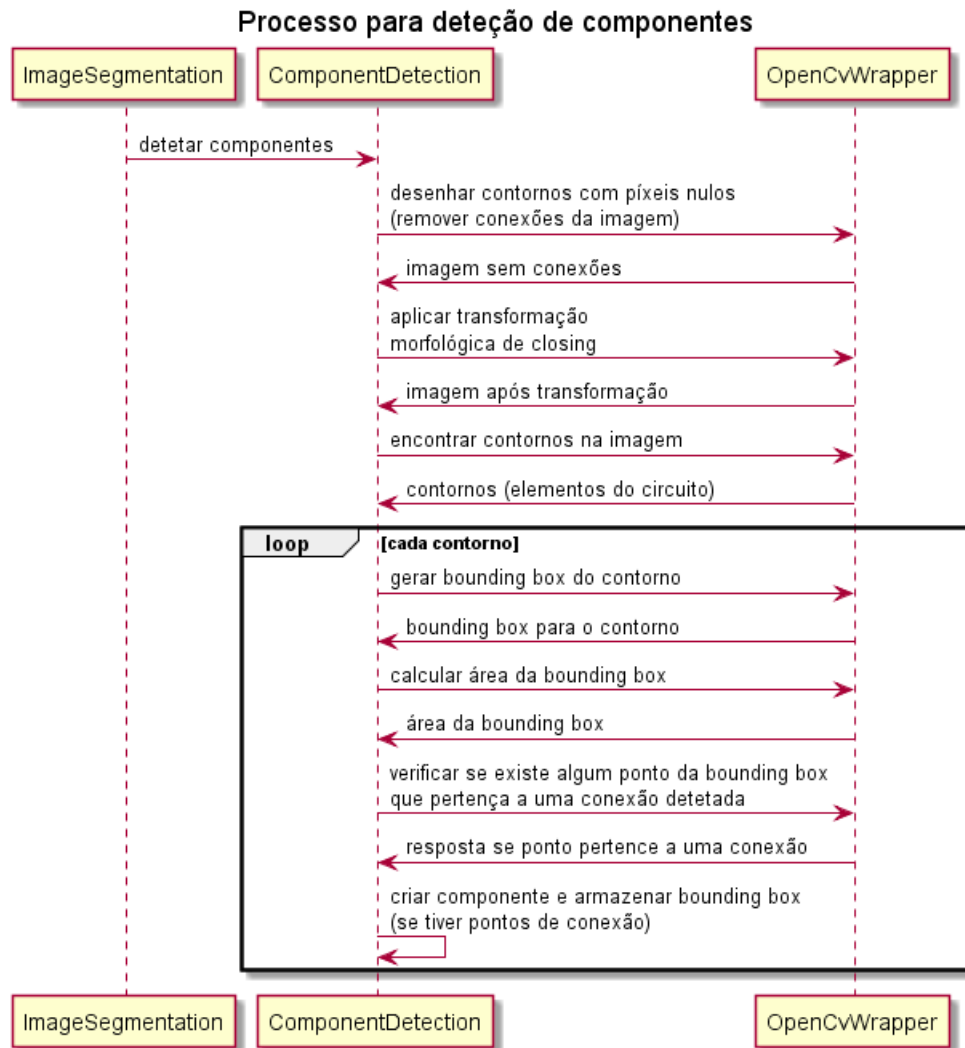


Figura 4.19: Diagrama de sequência do procedimento para deteção de componentes.

2. Procede-se a uma busca de contornos na qual é verificada a dimensão do mesmo, sendo necessário um tamanho mínimo de 20 píxeis para os contornos serem considerados como conexões.

Tal como demonstra a Figura 4.22, as conexões resultantes da atualização englobam também *labels*, no entanto esta etapa permite a deteção de conexões completas. No processo executado inicialmente para deteção de conexões podem resultar conexões quebradas, facto que ocorre no circuito-exemplo observável na Figura 4.18.

Deteção de nós e nova atualização de conexões

No início desta etapa do processo de segmentação, as conexões detetadas incluem também *labels*, logo é necessário uma nova atualização de conexões, bem como uma

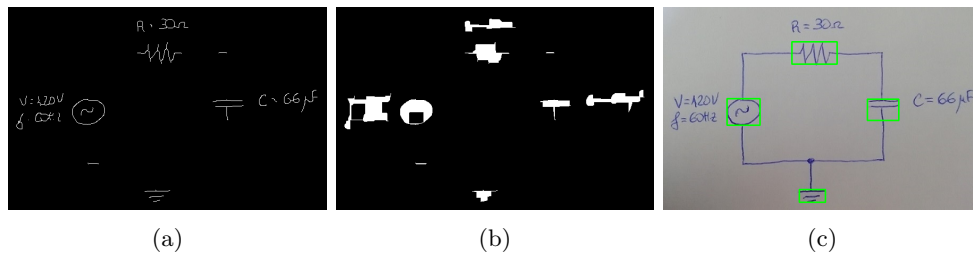


Figura 4.20: Operações durante a detecção de componentes: (a) remoção de conexões, (b) *closing* e (c) resultado da detecção de componentes (assinalados a verde).

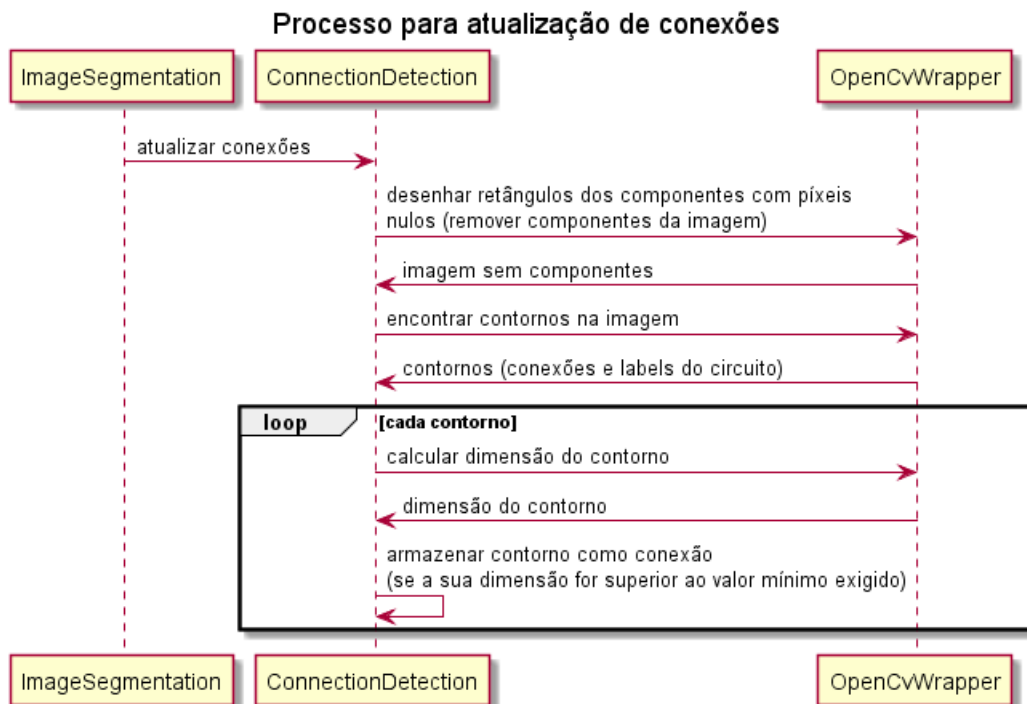


Figura 4.21: Diagrama de sequência do procedimento para atualização de conexões.

extração dos nós presentes no circuito, uma vez que a detecção de nós influencia no número de conexões existentes. A sequência de passos desta etapa de segmentação está demonstrada na Figura 4.23, nos quais se efetua o seguinte:

1. É verificado o número de pontos de interseção (N) entre cada conexão e as *bounding boxes* dos componentes anteriormente detetados:
 - Se $N = 0$: conexão detetada deve ser descartada, visto que não se encontra ligada a qualquer componente. Esta condição permite remover as *labels* do conjunto de conexões;
 - Se $N \leq 2$: conexão detetada deve ser mantida, já que interliga até dois componentes;

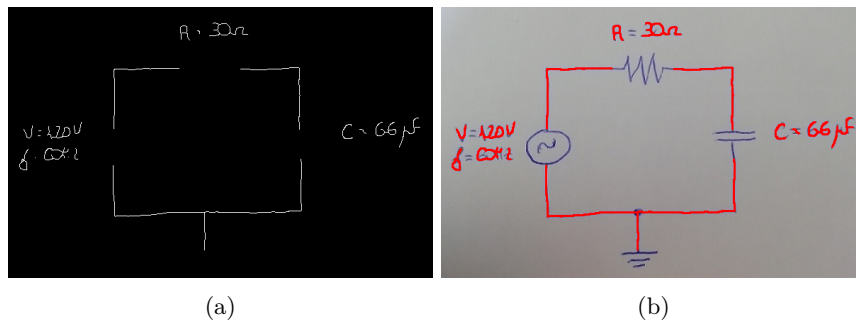


Figura 4.22: Operações durante a atualização de conexões: (a) remoção de componentes e (b) resultado da atualização de conexões (assinaladas a vermelho).

- Se $N > 2$: conexão detetada origina a criação de um nó e de N conexões, uma vez que está conectada a três ou mais componentes.

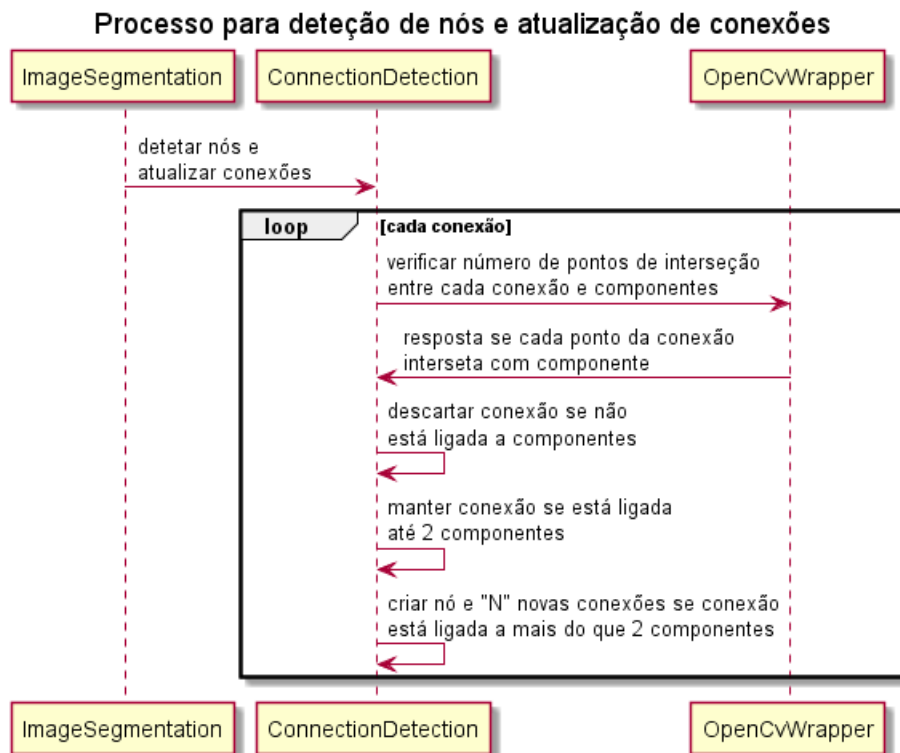


Figura 4.23: Diagrama de sequência do procedimento para deteção de nós e atualização de conexões.

Utilizando este algoritmo para a deteção de nós, não é possível localizar a posição exata de cada nó na imagem inicial, porque é a partir do contorno de uma conexão que é extraído. Assim, são recolhidos os pontos extremos do contorno da conexão, isto é, os pontos mínimos e máximos dos eixos x e y da conexão, permitindo calcular o ponto médio para cada eixo de forma a posicionar o nó no centro do contorno. A

conexão dá origem a um nó mas também a N conexões, cujas ligações vão desde o nó criado até aos respetivos componentes a que estava conectada. O resultado desta metodologia encontra-se exemplificado na Figura 4.24, onde se constata a criação de um nó e três novas conexões a partir de uma conexão ligada a três componentes. No final desta etapa, os vetores de nós e de conexões detetadas, que armazenam objetos do tipo *circuit::Node* e *circuit::Connection* respetivamente, encontram-se atualizados e podem ser usados pelas próximas etapas para segmentação do circuito.

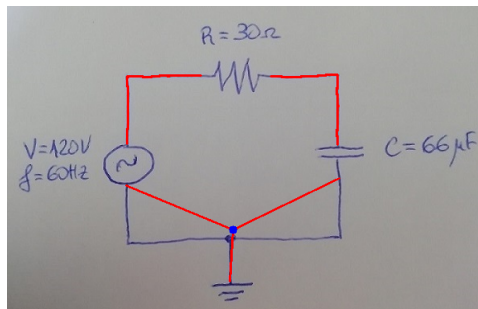


Figura 4.24: Resultado da deteção de nós e atualização de conexões (nós assinalados com pontos azuis e conexões com contornos vermelhos).

Deteção de *ports* de componentes

O termo *port* refere-se a um ponto de conexão de um componente, sendo a sua extração um requisito uma vez que este é um tipo secundário do modelo do mapa de segmentação (apresentado na Secção 3.2). A sua deteção constitui-se como o objetivo desta etapa do processo de segmentação, cujo procedimento está contido na Figura 4.25 e é composto pelos seguintes passos:

1. Para cada componente são verificados os pontos de interseção com cada conexão detetada previamente, constituindo assim os *ports* desse componente;
2. De acordo com o modelo JSON, a posição de um *port* é definida relativamente ao componente a que pertence, variando o seu valor das suas coordenadas (x , y) entre 0 e 1. Desta forma, as coordenadas dos pontos de interseção detetados não podem ser passadas diretamente para as coordenadas de posição do *port*, visto que estas são globais relativamente à origem da imagem. Assim, é necessário a realização de um cálculo da posição do *port* em relação à *bounding box* do componente a que diz respeito;
3. *Ports* detetados são associados ao respetivo componente.

De salientar que no processo de deteção de *ports* de componentes, são atualizados parâmetros das conexões detetadas, nomeadamente os *IDs* dos elementos a que estas

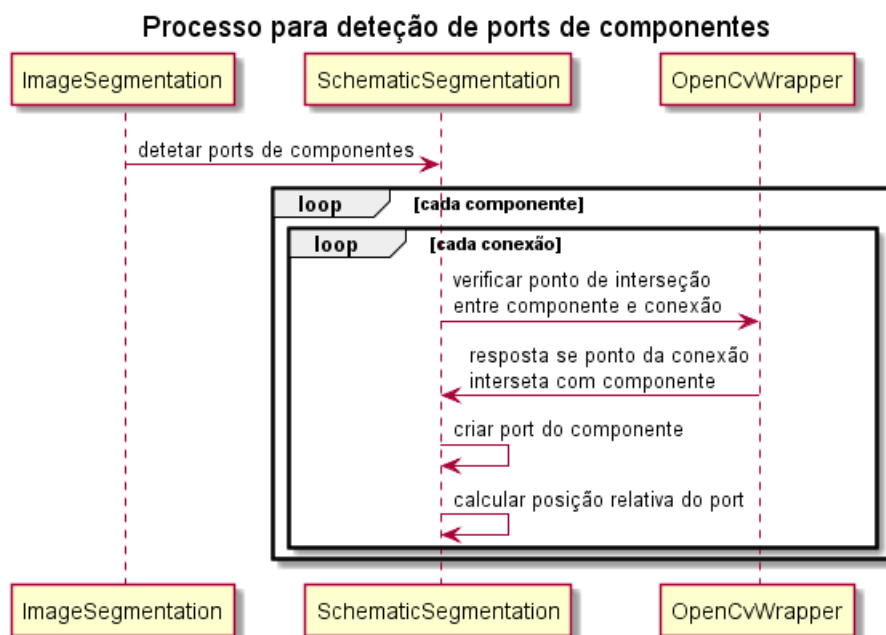


Figura 4.25: Diagrama de sequência do procedimento para detecção de *ports* de componentes.

se encontram ligadas, sendo colocado o *ID* do *port* no *start ID* da conexão se este campo ainda estiver vazio (pode já ter sido definido com *ID* de outro *port*), senão este é considerado como o *end ID* dessa conexão.

O resultado da detecção de *ports* no circuito-exemplo pode ser visualizado na Figura 4.26.

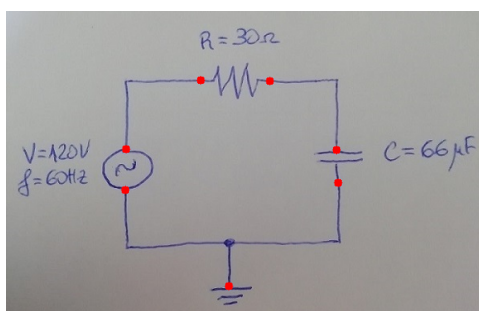


Figura 4.26: Resultado da detecção de *ports* de componentes (assinalados a vermelho).

Atualização de componentes detetados

Esta etapa do processo de segmentação apresenta um procedimento bastante simples:

1. É verificado se cada componente detetado possui, pelo menos, um *port*, o que significa que o componente se encontra ligado no mínimo a uma conexão do circuito;
2. Caso o componente não apresente pontos de conexão, é removido do correspondente vetor da classe *SchematicSegmentation* que armazena componentes detetados.

Este algoritmo permite assim a eliminação de eventuais incorretas deteções de componentes. Note-se também que durante esta etapa, os componentes que se mantiverem no vetor, são sujeitos a uma atualização das suas coordenadas de posição que são igualadas às coordenadas das respetivas *bounding boxes*.

Detecção de *labels*

O procedimento efetuado para a detecção de *labels* dos elementos do circuito está ilustrado graficamente na Figura 4.27, sendo composto pelos seguintes passos:

1. Os componentes e as conexões segmentadas do circuito são removidos da imagem utilizada neste método para processamento, desenhando as *bounding boxes* dos componentes e os contornos das conexões com píxeis nulos;
2. Uma transformação morfológica de *closing* é aplicada à imagem, usando um *kernel* retangular de dimensão de 9 píxeis, de forma a juntar todas as letras, palavras e dígitos;
3. Operação morfológica de *opening* é também utilizada neste processo, neste caso com um *kernel* de dimensão de 3 píxeis, tendo como objetivo a remoção apenas de eventuais conexões com nós do circuito que ainda permaneçam na imagem. Devido à metodologia empregue neste projeto na etapa “deteção de nós e nova atualização de conexões”, são gerados nós e conexões que não se localizam na mesma posição da imagem original, como explicado anteriormente. Assim, essas conexões “originais” que contêm nós não são removidas no passo 1 desta etapa, permanecendo ainda na imagem de processamento;
4. Neste ponto, apenas as *labels* se encontram representadas na imagem, logo é necessário proceder a uma busca de contornos para as localizar;
5. Para cada contorno, é criada uma *bounding box* e é calculada a sua respetiva área;
6. Se o valor da área for, no mínimo, de 50 u.a., um objeto do tipo *circuit::Label* é criado e armazenado num vetor com *labels* detetadas pela classe *LabelDetection*.

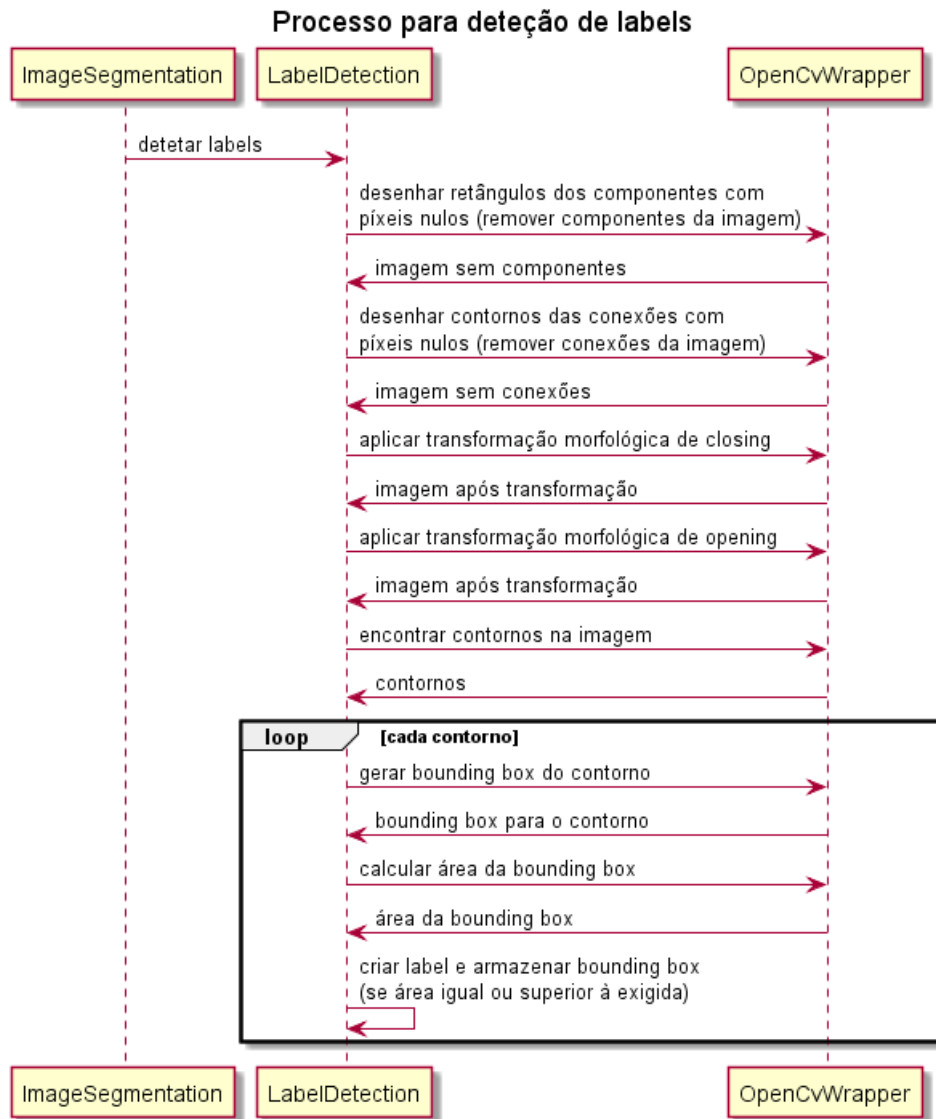


Figura 4.27: Diagrama de sequência do procedimento para detecção de *labels*.

Os resultados do procedimento realizado para o circuito-exemplo encontram-se expostos na Figura 4.28. Repare-se que na imagem referente à remoção dos elementos do circuito até aqui segmentados, encontra-se ainda a conexão que contém o nó deste circuito, daí ser necessária a aplicação da transformação morfológica de *opening* para que esta não seja considerada uma *label*.

Associação de *labels* aos elementos do circuito

O processo de associação de *labels* só ocorre se forem detetadas *labels* na etapa anterior, e tem como finalidade associar cada *label* detetada ao correspondente elemento

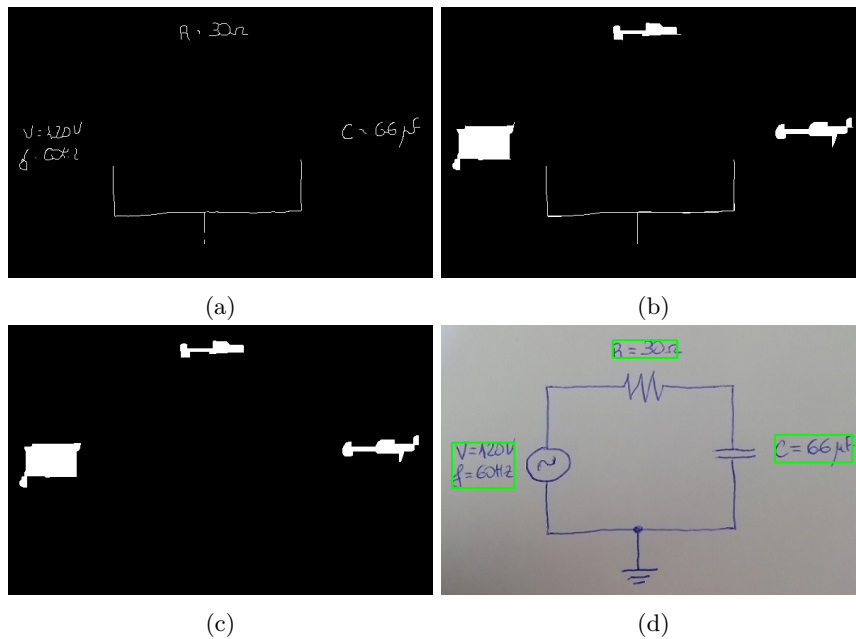


Figura 4.28: Operações durante a detecção de *labels*: (a) remoção dos elementos, (b) *closing*, (c) *opening* e (d) resultado da detecção de *labels* (assinaladas com cor verde).

do circuito, baseando-se na proximidade entre a *label* e o elemento. Para isso, são efetuados os passos descritos seguidamente, também representados na Figura 4.29:

1. Para cada conexão e nó do circuito segmentados, são geradas *bounding boxes* (para os componentes já foram anteriormente criadas, logo não é necessário nova geração para estes elementos);
2. É calculada a distância entre as *bounding boxes* de cada *label* e de cada componente, conexão e nó, de forma a se obterem as distâncias mínimas entre a *label* e cada elemento;
3. Os valores mínimos obtidos de distância são comparados, para identificação do elemento mais próximo da *label*;
4. A *label* é associada ao componente, conexão ou nó a que se encontra mais próxima, através da definição do campo *owner ID* para a *label*, e do campo *label* para o elemento associado (campos descritos na Secção 3.2).

As *bounding boxes* geradas para as conexões e nós para o circuito-exemplo estão demonstradas na Figura 4.30. Como se pode verificar, estas caixas são criadas com a largura e a altura da região ocupada pelos contornos das conexões, sendo que para os nós as caixas são amplificadas com 20 píxeis a partir dos mesmos, centrando assim o nó na respetiva caixa envolvente. Note-se que, para a conexão com o nó

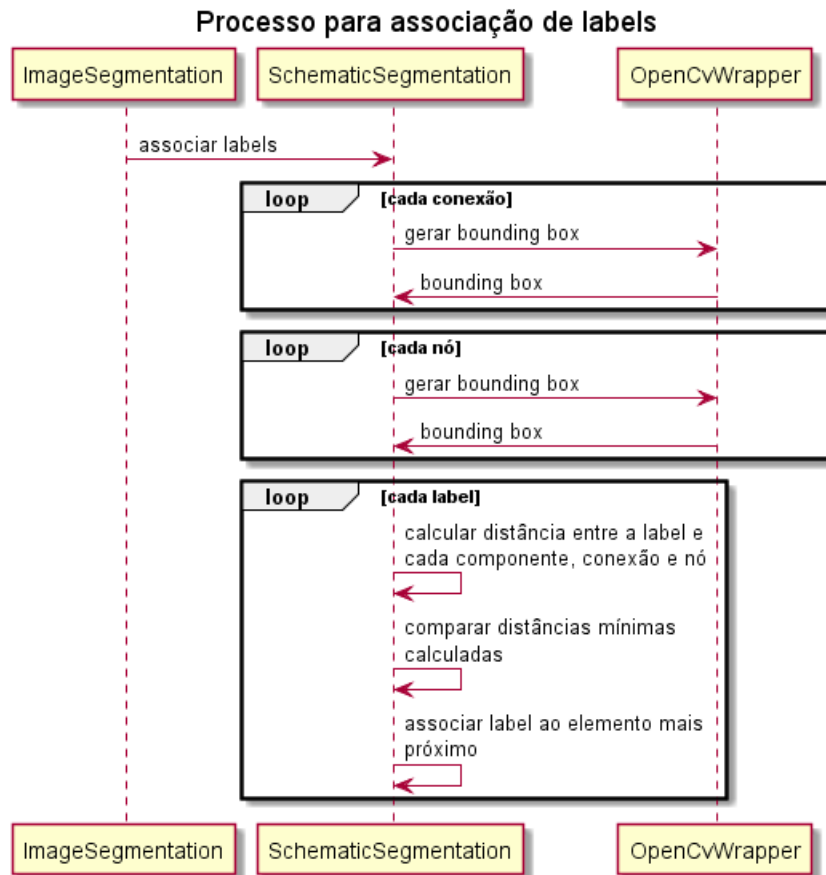


Figura 4.29: Diagrama de sequência do procedimento para associação de *labels*.

deste circuito-exemplo, as *bounding boxes* não correspondem à localização original destes elementos.

Adicionalmente, é importante mencionar que, no cálculo da distância mínima entre cada *label* e elementos do circuito, é tido em conta a posição do elemento relativamente à *label*, tal como apresenta a Figura 4.31. Quando o elemento se posiciona simultaneamente acima/abaixo e à esquerda/direita da *label*, é calculada a distância entre os pontos dos vértices das *bounding boxes*. Contudo, se o elemento se encontrar acima, abaixo, à esquerda ou à direita da *label*, a distância é obtida entre as arestas mais próximas.

Outra nota de realce é que poderão existir várias *labels* relativas ao mesmo elemento do circuito na imagem (por exemplo, a colocação do nome de um componente acima do seu símbolo e o seu valor abaixo deste). Para estes casos, o *software* é capaz de associar a cada elemento várias *labels*, gerando imagens ROI de cada *label* com identificação do *ID* do elemento a que está associado (processo apresentado na Secção 4.2.9), contudo coloca apenas o *ID* da última *label* associada ao elemento no campo respetivo do mapa de segmentação.

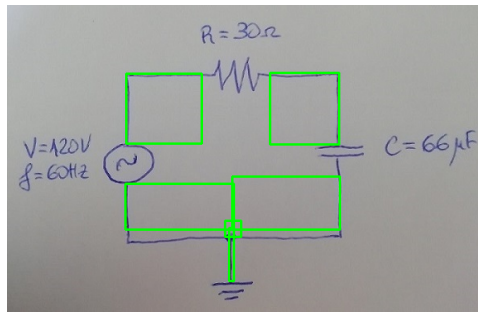


Figura 4.30: Geração de *bounding boxes* para conexões e nós durante a associação de *labels* (assinaladas a verde).

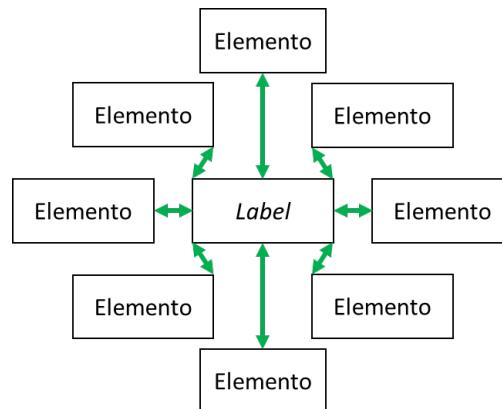


Figura 4.31: Possíveis posições do elemento relativamente a uma *label* e respectivas distâncias consideradas.

4.2.9 Regions of interest

Com o processo de segmentação concluído, estão encontrados os componentes e as *labels* presentes no circuito elétrico da imagem, seguindo-se a operação de geração de imagens com a ROI de cada um desses elementos. Estas imagens serão utilizadas por um outro módulo de *software*, que possibilitará a classificação dos componentes segmentados e o reconhecimento dos caracteres alfanuméricos contidos nas *labels*.

A geração de imagens ROI é iniciada com a invocação do método *generateImageRoi* do gestor de processamento de imagem (*ImageProcManager*), que procederá, como demonstra o fluxograma da Figura 4.32, da seguinte forma:

1. Geração de imagens com a ROI de cada componente segmentado (caso esta etapa não ocorra com sucesso, é considerado que este processo falhou);
2. Geração de imagens com a ROI de cada *label* segmentada (caso esta etapa não ocorra com sucesso, é considerado que este processo falhou).

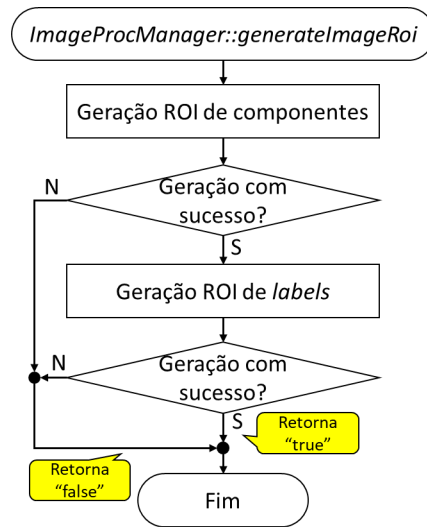


Figura 4.32: Fluxograma do processo de geração de imagens ROI.

Geração de ROI de componentes

O diagrama de sequência da Figura 4.33 ilustra graficamente o procedimento efetuado nesta etapa, sendo constituído pelos passos seguintes:

1. Para cada componente, é gerada uma nova imagem através do recorte da região da sua *bounding box* na imagem original do circuito. Esta operação recorre ao método *cropImage* de *OpenCvWrapper*, que retorna se o recorte foi realizado com sucesso (por exemplo, pode falhar se a ROI possui coordenadas que a localizam para além da dimensão da imagem a recortar);
2. A imagem ROI de cada componente é guardada num ficheiro.

Os ficheiros gerados das imagens ROI dos componentes apresentam o seguinte formato: “roi_component_<component-ID>.png”, onde “component-ID” corresponde ao *ID* do componente a que a ROI diz respeito. De referir também que estas imagens são criadas no diretório atual onde o programa se encontra a ser executado.

Para o circuito-exemplo, as ROI dos componentes geradas encontram-se representadas na Figura 4.34, estando também disponíveis no diretório do *software* do projeto em *docs/results/circuit-2/assets*.

Geração de ROI de labels

O procedimento realizado nesta etapa é o descrito seguidamente, ilustrado pelo diagrama da Figura 4.35:

1. Para cada elemento do circuito (componente, conexão e nó), é gerada uma ROI de cada *label* associada, recortando a correspondente área da sua *bounding box* na imagem original do circuito;

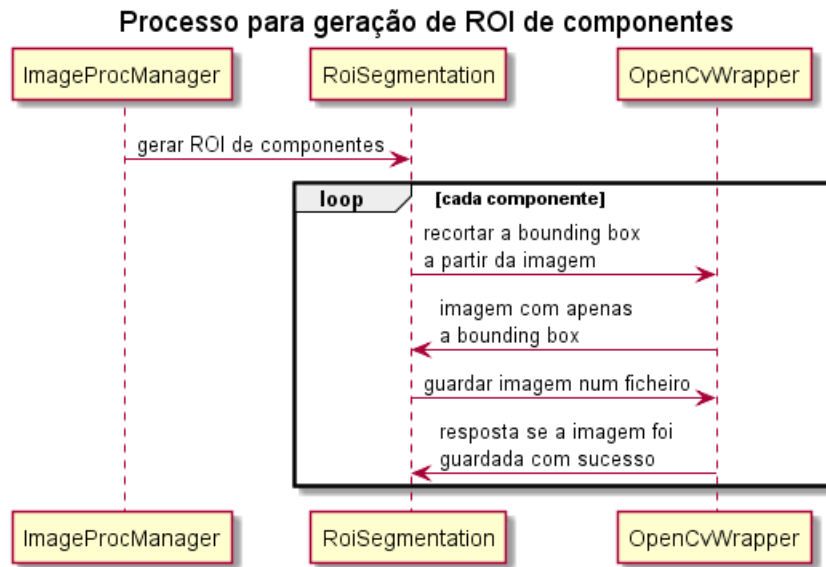


Figura 4.33: Diagrama de sequência do procedimento para geração de ROI de componentes.

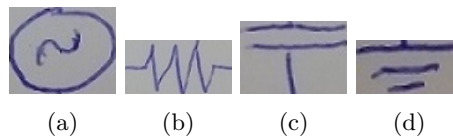


Figura 4.34: Imagens ROI de componentes: (a) fonte de tensão, (b) resistência, (c) condensador e (d) massa.

2. Posteriormente, são gerados ficheiros das imagens ROI de cada *label* associada ao respetivo elemento do circuito.

Igualmente ao efetuado com as ROI dos componentes, as imagens das *labels* são também guardadas no diretório no qual o programa foi arrancado. O formato respeitado no nome destes ficheiros é “roi_label__associated-element-id>__<n>.png”, em que:

- “associated-element-id” corresponde ao *ID* do elemento associado à *label* (note-se que este *ID* não é o *ID* da *label*);
- “n” equivale ao número da *label* associada. Visto que cada elemento do circuito pode ter mais do que uma *label* associada a si, este número apenas serve para numerar as várias *labels*. Por exemplo, se um elemento possuir duas *labels*, são criados dois ficheiros das respetivas ROI com o mesmo *ID* do elemento associado, mas no primeiro ficheiro tem “n” a “1” e no segundo a “2”.

A Figura 4.36 inclui as imagens ROI das *labels* geradas para o circuito-exemplo (disponíveis no diretório do *software* do projeto em *docs/results/circuit-2/assets*), para cada elemento a si associado.

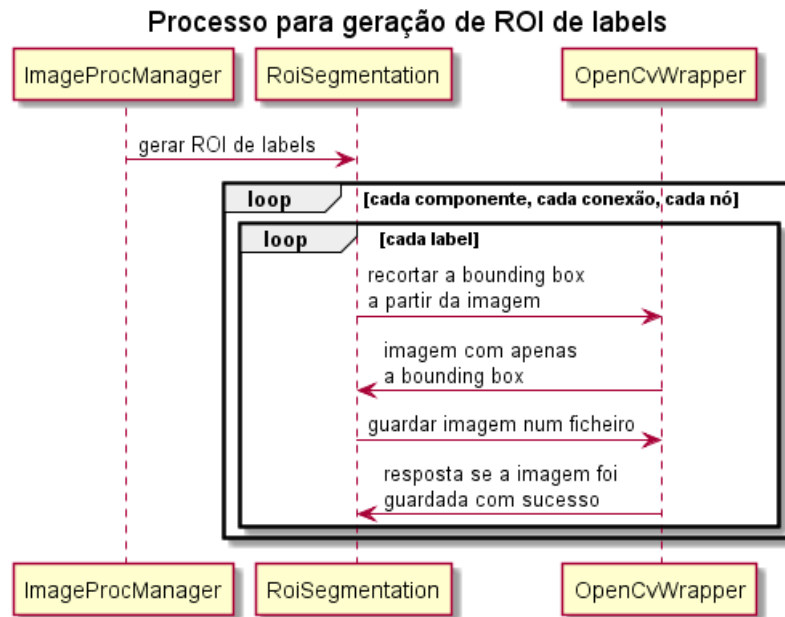


Figura 4.35: Diagrama de sequência do procedimento para geração de ROI de *labels*.

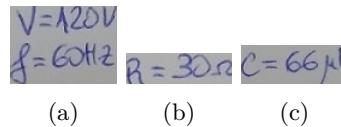


Figura 4.36: Imagens ROI de *labels* associadas: (a) à fonte de tensão, (b) à resistência, e (c) ao condensador.

4.2.10 Mapa de segmentação

O mapa de segmentação consiste num mapa/matriz que agrupa os elementos do circuito resultantes do processo de segmentação, bem como as suas interligações, estruturado segundo o modelo JSON apresentado na Secção 3.2. Assim, o circuito elétrico segmentado fica modelado neste mapa, todavia o modelo não está ainda completo no final deste processo, visto que falta a classificação de cada componente e a decodificação da informação contida em cada *label*. Desta forma, este *software* gera um ficheiro JSON com o mapa incompleto, que será posteriormente preenchido com os dados em falta referidos por um outro módulo de *software*.

O processo de segmentação realizado previamente prepara os objetos necessários a serem incluídos no mapa de segmentação, nomeadamente objetos dos tipos *Component*, *Connection* e *Node*. De referir que, na criação destes objetos durante a segmentação, são gerados *IDs* únicos para cada um deles, para que seja facilmente identificado no esquema elétrico. A criação do *ID* é feita automaticamente no construtor de cada objeto, tal como exemplifica a Listagem 4.3 para um objeto

do tipo *Component*, que contém também já a definição do *owner ID* da *label* associada a si. Como se pode constatar, o *ID* é gerado recorrendo à função *generateId* que está contida no ficheiro *Id.h* do *software*. Esta função, por sua vez, invoca o método *generateStringUuid* pertencente à classe *UuidGen* (*namespace common*), classe que permite a geração de UUID e que funciona como um *wrapper* da biblioteca *stduuid* [57] (biblioteca de geração de UUID). Estas relações estão espelhadas no diagrama da Figura 4.37, no qual se verifica também que o tipo *Position* não se encontra ligada ao bloco *Id*, pois este é um tipo básico do modelo que não requer nenhum *ID* para identificação.

```

1 Component::Component()
2   : mId{generateId()}
3 {
4   // Set label owner ID
5   mLabel.mOwnerId = mId;
6 }

```

Listagem 4.3: Construtor da *struct Component*.

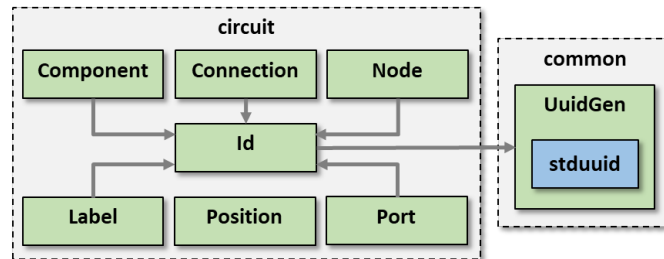


Figura 4.37: Relação entre classes para geração de UUID.

A geração do mapa de segmentação é desencadeada pelo gestor de processamento de imagem (*ImageProcManager*), que realiza as seguintes etapas descritas também graficamente no diagrama de sequência da Figura 4.38:

1. Os vetores dos elementos do circuito segmentados são passados para a classe *SegmentationMap*, onde são lidos os campos das estruturas de cada componente, conexão e nó, sendo estes adicionados ao membro *mJsonMap* desta classe. Este membro guarda o mapa de segmentação preenchido e é um objeto de um tipo de dados JSON fornecido pela biblioteca “JSON for Modern C++” [58];
2. Após a criação e preenchimento do mapa com sucesso (pode falhar se não for preenchido com os tipos de dados e estruturas corretas), este pode ser guardado

num ficheiro de formato JSON. Assim, é criado e aberto pelo *software* um ficheiro deste tipo, permitindo a escrita para este do mapa de segmentação estabelecido anteriormente (objeto *mJsonMap*).

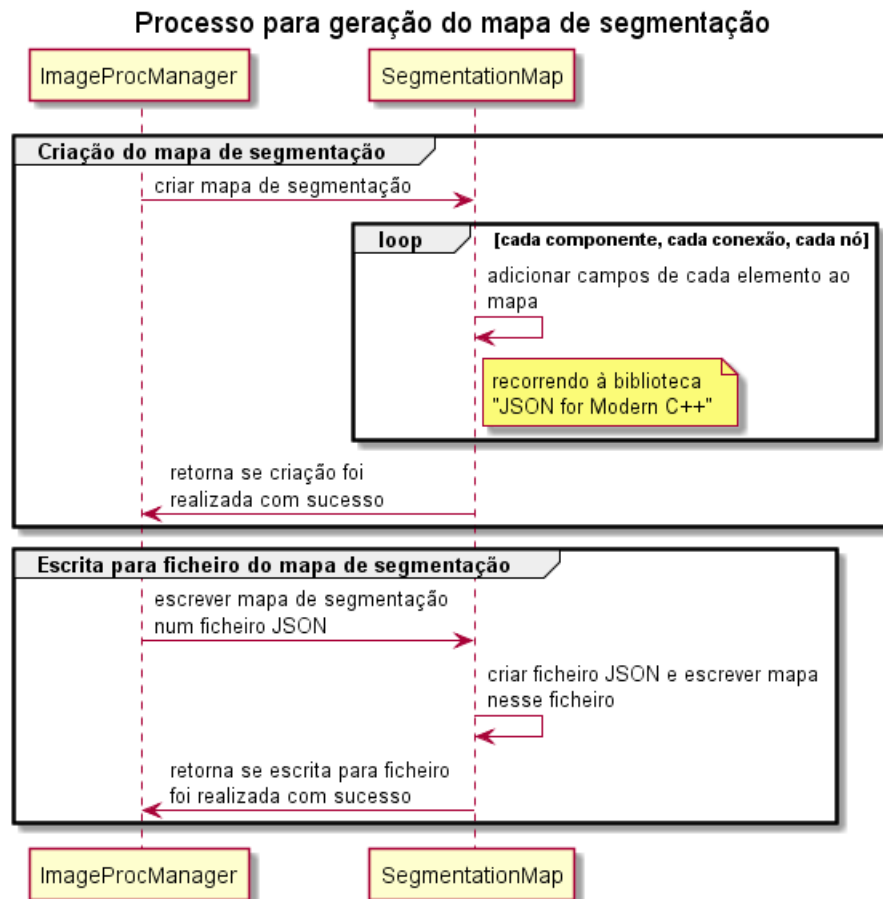


Figura 4.38: Diagrama de sequência do procedimento para geração do mapa de segmentação.

O mapa de segmentação é guardado num ficheiro no diretório onde o programa se encontra em execução, com o nome “segmentation_map.json”.

Um exemplo de mapa de segmentação gerado por este programa está contido no Anexo A, referente ao circuito-exemplo utilizado ao longo desta dissertação. Como se pode constatar, a estrutura do mapa respeita o modelo JSON, baseada sobretudo em três *arrays* correspondentes aos tipos principais do modelo, tal como descreve a Listagem 4.4. Além disso, verifica-se que apresenta quatro componentes, um nó, e cinco conexões, que correspondem respetivamente ao número desses elementos segmentados pelo programa para o circuito-exemplo. Este mapa está também disponível no diretório do projeto em *docs/results/circuit-2/assets*.

```
1 {
2   "components": [
3     ...
4   ],
5   "nodes": [
6     ...
7   ],
8   "connections": [
9     ...
10  ]
11 }
```

Listagem 4.4: Estrutura base do mapa de segmentação.

4.3 Testes unitários

Durante o desenvolvimento do módulo de *software*, foram construídos diversos testes unitários, que permitem o teste de unidades individuais de código para verificação do seu correto funcionamento. São testes automatizados que possibilitam o isolamento de uma unidade de código (frequentemente esta equivale a uma classe ou a um método individual em programação orientada a objetos, categoria em que se insere o *software* deste projeto), fazendo uso de objetos *mock* para providenciar esse isolamento. Um objeto *mock* implementa a mesma interface que um objeto real, no entanto permite ao programador definir o seu comportamento e verificar como este foi utilizado (quais os métodos deste *mock* que devem ser chamados, com que ordem, quantas vezes, com quais argumentos, o que deve retornar, etc).

Os testes unitários implementados neste projeto estão localizados no diretório do *software* em *tests/unit*, testes que utilizam os *mocks* definidos em *tests/mocks*. Para facilitar o seu desenvolvimento, este projeto utiliza a biblioteca GoogleTest [59], que proporciona funcionalidades para construção tanto de testes unitários como de *mocks*.

Os comandos para execução dos testes unitários estão descritos no Anexo B, cujo resultado encontra-se na Figura 4.39. É possível vislumbrar que o projeto apresenta 184 testes unitários, estando 100% dos testes a passarem.

A cobertura de código, vulgarmente designada em inglês como *code coverage*, é uma métrica que auxilia na percepção de quanto do código-fonte do *software* está testado. A aplicação LCOV é a ferramenta usada neste projeto para obtenção desta métrica, que é um *front-end* gráfico da ferramenta de teste de cobertura GCOV de *GNU Compiler Collection* (GCC) [60]. O relatório gerado por esta ferramenta está contido na Figura 4.40, cuja análise demonstra que os testes unitários implementados

```

Start 182: SegmentationUtilsTest.distanceRectanglesWhenIntersection
182/184 Test #182: SegmentationUtilsTest.distanceRectanglesWhenIntersection ..... Passed
0.19 sec
Start 183: SegmentationUtilsTest.distanceRectanglesSameRectangle
183/184 Test #183: SegmentationUtilsTest.distanceRectanglesSameRectangle ..... Passed
0.20 sec
Start 184: SegmentationUtilsTest.roundsDouble
184/184 Test #184: SegmentationUtilsTest.roundsDouble ..... Passed
0.19 sec

100% tests passed, 0 tests failed out of 184

Total Test time (real) = 33.97 sec

```

Figura 4.39: Resultado da execução dos testes unitários.

cobrem 91.2% das linhas de código desenvolvido. De notar que o código da função *main* e da classe *Application* estão excluídas desta análise e não influenciam na percentagem obtida, uma vez que a lógica envolvida neste código é verificada com testes manuais ao funcionamento do programa completo.

LCOV - code coverage report

Current view: top level		Hit	Total	Coverage
Test: coverage_filtered.info	Lines:	1175	1288	91.2 %
Date: 2022-08-24 11:17:11	Functions:	161	179	89.9 %

Directory	Line Coverage ↕	Functions ↕
application	100.0 % 37 / 37	100.0 % 6 / 6
circuit	83.3 % 30 / 36	100.0 % 11 / 11
cmdLineParser	100.0 % 52 / 52	100.0 % 17 / 17
common	100.0 % 7 / 7	100.0 % 4 / 4
computerVision	97.0 % 129 / 133	100.0 % 27 / 27
imageProcessing	95.7 % 222 / 232	78.0 % 32 / 41
logging	100.0 % 44 / 44	92.3 % 12 / 13
schematicSegmentation	87.6 % 654 / 747	86.7 % 52 / 60

Generated by: [LCOV version 1.14](#)

Figura 4.40: Relatório gerado pelo LCOV.

Capítulo 5

Casos de estudo

Este capítulo contém alguns exemplos de circuitos elétricos utilizados para avaliar o desempenho do módulo de *software* implementado neste projeto. Estes casos de estudo apresentam diferentes níveis de complexidade, quer em termos de qualidade da imagem do circuito, quer também em número de componentes, conexões e nós presentes no mesmo. Além disso, são ainda mencionadas as limitações deste *software* usando também casos reais de circuitos.

5.1 Circuito elétrico desenhado “à mão”

O primeiro caso de estudo consiste num circuito elétrico simples, desenhado “à mão”, com uma pequena quantidade de componentes e de conexões e não contendo qualquer nó, encontrando-se representado na Figura 5.1, cujas dimensões originais são de 800 x 587 (largura x altura) píxeis.

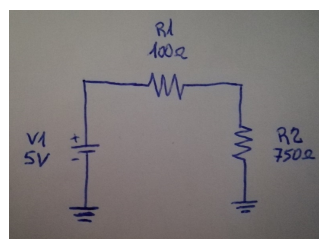


Figura 5.1: Circuito elétrico do caso de estudo 1.

As imagens resultantes do processamento deste caso de estudo, as ROI dos componentes e das *labels* criadas, bem como o mapa de segmentação gerado para este circuito, estão disponíveis em *docs/results/circuit-1* no diretório do *software* do projeto.

A imagem deste circuito apresenta condições de luminosidade variáveis em diferentes regiões. Nestas condições, o pré-processamento da imagem assume um papel ainda mais fulcral para que o circuito seja posteriormente segmentado com eficiência. A Figura 5.2 demonstra as diferentes etapas do pré-processamento realizado para a imagem deste caso de estudo, nas quais se pode constatar que o *software* conseguiu realçar com sucesso o circuito e as suas *labels*.

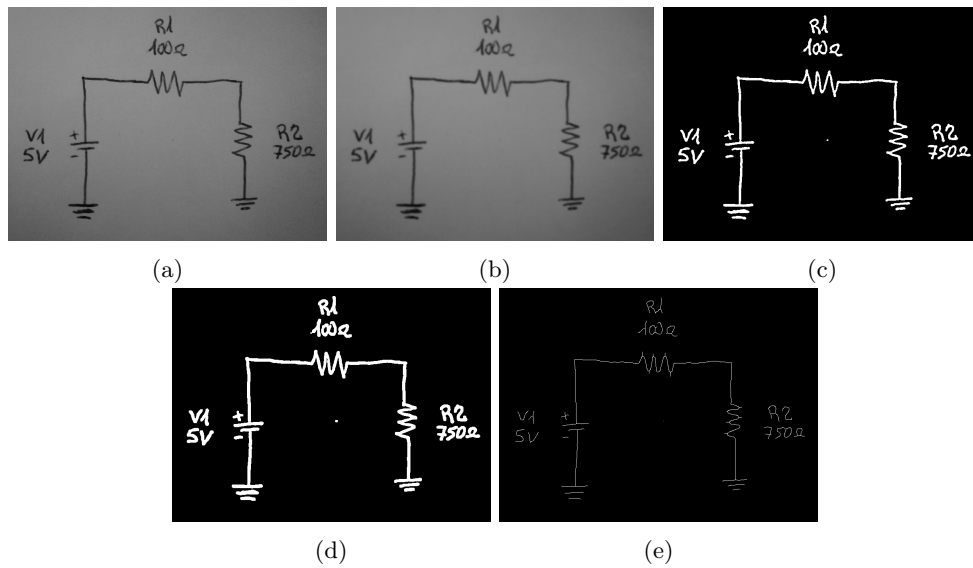


Figura 5.2: Pré-processamento do caso de estudo 1: (a) escala de cinza, (b) filtragem, (c) *thresholding*, (d) dilatação e (e) esqueletização.

Após o pré-processamento, o processo de segmentação é despoletado, iniciando a deteção das conexões do circuito. Os resultados nesta etapa estão ilustrados nas Figuras 5.3 e 5.4, em que se destaca o facto das conexões detetadas que interligam V1 a R1 e R1 a R2, estarem quebradas em segmentos que restaram na imagem após a operação de *opening*, visualizável na Figura 5.3b. Estas conexões serão corrigidas num passo posterior durante a segmentação.

A próxima etapa é a deteção de componentes, cuja segmentação está apresentada na Figura 5.5, onde se conclui que os componentes presentes neste circuito são localizados com sucesso.

Com os componentes segmentados, pode proceder-se a uma atualização do conjunto de conexões detetadas, através da remoção da imagem desses mesmos componentes. Porém, as conexões agora atualizadas englobam também as *labels* dos elementos do circuito, tal como ilustra a Figura 5.6. Após o processo de deteção de

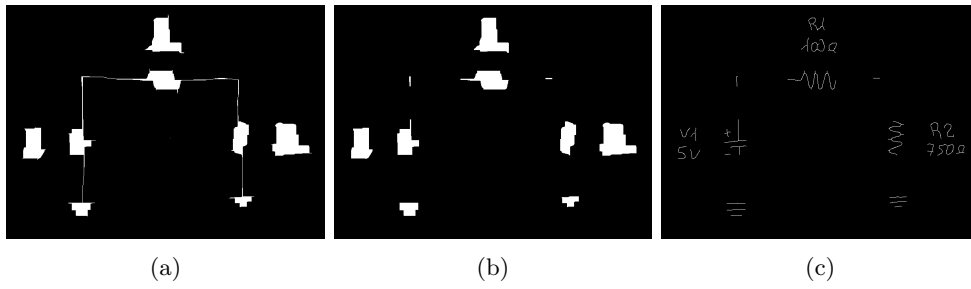


Figura 5.3: Detecção de conexões do caso de estudo 1: (a) *closing*, (b) *opening* e (c) interseção.

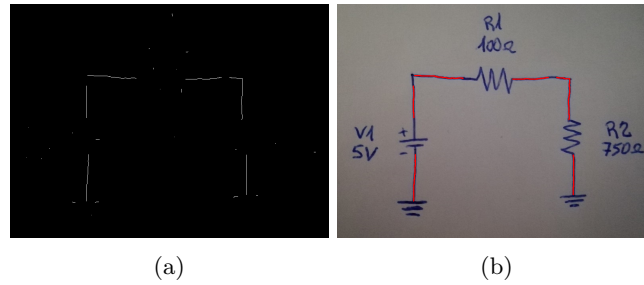


Figura 5.4: Detecção de conexões do caso de estudo 1: (a) remoção de componentes e de *labels*, e (b) conexões detetadas.

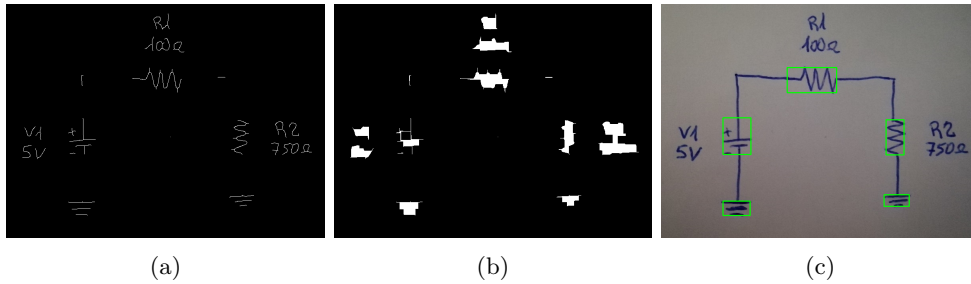


Figura 5.5: Detecção de componentes do caso de estudo 1: (a) remoção de conexões, (b) *closing* e (c) componentes detetados.

nós, o grupo destas conexões é novamente sujeito a verificação, no qual se descartam as *labels* como conexões, chegando-se finalmente à sua correta segmentação, contida na Figura 5.7a (note-se a ausência de pontos representativos de nós nesta imagem, pois este circuito não possui nenhuma conexão entre mais do que dois componentes). Utilizando as conexões extraídas, os *ports* de cada componente poderão ser detetados, através da interseção dos contornos de cada conexão com a respetiva *bounding box* do componente, gerando para este circuito o resultado ilustrado na Figura 5.7b.

Neste ponto do processo de segmentação, resta apenas a extração das *labels* presentes na imagem do circuito. Sabendo a localização dos componentes e das conexões, estes elementos podem ser removidos da imagem, o que permite desta forma detetar mais facilmente as *labels*. A Figura 5.9 apresenta o resultado da sua segmentação para este caso de estudo.

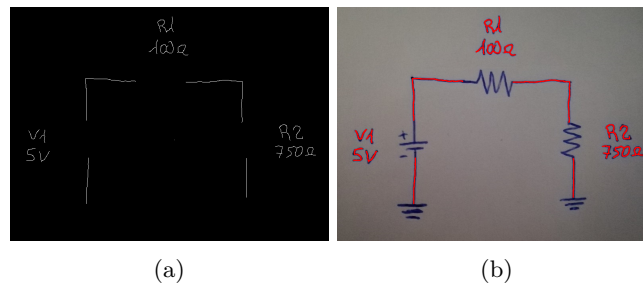


Figura 5.6: Atualização de conexões detetadas do caso de estudo 1: (a) remoção de componentes e (b) conexões atualizadas.

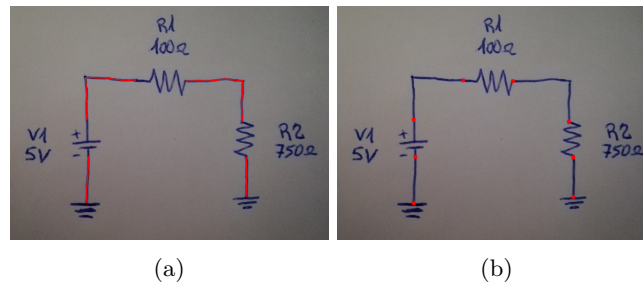


Figura 5.7: Caso de estudo 1: (a) nós detetados e conexões atualizadas, e (b) *ports* de componentes detetados.

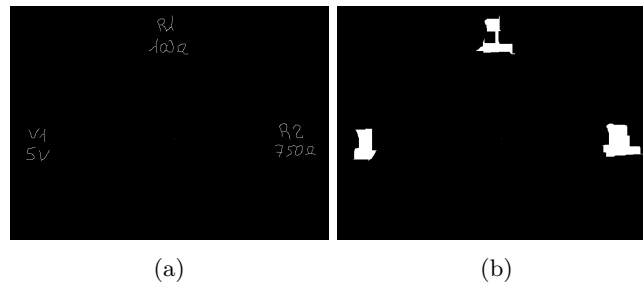


Figura 5.8: Detecção de *labels* do caso de estudo 1: (a) remoção dos elementos e (b) *closing*.

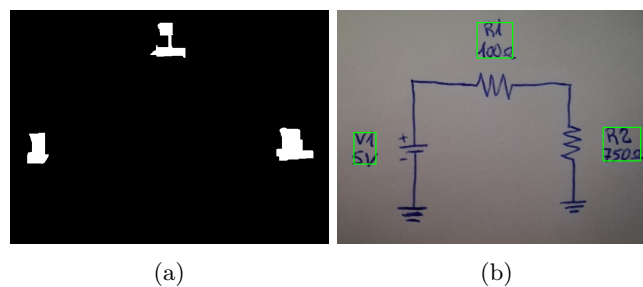


Figura 5.9: Detecção de *labels* do caso de estudo 1: (a) *opening* e (b) *labels* detetadas.

Com todos os elementos do circuito segmentados, o passo seguinte é a geração de imagens com as ROI relativas aos componentes e às *labels* detetadas. As ROI criadas deste circuito encontram-se contidas na Figura 5.10, cujos ficheiros gerados

foram corretamente associados aos *IDs* dos componentes a que dizem respeito.



Figura 5.10: Geração de ROI do caso de estudo 1: (a) de componentes e (b) de *labels*.

Relativamente ao mapa de segmentação, de referir que é criado com sucesso contendo assim o modelo correto deste circuito. Uma vez que este mapa é extenso, aqui o foco residirá apenas nos resultados obtidos para o componente V1 do circuito, incluídos no excerto descrito na Listagem 5.1 (note-se que foram omitidos campos). Analisando esse excerto, verifica-se que V1, cujo *ID* está definido na linha 3, apresenta dois *ports* que estão ligados a duas conexões com *IDs* declarados nas linhas 7 e 11. Repare-se que os campos de *start/end* dessas conexões possuem os respetivos *IDs* dos *ports* do componente a que estão associadas (linhas 20 e 24), definindo assim as interligações deste componente. Visto que o circuito deste caso de estudo não apresenta nós, o campo *nodes* do mapa deverá estar vazio, tal como mostra a linha 16 da listagem.

```

1  "components": [
2      {
3          "id": "a479d047-a102-494a-9516-2ed4ebe1257c",
4          "ports": [
5              {
6                  "id": "ef2ee55c-af56-4824-ae27-513dec71bf6d",
7                  "connection": "74a67485-e333-4b2e-be1e-
                                aa4f50787bc5"
8              },
9              {
10                 "id": "729374f2-36bd-4cde-8de0-86ee163239b9",
11                 "connection": "0c17d0fb-7b14-4e29-9596-43
                                d59f1a2cde"
12             }
13         ]
14     }
15 ],
16 "nodes": [],
17 "connections": [
18     {
19         "id": "74a67485-e333-4b2e-be1e-aa4f50787bc5",
20         "end": "ef2ee55c-af56-4824-ae27-513dec71bf6d"
21     },
22     {
23         "id": "0c17d0fb-7b14-4e29-9596-43d59f1a2cde",
24         "start": "729374f2-36bd-4cde-8de0-86ee163239b9"
25     }
26 ]

```

Listagem 5.1: Excerto do mapa de segmentação do caso de estudo 1.

5.2 Circuito elétrico desenhado a computador

Como o módulo de *software* implementado deve também ter a capacidade de segmentação de circuitos desenhados a computador, o próximo caso de estudo pretende verificar exatamente esse requisito. O circuito elétrico em questão está ilustrado na Figura 5.11 (dimensões originais de 1100 x 490 píxeis), diferenciando-se do caso de estudo anterior pela presença de dois nós e pela associação de mais do que uma *label* ao mesmo componente (os componentes R1, R2 e R3 possuem duas *labels* respetivamente associadas a si).

Todos os resultados de processamento, as ROI geradas dos componentes e das *labels*, e o mapa de segmentação referentes a este circuito podem ser acedidos em *docs/results/circuit-3*.

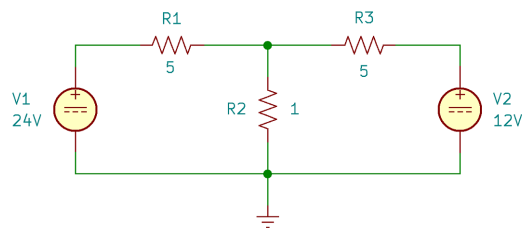


Figura 5.11: Circuito elétrico do caso de estudo 2.

Para o presente caso de estudo, é possível verificar que a imagem do circuito é mais nítida comparando com a imagem do caso de estudo apresentado previamente, encontrando-se o circuito neste caso sobre um fundo branco com os seus elementos bem delineados. Os resultados do pré-processamento desta imagem estão representados nas Figuras 5.12 e 5.13, preparando desta forma a imagem para o processo de segmentação.

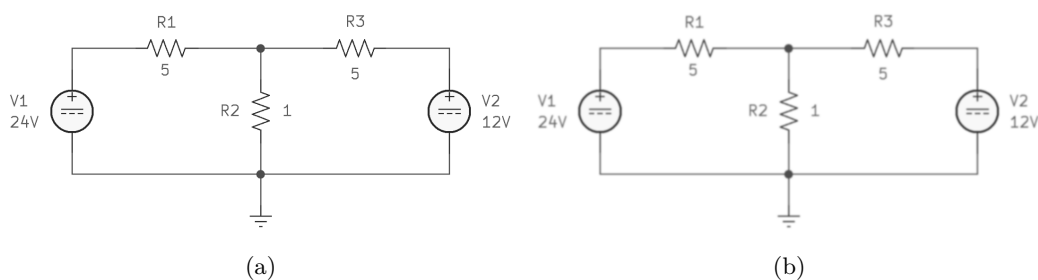
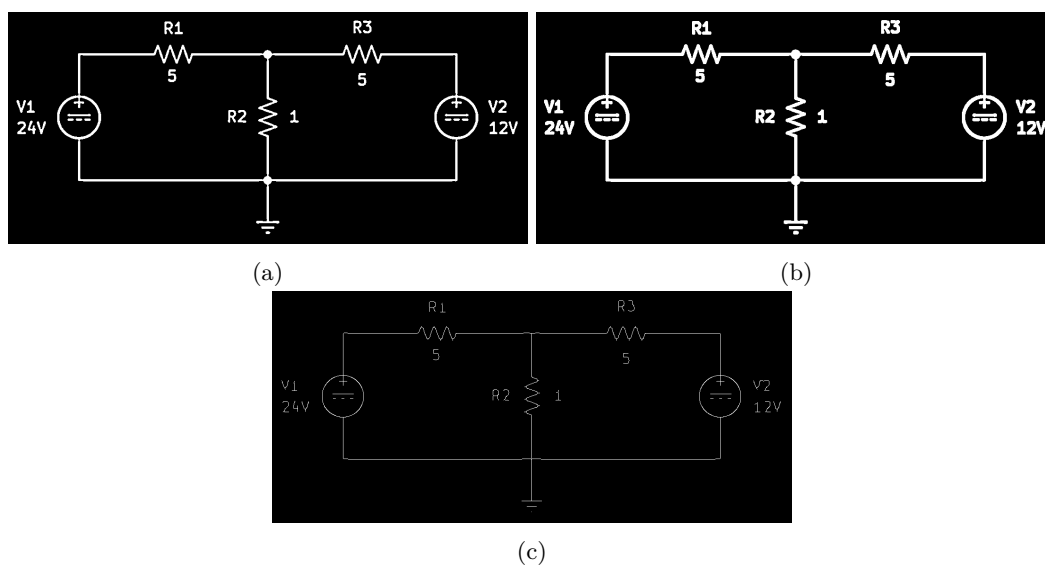


Figura 5.12: Pré-processamento do caso de estudo 2: (a) escala de cinza e (b) filtragem.

Figura 5.13: Pré-processamento do caso de estudo 2: (a) *thresholding*, (b) dilatação e (c) esqueletização.

Na situação de um circuito desenhado a computador, as linhas das conexões do circuito normalmente são retas e contínuas, logo não se verifica a extração de interligações quebradas entre os componentes como se observou no passo análogo para o caso de estudo anterior, cujo circuito foi desenhado “à mão”. As etapas do processo de detecção das conexões para este caso de estudo podem ser visualizadas nas Figuras 5.14 e 5.15.

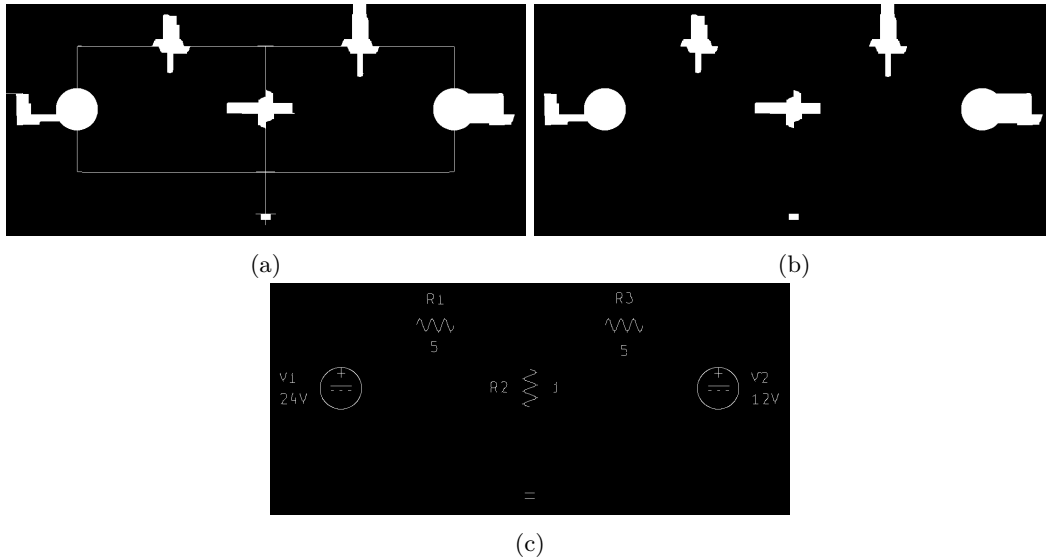


Figura 5.14: Detecção de conexões do caso de estudo 2: (a) *closing*, (b) *opening* e (c) interseção.

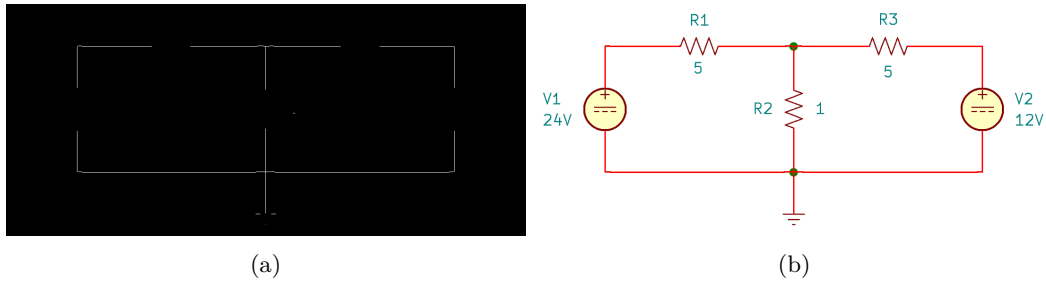


Figura 5.15: Detecção de conexões do caso de estudo 2: (a) remoção de componentes e de *labels*, e (b) conexões detetadas.

Relativamente à segmentação dos componentes do circuito, esta é realizada com sucesso através dos passos incluídos na Figura 5.16, detetando-se corretamente todos os componentes presentes no mesmo.

Na próxima etapa são atualizadas as conexões detetadas que, para o caso deste circuito, acaba por apenas adicionar as *labels* ao conjunto das interligações extraídas, como demonstra a Figura 5.17. Não obstante, no processo de detecção de nós são removidas as *labels* adicionadas ao agrupamento das conexões, sendo também identificada a presença de dois nós para este caso de estudo. A Figura 5.18a inclui

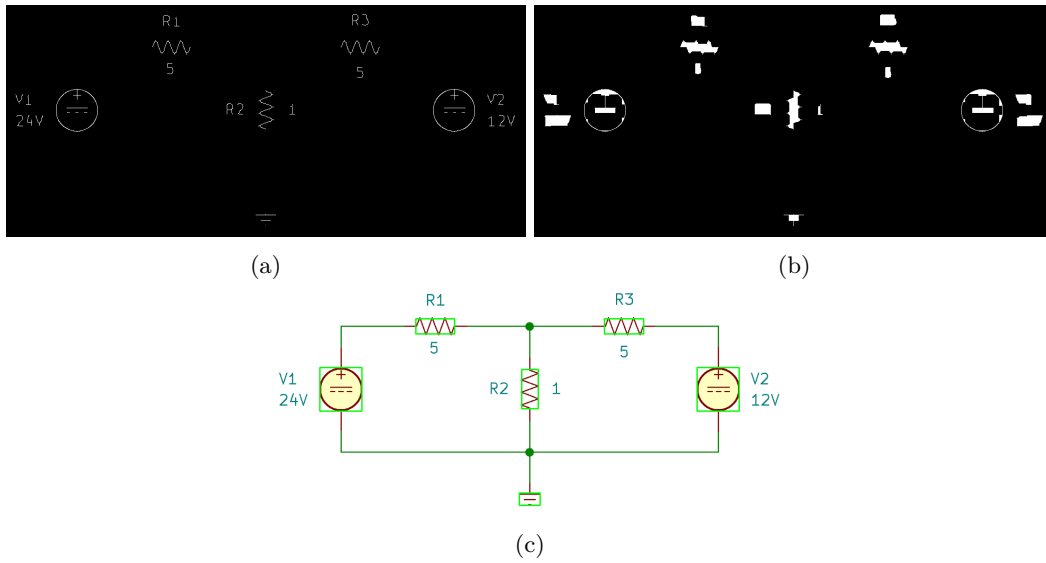


Figura 5.16: Detecção de componentes do caso de estudo 2: (a) remoção de conexões, (b) *closing* e (c) componentes detetados.

a detecção resultante dos nós localizados entre R1, R2 e R3, e também entre V1, R2, V2 e a massa do circuito, constatando-se que estes nós derivam de conexões com mais do que dois componentes. Com as conexões e os nós segmentados, são localizados os *ports* dos componentes anteriormente segmentados, com as posições assinaladas na Figura 5.18b para este circuito.

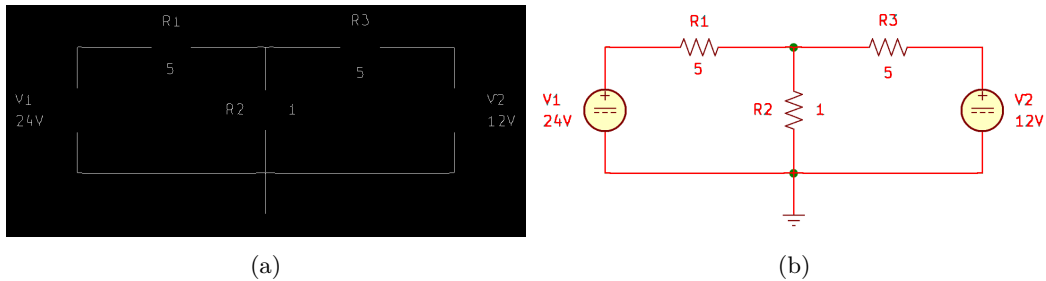


Figura 5.17: Atualização de conexões detetadas do caso de estudo 2: (a) remoção de componentes e (b) conexões atualizadas.

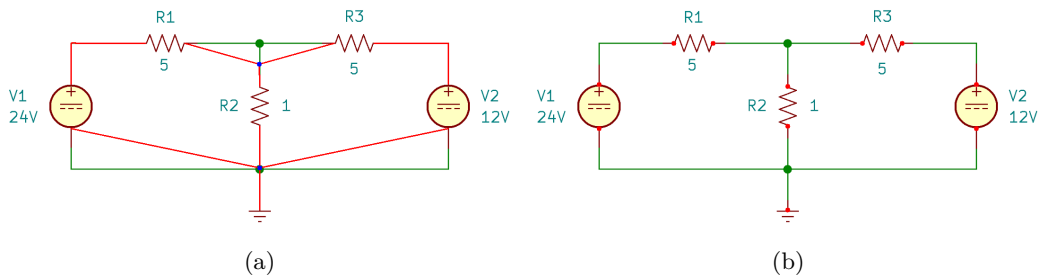


Figura 5.18: Caso de estudo 2: (a) nós detetados e conexões atualizadas, e (b) *ports* de componentes detetados.

Somente falta a detecção das *labels* dos elementos deste circuito. Nesse sentido, os componentes, as conexões e os nós são removidos da imagem, tal como demonstra a Figura 5.19a. Porém, a imagem após a remoção dos elementos não possui apenas as *labels*, contendo adicionalmente as conexões que abrangem os nós do circuito. Assim, após a transformação morfológica de *closing* (Figura 5.19b), é necessário a aplicação da operação de *opening*, eliminando estas conexões da imagem e possibilitando seguidamente a extração com sucesso das *labels* do circuito, como se verifica na Figura 5.20.

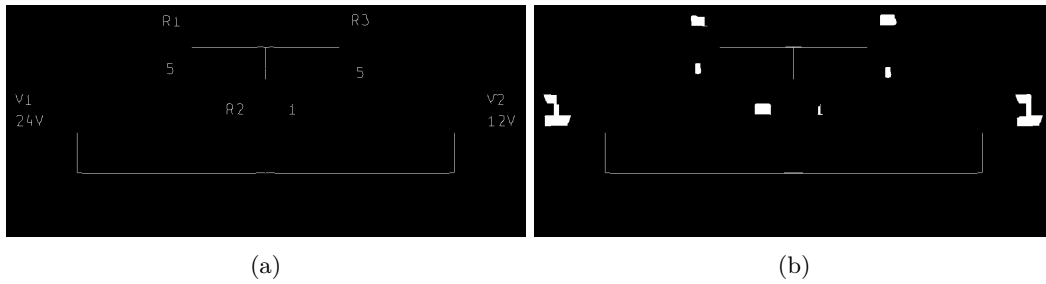


Figura 5.19: Detecção de *labels* do caso de estudo 2: (a) remoção dos elementos e (b) *closing*.

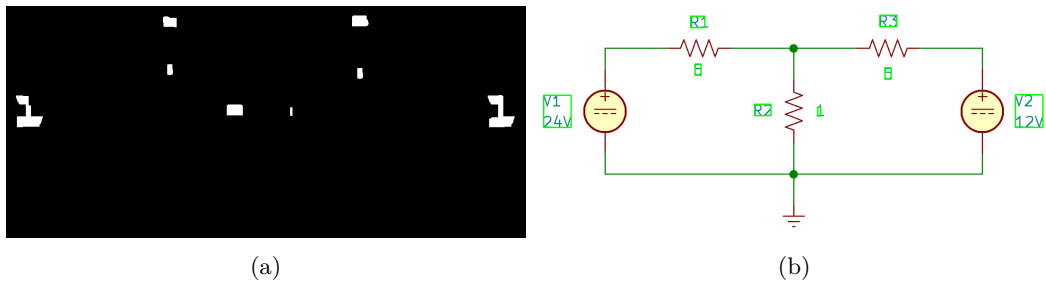


Figura 5.20: Detecção de *labels* do caso de estudo 2: (a) *opening* e (b) *labels* detetadas.

O processo de segmentação chega ao fim, e as ROI relativas a cada componente e *label* podem ser geradas pelo *software*. A Figura 5.21 incorpora estas imagens criadas para este caso de estudo.



Figura 5.21: Geração de ROI do caso de estudo 2: (a) de componentes e (b) de *labels*.

O último processo a executar pelo *software* é a geração do mapa de segmentação referente ao modelo deste circuito, sendo também criado corretamente para este caso de estudo. De destacar aqui o campo do mapa relativo aos nós do circuito,

particularmente o nó que abarca três conexões que ligam os componentes R1, R2 e R3, utilizando o fragmento do modelo da Listagem 5.2 (com diversos campos suprimidos). Na descrição deste nó, cujo *ID* está definido na linha 3, constata-se que está ligado às conexões com os *IDs* das linhas 5, 6 e 7. Observando com atenção os campos *end* dessas conexões, retira-se que estas estão relacionadas com este nó, modelizando assim com sucesso esta parte do circuito elétrico.

```
1  "nodes": [  
2    {  
3      "id": "2fb1a1d6-5961-478b-9abc-5d5539d07f35",  
4      "connections": [  
5        "6285eb89-0f95-445a-afc9-7f9d94512d50",  
6        "c2e0d80c-abb5-4c34-8b64-f31875b24150",  
7        "4d355ecc-c5bf-437d-a2ef-ec164c57bd51"  
8      ]  
9    }  
10 ],  
11 "connections": [  
12   {  
13     "id": "6285eb89-0f95-445a-afc9-7f9d94512d50",  
14     "end": "2fb1a1d6-5961-478b-9abc-5d5539d07f35"  
15   },  
16   {  
17     "id": "c2e0d80c-abb5-4c34-8b64-f31875b24150",  
18     "end": "2fb1a1d6-5961-478b-9abc-5d5539d07f35"  
19   },  
20   {  
21     "id": "4d355ecc-c5bf-437d-a2ef-ec164c57bd51",  
22     "end": "2fb1a1d6-5961-478b-9abc-5d5539d07f35"  
23   }  
24 ]
```

Listagem 5.2: Excerto do mapa de segmentação do caso de estudo 2.

5.3 Circuito elétrico complexo

Neste caso de estudo é aplicado um circuito elétrico mais complexo que os anteriores, apresentando múltiplos componentes, conexões e nós, de forma a verificar o comportamento do *software* quando recebe circuitos com um grande número de elementos. Este encontra-se representado na Figura 5.22, com dimensões originais da respetiva imagem de 1158 x 864 píxeis, tendo sido desenhado a computador.

Algumas imagens obtidas do processamento relativas a este caso de estudo serão apresentadas nesta secção. Para este caso, a reflexão recairá sobre os resultados de

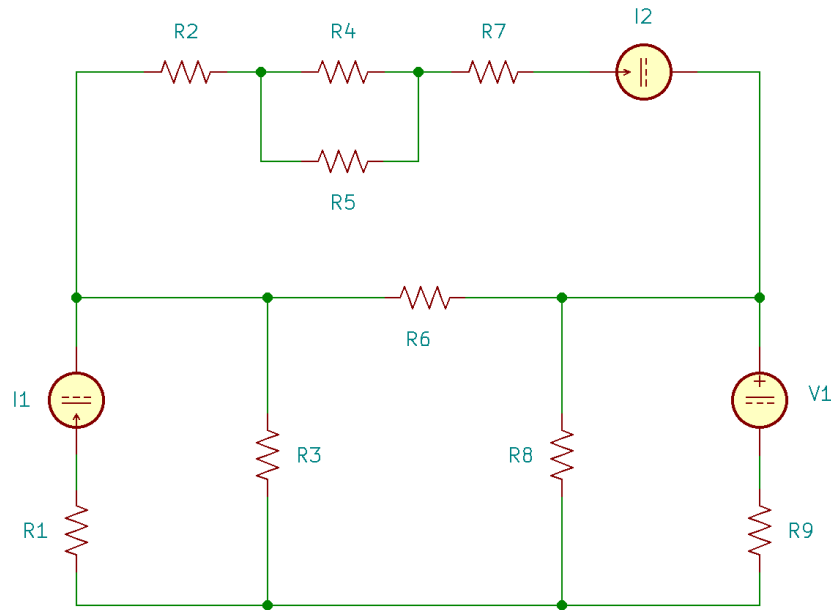


Figura 5.22: Circuito elétrico do caso de estudo 3.

pré-processamento e de segmentação finais dos elementos do circuito, sem demonstrar as etapas intermédias que conduziram a tais desfechos. Contudo, para acesso a todas as imagens resultantes do processamento, assim como às ROI dos componentes e das *labels* e ao mapa de segmentação gerados para este circuito, pode ser explorado o diretório *docs/results/circuit-4* do *software* do projeto.

A Figura 5.23 ilustra o resultado do pré-processamento da imagem do circuito referente a este caso de estudo. Os contornos dos componentes e das conexões do circuito estão bem delineados sobre um fundo escuro, bem como os caracteres alfanuméricos das *labels* dos componentes, habilitando deste modo o circuito para o processo subsequente de deteção e extração dos seus elementos.

Os componentes segmentados pelo *software* contidos neste caso de estudo estão assinalados na Figura 5.24. Estes resultados demonstram que, apesar do grande número destes elementos que compõem o circuito (12 componentes), o módulo foi capaz de os localizar na imagem e extrair corretamente todas as resistências e as fontes de tensão e de corrente presentes no mesmo.

Em relação às conexões e aos nós deste circuito, estes foram corretamente detetados como comprova a Figura 5.25, registando-se a extração de 5 nós e de 21 conexões a partir da imagem. De notar a forma como os nós foram segmentados para este circuito, em particular nas situações em que na imagem existem dois pontos que pertencem ao mesmo nó e que interligam várias conexões. Exemplo desta ocorrência é o nó entre I1, R2, R3 e R6, em que o *software* reconhece que existe apenas uma conexão entre estes quatro elementos, e como esta caracteriza uma interligação entre mais do que dois componentes, é considerado que existe aqui um nó do circuito.

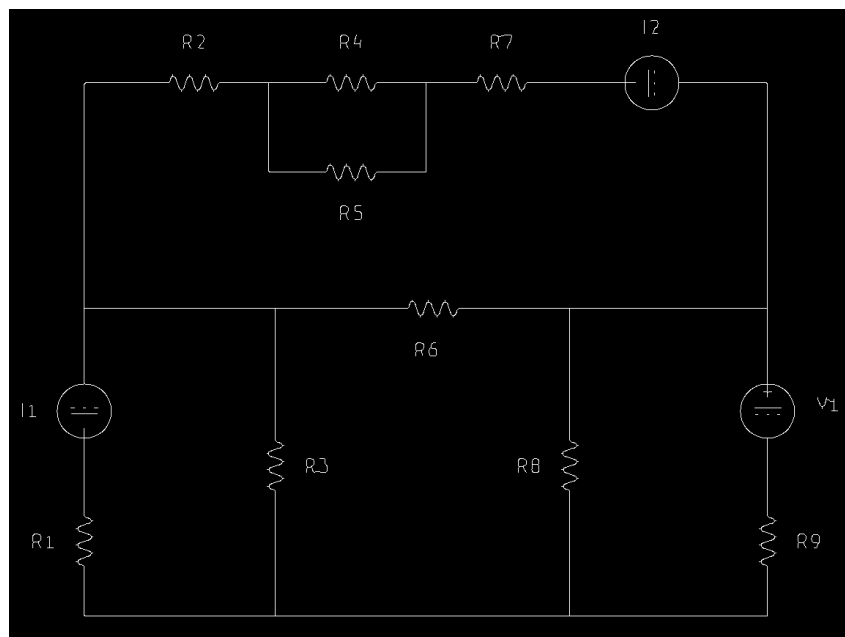


Figura 5.23: Pré-processamento do caso de estudo 3.

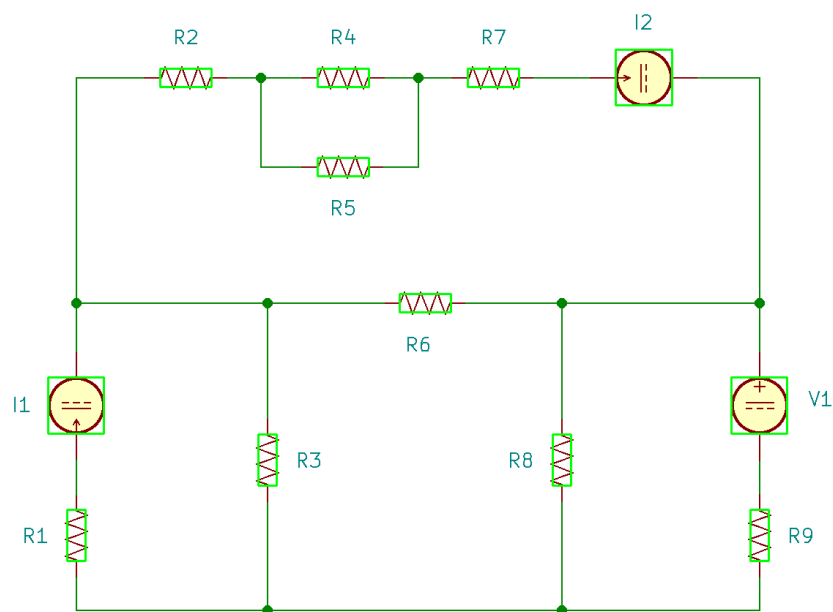


Figura 5.24: Detecção de componentes do caso de estudo 3.

Com a descoberta dos nós e das conexões, os *ports* dos componentes segmentados podem ser situados, resultando para este circuito os pontos de conexão demarcados na Figura 5.26.

Por último, as *labels* alusivas aos elementos do circuito podem ser detetadas na imagem, cuja segmentação obtida para este caso de estudo está representada na Figura 5.27.

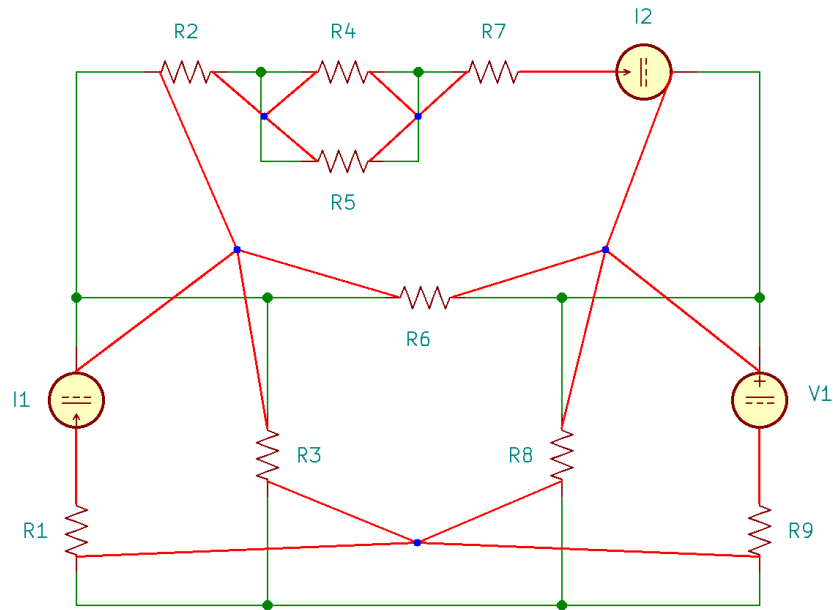


Figura 5.25: Detecção de nós e de conexões do caso de estudo 3.

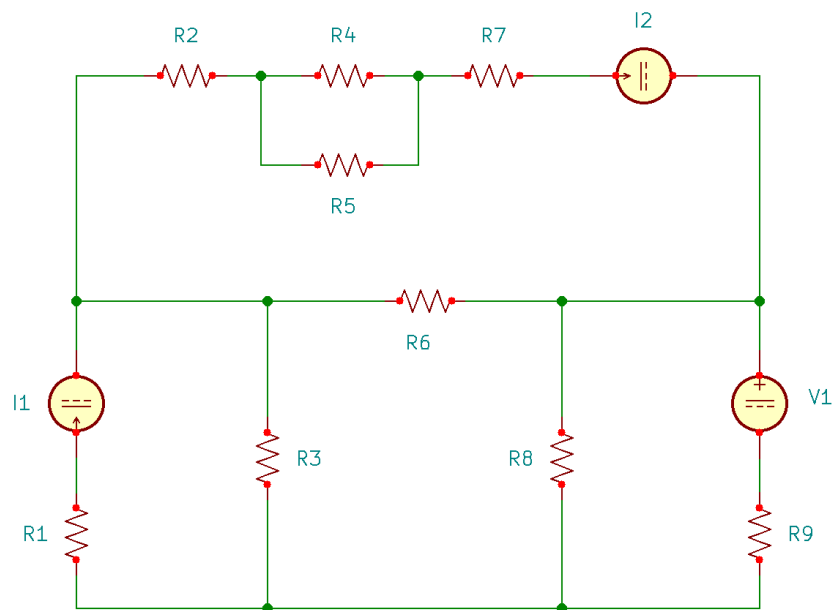


Figura 5.26: Detecção de *ports* de componentes do caso de estudo 3.

Analisando os resultados obtidos, conclui-se que o módulo de *software* foi capaz de segmentar um circuito elétrico de nível de complexidade elevado, abrangendo um número considerável de componentes, nós, conexões e *labels*.

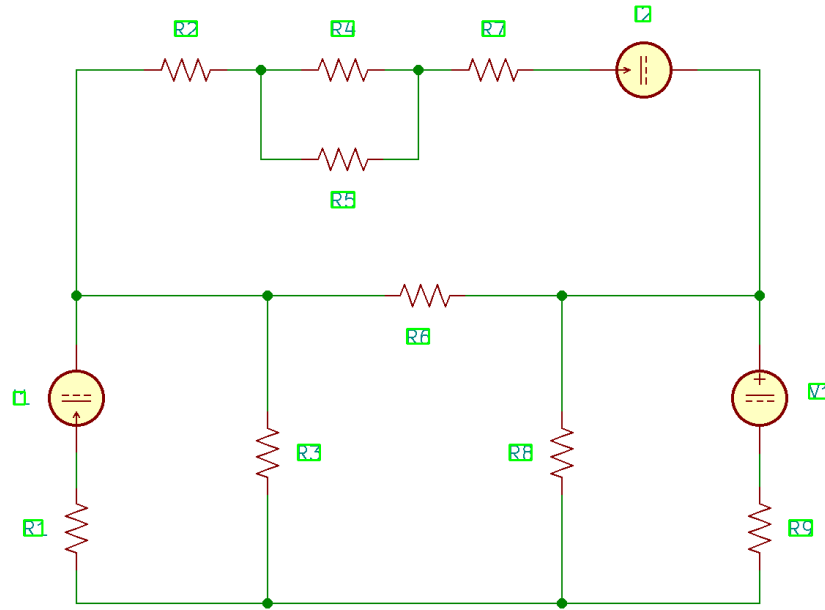


Figura 5.27: Detecção de *labels* do caso de estudo 3.

5.4 Limitações

O módulo de *software* desenvolvido poderá não gerar sempre um modelo do circuito elétrico de uma imagem de forma correta, apresentando assim algumas limitações que o utilizador deverá ter em conta no seu uso.

Uma das limitações é o facto da dimensão das áreas ocupadas na imagem pelos diferentes elementos do circuito influenciar nos resultados de segmentação gerados. Isto deve-se aos valores constantes utilizados para os tamanhos de *kernel* usados nas transformações morfológicas durante o processo de segmentação, que delimitam a área de aplicação destas operações. Esta limitação é constatada com a utilização de uma imagem do circuito-exemplo com dimensão superior à usada na Secção 4.2. Como se verificou anteriormente, o circuito-exemplo é corretamente segmentado quando a sua imagem possui um tamanho de 600 x 382 píxeis, porém quando este mesmo circuito está redimensionado com 1200 x 748 píxeis, o mesmo resultado já não é obtido. A Figura 5.28 compara os resultados obtidos da aplicação da transformação morfológica de *closing* durante o processo de deteção de componentes, para ambas as imagens do circuito-exemplo. Repare-se como as regiões homólogas das imagens possuem amplificações diferentes, em especial a zona que engloba o símbolo da fonte de tensão alternada do circuito, onde surge quase todo preenchido com píxeis brilhantes para a imagem de dimensão inferior, não se verificando quase nenhuma alteração na imagem maior, uma vez que este símbolo ocupa uma área superior na última imagem à ocupada na primeira. Assim, a segmentação dos componentes do circuito é condicionada pelo efeito gerado na imagem

pelas operações morfológicas, tal como demonstra a Figura 5.29, na qual se retira que a deteção não foi realizada corretamente para a imagem de dimensão superior, considerando uma das *labels* presentes, bem como o nó e partes de conexões, como componentes.

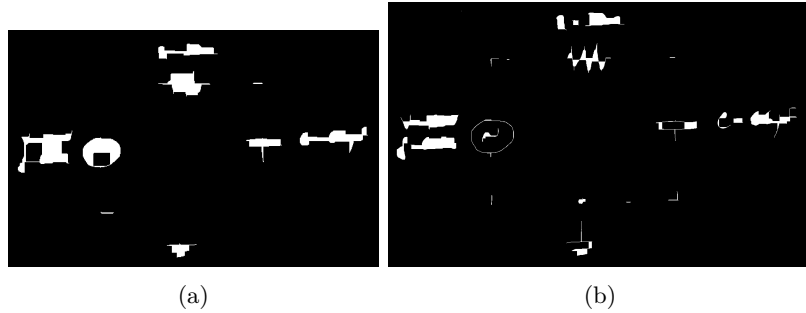


Figura 5.28: Comparação da transformação de *closing* para a imagem do circuito com dimensões de (a) 600 x 382 e de (b) 1200 x 748 píxeis.

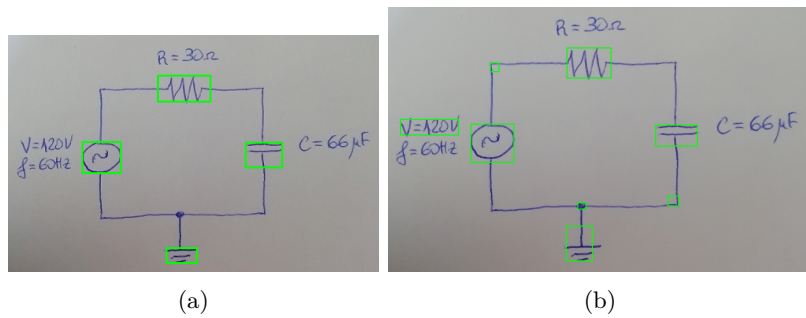


Figura 5.29: Comparação da deteção de componentes para a imagem do circuito com dimensões de (a) 600 x 382 e de (b) 1200 x 748 píxeis.

Outra limitação também conhecida está relacionada com a proximidade entre componentes, visto que diferentes componentes que se encontrem muito próximos podem ser detetados como sendo somente um único componente. Esta limitação é devido também às transformações morfológicas que, com a dilatação excessiva dos símbolos dos componentes, acabam por conduzir à sua junção. Um exemplo ilustrativo desta limitação está representado na Figura 5.30, em que os componentes R1 e R2 são detetados como um só componente.

Similar à limitação anterior, verifica-se igualmente que *labels* muito próximas de componentes podem ser consideradas como pertencendo a esses componentes, consequência exatamente da mesma razão referida anteriormente, provocando a união destes elementos. A Figura 5.31 exemplifica esta limitação, onde as *labels* de R8, R10 e R11 são detetadas como pertencentes aos respetivos componentes.

Finalmente, utilizando o mesmo circuito da limitação mencionada previamente, é também possível conferir que *labels* referentes a nós de um circuito podem não ser associadas aos correspondentes nós mas a conexões ligadas a estes. A razão para

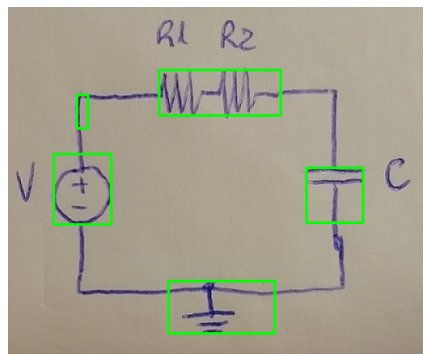


Figura 5.30: Junção de dois componentes em apenas um.

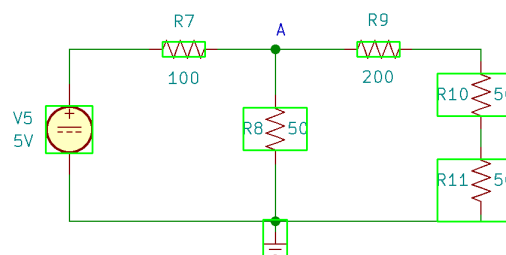


Figura 5.31: Detecção de *labels* como componentes do circuito.

esta incorreta associação é originada pela metodologia aplicada no *software* para a deteção de nós, em que não se obtém a posição original do nó levando assim a que o nó detetado não esteja situado nas mesmas coordenadas iniciais. Na Figura 5.32 observa-se que a *label* com a letra “A” diz respeito ao nó entre os componentes R7, R8 e R9, todavia esta é associada à conexão que vai desde o nó “equivalente” criado pelo *software* até a R9, devido a se encontrar mais próxima da área de associação dessa conexão (delimitada a verde) e estar mais afastada do nó criado.

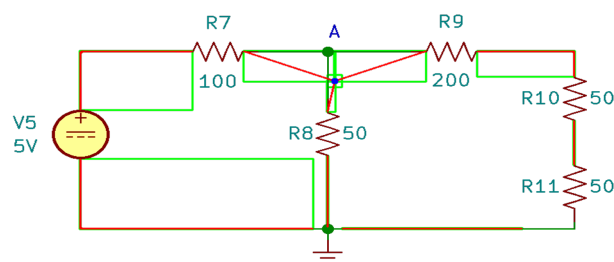


Figura 5.32: *Label* de um nó associada a uma conexão do circuito.

Capítulo 6

Conclusões

O desenvolvimento do presente projeto teve, como principal motivação, contribuir para o ensino e aprendizagem de metodologias de análise de circuitos elétricos, direcionado em particular para os estudantes que se encontrem na fase inicial de aquisição de conhecimentos desta temática da Engenharia Eletrotécnica. Com isso em mente, esta dissertação incidiu na implementação de um módulo de *software* que fosse capaz de interpretar e modelar esquemas elétricos a partir de uma imagem, desenhados “à mão” ou a computador, baseando-se em algoritmos de visão computacional. A conceção da arquitetura deste módulo foi efetuada tendo em conta diversos requisitos específicos, como o modelo a ser seguido pelo mapa resultante do processo de segmentação, para que este possa ser mais facilmente integrado na ferramenta U=RI solve no futuro.

Para a concretização do *software* deste trabalho, foi necessário a realização de um estudo aprofundado de noções e conceitos básicos de visão computacional, para obtenção de conhecimentos sobre os algoritmos pertencentes a esta área. Visto que o presente projeto estabeleceu o primeiro contacto com o vasto campo da visão computacional, este estudo exigiu algum tempo para entendimento, não só dos aspetos teóricos envolvidos nos seus algoritmos, mas também de como estes poderiam ser aplicados na prática para o pré-processamento e segmentação de uma imagem de um circuito elétrico. Esta foi uma dificuldade inicialmente sentida que, simultaneamente, também constituiu um grande desafio para o desenvolvimento do *software* deste projeto. Em conjunto com o estudo da visão computacional, procedeu-se a uma análise de várias investigações científicas relacionadas com a segmentação e

o reconhecimento de esquemas elétricos, para percepção de quais as metodologias que são frequentemente mais utilizadas e apropriadas para a detecção e extração de componentes, nós, conexões e *labels* através de uma imagem de um circuito. Aqui também foram enfrentadas algumas dificuldades, uma vez que não existe um método único que seja sempre aplicado para a segmentação de um circuito, adicionado ao facto de nem todos os estudos servirem para a mesma finalidade, isto é, alguns focam-se apenas na detecção de componentes sem verificarem as suas interligações, outros pretendem extrair os componentes e as conexões mas em circuitos que não contêm *labels*, registando-se ainda outras investigações que combinam algoritmos de visão computacional com técnicas de OCR (para identificação de *labels*), de *machine learning* e de *deep learning* para o reconhecimento de um circuito elétrico. Assim, foram experimentados diversos algoritmos e efetuadas inúmeras tentativas para a seleção de qual a metodologia mais adequada a implementar para este projeto, tendo em conta a qualidade de imagem do circuito elétrico e também as várias formas com que este pode ser desenhado.

Os objetivos inicialmente traçados para este trabalho foram concretizados, uma vez que se desenvolveu um módulo de *software* com capacidade de segmentação e de modelação de um circuito a partir de uma imagem. Com recurso a métodos de visão computacional, este módulo deteta e extrai os elementos presentes num circuito elétrico, nomeadamente os componentes, nós, interligações entre os elementos, e as *labels* associadas a cada elemento do circuito. Além disso, o *software* gera ficheiros de imagem relativos às ROI dos componentes e das *labels* extraídas, que poderão ser posteriormente utilizados por um outro módulo de *software* dedicado à classificação dos componentes segmentados e ao reconhecimento dos caracteres alfanuméricos presentes nas *labels*. Do mesmo modo, o módulo implementado produz também um ficheiro em formato JSON representativo do modelo do circuito, que contém o mapa/matriz resultante do processo de segmentação efetuado pelo *software*, agrupando assim os elementos do circuito detetados e as suas interligações.

Este projeto, atendendo aos desafios que foram surgindo durante o seu desenvolvimento referidos previamente, possibilitou a aprendizagem de conceitos relacionados com a tecnologia da visão computacional, que é uma área cada vez mais relevante atualmente e com um grande número de aplicações. Adicionalmente, também concedeu a consolidação de competências de programação usando a linguagem C++, assim como permitiu trabalhar com diversas ferramentas e bibliotecas de *software*, tais como CMake, Docker, OpenCV e GoogleTest. De referir ainda que a realização deste projeto providenciou também progressos a nível de *soft skills*, especialmente de organização, gestão de tempo e de comunicação, facultados pelas reuniões e discussões ocorridas ao longo da sua elaboração.

6.1 Trabalho Futuro

O módulo de *software* desenvolvido apresenta atualmente algumas limitações no processo de segmentação, fundamentalmente devido à dependência da área de píxeis ocupada por cada elemento do circuito na imagem, que pode levar à sua incorreta detecção. Assim, a adição de um algoritmo que permitisse uma adaptação dinâmica da segmentação a diferentes dimensões dos elementos do circuito na imagem, melhoraria de forma significativa os resultados gerados por este *software*. A metodologia utilizada para a detecção de nós também poderá ser sujeita a ajustes, para possibilitar a extração da posição original do nó na imagem do circuito, o que providencia posteriormente a correta associação da correspondente *label* que possa apresentar. Adicionalmente, podem ser também introduzidos melhoramentos nas imagens geradas com as ROI do circuito, nomeadamente centrando os componentes e as *labels* nestas imagens, visto que no módulo atual essa disposição não se efetiva sempre, sobretudo em circuitos desenhados “à mão”.

Para que este módulo possa ser integrado na aplicação U=RIolve, será necessário providenciar-lhe a funcionalidade de receção da imagem do circuito por um método de requisição suportado por *Hypertext Transfer Protocol* (HTTP), permitindo que a imagem do circuito possa ser submetida pela interface gráfica de U=RIolve e, posteriormente, processada pelo *software* deste projeto. Como o módulo desenvolvido não efetua a classificação dos componentes segmentados e também não reconhece os caracteres alfanuméricos das *labels*, essas tarefas deverão ser incluídas num outro módulo de *software* a implementar no futuro, proporcionado a criação de um modelo completo do circuito segmentado. Deste modo, os estudantes não necessitarão de desenhar o circuito pretendido para análise no editor de circuitos da *framework* U=RIolve, possibilitando assim a obtenção dos resultados de análise providenciados por esta ferramenta de forma mais rápida.

Adicionalmente, está a ser planeada a escrita de um artigo científico relacionado com a presente dissertação.

Referências

- [1] U=RIolve, “U=RIolve Simulator.” Disponível em <https://urisolve.pt/app/>. (Consultado a 25/08/2022). [Citado na página 2]
- [2] L. Sousa, “U=RIolve: a web-based tool for teaching & self-learning the Nodal Voltage Method.” Mestrado em Engenharia Eletrotécnica e de Computadores, Dissertação, Instituto Superior de Engenharia do Porto, 2020. [Citado na página 2]
- [3] M. Duarte, “Aplicação para geração da metodologia/equações do Método das Correntes nas Malhas, a partir do QUCS.” Licenciatura em Engenharia Eletrotécnica e de Computadores, Projeto Final, Instituto Superior de Engenharia do Porto, 2019. [Citado na página 2]
- [4] J. Soares, “Aplicação para geração da metodologia/equações do Método das Correntes nos Ramos a partir do QUCS.” Licenciatura em Engenharia Eletrotécnica e de Computadores, Projeto Final, Instituto Superior de Engenharia do Porto, 2019. [Citado na página 2]
- [5] P. Morim, “Circuit Editor - Design, Implementation, and Integration into the U=RIolve web application.” Licenciatura em Engenharia Eletrotécnica e de Computadores, Projeto Final, Instituto Superior de Engenharia do Porto, 2022. [Citado nas páginas 2 e 42]
- [6] L. Sousa, A. Rocha, M. Alves and F. Pereira, “U=RIolve: A Web-Based Application for Learning Electrical Circuit Analysis [Education],” *IEEE Circuits and Systems Magazine*, vol. 21, no. 3, pp. 66–95, 2021. [Citado na página 2]
- [7] L. Sousa, A. Rocha, M. Alves and F. Pereira, “Revisiting the nodal voltage method for both human comprehension and software implementation: Towards a teaching/self-learning simulation tool,” *Computer Applications in Engineering Education*, vol. 29, pp. 1642–1664, Nov. 2021. [Citado na página 2]
- [8] U=RIolve, “Circuit Segmentation and Modeling.” Disponível em <https://github.com/urisolve/circuit-segmentation>, 2022. (Consultado a 18/10/2022). [Citado na página 4]
- [9] N. Babich, “What Is Computer Vision & How Does it Work? An Introduction.” Disponível em <https://xd.adobe.com/ideas/principles/>

- emerging-technology/what-is-computer-vision-how-does-it-work/, July 2020. (Consultado a 29/08/2022). [Citado nas páginas ix, 7, 8 e 14]
- [10] R. Szeliski, *Computer Vision: Algorithms and Applications*, pp. 1–17. Springer, 2nd ed., Jan. 2022. [Citado nas páginas ix, 7 e 10]
- [11] R. Davies, *Computer Vision: Principles, Algorithms, Applications, Learning*, pp. 1–12. Academic Press, 5th ed., Nov. 2017. [Citado na página 7]
- [12] IBM, “What is computer vision?.” Disponível em <https://www.ibm.com/topics/computer-vision>. (Consultado a 29/08/2022). [Citado na página 8]
- [13] S. J. D. Prince, *Computer Vision: Models, Learning, and Inference*, pp. 1–2. Cambridge University Press, 1st ed., June 2012. [Citado nas páginas ix, 8, 9 e 11]
- [14] A. Rizzoli, “27+ Most Popular Computer Vision Applications and Use Cases in 2022.” Disponível em <https://www.v7labs.com/blog/computer-vision-applications>, Aug. 2022. (Consultado a 04/09/2022). [Citado nas páginas ix, 12 e 13]
- [15] J. Cohen, “Computer Vision Applications in Self-Driving Cars.” Disponível em <https://www.thinkautonomous.ai/blog/computer-vision-applications-in-self-driving-cars/>, Nov. 2019. (Consultado a 04/09/2022). [Citado nas páginas ix e 12]
- [16] G. Boesch, “Top 10 Applications Of Deep Learning and Computer Vision In Healthcare.” Disponível em <https://viso.ai/applications/computer-vision-in-healthcare/>. (Consultado a 06/09/2022). [Citado nas páginas ix e 14]
- [17] M. Stumpe and L. Peng, “Assisting Pathologists in Detecting Cancer with Deep Learning.” Disponível em <https://ai.googleblog.com/2017/03/assisting-pathologists-in-detecting.html>, Mar. 2017. (Consultado a 08/09/2022). [Citado nas páginas ix, 14 e 15]
- [18] G. Boesch, “Face Detection: Real-time applications with deep learning (2022 Guide).” Disponível em <https://viso.ai/deep-learning/face-detection-overview/>. (Consultado a 08/09/2022). [Citado nas páginas ix, 14 e 15]
- [19] R. Kundu, “Image Processing: Techniques, Types, & Applications [2022].” Disponível em <https://www.v7labs.com/blog/image-processing-guide>, Aug. 2022. (Consultado a 10/09/2022). [Citado nas páginas ix, 16, 17 e 18]
- [20] Simplilearn, “What Is Image Processing: Overview, Applications, Benefits, and Who Should Learn It.” Disponível em <https://www.simplilearn.com/>

- image-processing-article, July 2022. (Consultado a 10/09/2022). [Citado na página 16]
- [21] A. Rosebrock, “Convolutions with OpenCV and Python.” Disponível em <https://pyimagesearch.com/2016/07/25/convolutions-with-opencv-and-python/>, July 2016. (Consultado a 10/09/2022). [Citado nas páginas ix, 16, 18 e 19]
- [22] H. Marius, “Understanding the Basics of Digital Image Processing and Computer Vision using OpenCV.” Disponível em <https://towardsdatascience.com/understanding-the-basics-of-digital-image-processing-and-computer-vision-using-opencv-8bc0e67243c8>, June 2020. (Consultado a 10/09/2022). [Citado nas páginas 16 e 18]
- [23] Y. Ç. Aktaş, “A Comprehensive Guide to Image Processing: Fundamentals.” Disponível em <https://towardsdatascience.com/image-processing-4391c5bcef78>, Aug. 2021. (Consultado a 10/09/2022). [Citado nas páginas ix e 17]
- [24] M. Basavarajaiah, “6 basic things to know about Convolution.” Disponível em <https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411>, Mar. 2019. (Consultado a 11/09/2022). [Citado nas páginas ix e 20]
- [25] Dharmaraj, “Zero-Padding in Convolutional Neural Networks.” Disponível em <https://medium.com/@draj0718/zero-padding-in-convolutional-neural-networks-bf1410438e99>, Sept. 2021. (Consultado a 11/09/2022). [Citado nas páginas ix e 20]
- [26] A. Kaehler and G. Bradski, *Learning OpenCV 3*, pp. 1–38, 249–294, 335–371. O’Reilly Media, 1st ed., Dec. 2016. [Citado nas páginas ix, 20, 21, 22, 23, 24, 26, 28, 37 e 38]
- [27] A. Rosebrock, “OpenCV Smoothing and Blurring.” Disponível em <https://pyimagesearch.com/2021/04/28/opencv-smoothing-and-blurring/>, Apr. 2021. (Consultado a 11/09/2022). [Citado nas páginas 20, 21, 22 e 23]
- [28] A. Huamán, “Smoothing Images.” Disponível em https://docs.opencv.org/4.5.5/dc/dd3/tutorial_gaussian_median_blur_bilateral_filter.html, Dec. 2021. (Consultado a 11/09/2022). [Citado na página 21]
- [29] A. Rosebrock, “OpenCV Thresholding (cv2.threshold).” Disponível em <https://pyimagesearch.com/2021/04/28/opencv-thresholding-cv2-threshold/>, Apr. 2021. (Consultado a 12/09/2022). [Citado nas páginas 24 e 26]

- [30] A. Huamán, “Basic Thresholding Operations.” Disponível em https://docs.opencv.org/4.5.5/db/d8e/tutorial_threshold.html, Dec. 2021. (Consultado a 12/09/2022). [Citado nas páginas ix, x, 24, 25, 26 e 27]
- [31] A. Rosebrock, “Adaptive Thresholding with OpenCV (cv2.adaptiveThreshold).” Disponível em <https://pyimagesearch.com/2021/05/12/adaptive-thresholding-with-opencv-cv2-adaptivethreshold/>, May 2021. (Consultado a 12/09/2022). [Citado nas páginas x, 26 e 27]
- [32] A. Rosebrock, “OpenCV Morphological Operations.” Disponível em <https://pyimagesearch.com/2021/04/28/opencv-morphological-operations/>, Apr. 2021. (Consultado a 12/09/2022). [Citado nas páginas x, 28, 30 e 31]
- [33] A. Huamán, “Eroding and Dilating.” Disponível em https://docs.opencv.org/4.5.5/db/df6/tutorial_erosion_dilatation.html, Dec. 2021. (Consultado a 12/09/2022). [Citado nas páginas x e 29]
- [34] A. Huamán, “More Morphology Transformations.” Disponível em https://docs.opencv.org/4.5.5/d3/db6/tutorial_opening_closing_hats.html, Dec. 2021. (Consultado a 12/09/2022). [Citado nas páginas x, 29 e 30]
- [35] M. Moetesum, S. W. Younus, M. A. Warsi and I. Siddiqi, “Segmentation and Recognition of Electronic Components in Hand-Drawn Circuit Diagrams,” *EAI Endorsed Transactions on Scalable Information Systems*, vol. 4, Apr. 2018. [Citado nas páginas x, 31, 32, 33, 34 e 35]
- [36] M. S. Abhinav, Shrimanth V. L., Ganesh P. N., Nishanth P. V. and Bhargavi K., “Electric Circuit Diagram Detection, Recognition And Simulation,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 7, pp. 3221–3225, Sept. 2020. [Citado na página 32]
- [37] A. Mohan, A. Mohan, Indushree B., Malavikaa M., Narendra C. P., “Generation of Netlist from a Hand drawn Circuit through Image Processing and Machine Learning,” in *2022 2nd International Conference on Artificial Intelligence and Signal Processing (AISP)*, pp. 1–4, 2022. [Citado nas páginas x, 32, 35 e 36]
- [38] M. V. Rao, K. Hemanth, S. Razia and K. R. Kumar, “Synthesizing of Hand-drawn Electrical Circuits Using Machine Learning Techniques,” *International Journal of Emerging Trends in Engineering Research*, vol. 8, pp. 5523–5529, Sept. 2020. [Citado nas páginas 32 e 36]
- [39] M. Rabbani, R. Khoshkangini, H.S. Nagendraswamy and M. Conti, “Hand Drawn Optical Circuit Recognition,” *Procedia Computer Science*, vol. 84, pp. 41–48, 2016. [Citado nas páginas 32 e 36]

-
- [40] Y. Liu and Y. Xiao, “Circuit Sketch Recognition.” Stanford University, Stanford, CA. [Citado na página 32]
- [41] Rachala R.R. and Panicker M.R., “Hand-Drawn Electrical Circuit Recognition Using Object Detection and Node Recognition,” *SN Computer Science* 3, no. 244, 2022. [Citado nas páginas xi, 32, 35 e 36]
- [42] A. Bhattacharya, N. Sarkar, R. Sarkar, S. Roy and S. Malakar, “Circuit Component Detection in Offline Hand-drawn Electrical/Electronic Circuit Diagram,” in *2020 IEEE Calcutta Conference (CALCON)*, pp. 80–84, 2020. [Citado na página 35]
- [43] A. Huamán, “Hough Line Transform.” Disponível em https://docs.opencv.org/4.5.5/d9/db0/tutorial_hough_lines.html, Dec. 2021. (Consultado a 13/09/2022). [Citado na página 35]
- [44] A. Huamán, “Hough Circle Transform.” Disponível em https://docs.opencv.org/4.5.5/d4/d70/tutorial_hough_circle.html, Dec. 2021. (Consultado a 13/09/2022). [Citado na página 35]
- [45] G. Boesch, “The 12 Most Popular Computer Vision Tools in 2022.” Disponível em <https://viso.ai/computer-vision/the-most-popular-computer-vision-tools/>. (Consultado a 14/09/2022). [Citado na página 37]
- [46] OpenCV, “OpenCV: Open Source Computer Vision Library.” Disponível em <https://github.com/opencv/opencv>, 2022. (Consultado a 14/09/2022). [Citado nas páginas 37 e 38]
- [47] OpenCV, “About.” Disponível em <https://opencv.org/about/>, 2022. (Consultado a 14/09/2022). [Citado na página 37]
- [48] OpenCV documentation, “Introduction.” Disponível em <https://docs.opencv.org/4.6.0/d1/dfb/intro.html>, June 2022. (Consultado a 14/09/2022). [Citado na página 38]
- [49] Ihar Yermalayeu, “Simd.” Disponível em <https://github.com/ermig1979/Simd>, 2022. (Consultado a 14/09/2022). [Citado na página 39]
- [50] Boost, “Boost GIL.” Disponível em <https://github.com/boostorg/gil>, 2022. (Consultado a 14/09/2022). [Citado na página 39]
- [51] ImageMagick, “Magick++.” Disponível em <https://github.com/ImageMagick/ImageMagick/tree/main/Magick%2B%2B>, 2022. (Consultado a 14/09/2022). [Citado na página 39]

- [52] ImageMagick, “ImageMagick.” Disponível em <https://github.com/ImageMagick/ImageMagick>, 2022. (Consultado a 14/09/2022). [Citado na página 39]
- [53] U=RI solve, “Circuit Editor.” Disponível em <https://github.com/urisolve/circuiteditor>, 2022. (Consultado a 19/10/2022). [Citado na página 42]
- [54] Microsoft Corporation, “TypeScript is JavaScript with syntax for types.” Disponível em <https://www.typescriptlang.org/>, 2022. (Consultado a 15/09/2022). [Citado na página 42]
- [55] Docker Inc., “Develop faster. Run anywhere..” Disponível em <https://www.docker.com/>, 2022. (Consultado a 20/09/2022). [Citado nas páginas 49 e 127]
- [56] Dimitri van Heesch, “Doxygen.” Disponível em <https://doxygen.nl/index.html>, Aug. 2022. (Consultado a 06/10/2022). [Citado na página 49]
- [57] M. Bancila, “stduuid: A C++17 cross-platform implementation for UUIDs.” Disponível em <https://github.com/mariusbancila/stduuid>, 2022. (Consultado a 19/09/2022). [Citado nas páginas 83 e 127]
- [58] N. Lohmann, “JSON for Modern C++.” Disponível em <https://github.com/nlohmann/json>, 2022. (Consultado a 19/09/2022). [Citado nas páginas 83 e 127]
- [59] Google, “GoogleTest - Google Testing and Mocking Framework.” Disponível em <https://github.com/google/googletest>, 2022. (Consultado a 19/09/2022). [Citado nas páginas 85 e 126]
- [60] Linux Test Project, “LCOV.” Disponível em <https://github.com/linux-test-project/lcov>, 2022. (Consultado a 06/10/2022). [Citado na página 85]
- [61] CMake, “Build with CMake. Build with Confidence..” Disponível em <https://cmake.org/>. (Consultado a 17/09/2022). [Citado na página 126]
- [62] CMake, “Installing CMake.” Disponível em <https://cmake.org/install/>. (Consultado a 17/09/2022). [Citado na página 126]
- [63] OpenCV documentation, “OpenCV installation overview.” Disponível em https://docs.opencv.org/4.6.0/d0/d3d/tutorial_general_install.html, June 2022. (Consultado a 19/09/2022). [Citado na página 126]
- [64] Docker Inc., “Use containers to Build, Share and Run your applications.” Disponível em <https://www.docker.com/resources/what-container/>, 2022. (Consultado a 20/09/2022). [Citado na página 127]

-
- [65] CMake documentation, “CMAKE_BUILD_TYPE.” Disponível em https://cmake.org/cmake/help/latest/variable/CMAKE_BUILD_TYPE.html, 2022. (Consultado a 06/10/2022). [Citado na página 128]
- [66] CMake documentation, “CMAKE_CONFIGURATION_TYPES.” Disponível em https://cmake.org/cmake/help/latest/variable/CMAKE_CONFIGURATION_TYPES.html, 2022. (Consultado a 06/10/2022). [Citado na página 128]

Anexo A

Exemplo de mapa de segmentação

Este anexo apresenta um exemplo de mapa de segmentação gerado pelo *software*, referente ao circuito-exemplo utilizado na Secção 4.2. Este mapa está também disponível no diretório do *software* do projeto em *docs/results/circuit-2/assets*, no ficheiro “segmentation_map.json”.

```
1 {
2     "components": [
3         {
4             "id": "0fc564-ee6f-4b8b-ad96-5c59cd4bffb3",
5             "type": "",
6             "fullName": "",
7             "label": {
8                 "id": "91a8bb39-f122-4560-bb80-02a302e90ce8",
9                 "owner": "0fc564-ee6f-4b8b-ad96-5c59cd4bffb3",
10                "name": "",
11                "value": "",
12                "unit": "",
13                "position": {
14                    "x": 0,
15                    "y": 0,
16                    "angle": 0
17                },
18            },
19        },
20    ],
21 }
```

```

18         "isNameHidden": true,
19         "isValueHidden": true
20     },
21     "ports": [
22         {
23             "id": "c6724690-497d-4518-9dd2-855342a85615",
24             "owner": "0fc564-ee6f-4b8b-ad96-5c59cd4bffbcb",
25             "type": "hybrid",
26             "position": {
27                 "x": 0.4,
28                 "y": 0.0,
29                 "angle": 0
30             },
31             "connection": "0e27769a-d84e-4808-935d-6dae427654e5"
32         }
33     ],
34     "position": {
35         "x": 256,
36         "y": 343,
37         "angle": 0
38     }
39 },
40 {
41     "id": "19c61ed6-1a49-4c78-891d-bc10db407e9f",
42     "type": "",
43     "fullName": "",
44     "label": {
45         "id": "711ecd98-ca70-426a-b9a4-aaed3fc20c1e",
46         "owner": "19c61ed6-1a49-4c78-891d-bc10db407e9f",
47         "name": "",
48         "value": "",
49         "unit": "",
50         "position": {
51             "x": 471,
52             "y": 155,
53             "angle": 0
54         },
55         "isNameHidden": true,
56         "isValueHidden": true
57     },
58     "ports": [
59         {
60             "id": "0c40ac2b-ccc8-44ba-97c4-aa039134a49e",
61             "owner": "19c61ed6-1a49-4c78-891d-bc10db407e9f",
62             "type": "hybrid",
63             "position": {

```

```

64         "x": 0.5,
65         "y": 1.0,
66         "angle": 0
67     },
68     "connection": "4cdab47d-a5fc-4f1b-80be-5705
        bde6e4c5"
69 },
70 {
71     "id": "871969fe-1cfa-4502-a7e7-5960af667e56",
72     "owner": "19c61ed6-1a49-4c78-891d-bc10db407e9f",
73     "type": "hybrid",
74     "position": {
75         "x": 0.5,
76         "y": 0.0,
77         "angle": 0
78     },
79     "connection": "9b5f98a7-5851-469d-8645-294
        a2acbee7a"
80 }
81 ],
82 "position": {
83     "x": 384,
84     "y": 173,
85     "angle": 0
86 }
87 },
88 {
89     "id": "a0535f49-8386-4dca-8f7d-6e09c0c3eb9e",
90     "type": "",
91     "fullName": "",
92     "label": {
93         "id": "bba774f8-4e34-4db7-8b7f-080d887150f1",
94         "owner": "a0535f49-8386-4dca-8f7d-6e09c0c3eb9e",
95         "name": "",
96         "value": "",
97         "unit": "",
98         "position": {
99             "x": 15,
100             "y": 166,
101             "angle": 0
102         },
103         "isNameHidden": true,
104         "isValueHidden": true
105     },
106     "ports": [
107         {
108             "id": "99302f6e-e3c6-414e-ab76-cfc925a25ba5",

```

```

109         "owner": "a0535f49-8386-4dca-8f7d-6e09c0c3eb9e
110         ",
111         "type": "hybrid",
112         "position": {
113             "x": 0.4,
114             "y": 1.0,
115             "angle": 0
116         },
117         "connection": "47411d3c-36e3-44a5-8a92-
118             fb11263cf2e1"
119     },
120     {
121         "id": "dd5d220d-f836-4044-9a0b-d47fe18b3cee",
122         "owner": "a0535f49-8386-4dca-8f7d-6e09c0c3eb9e
123         ",
124         "type": "hybrid",
125         "position": {
126             "x": 0.5,
127             "y": 0.0,
128             "angle": 0
129         },
130         "connection": "4c9887ac-67ef-411c-b5c4-5
131             f47169380e2"
132     }
133 ],
134 "position": {
135     "x": 120,
136     "y": 172,
137     "angle": 0
138 }
139 },
140 {
141     "id": "a8ccc791-b529-4bc3-bd61-4c87c769af2d",
142     "type": "",
143     "fullName": "",
144     "label": {
145         "id": "56187dfa-aa02-4b5a-b7be-66f6fb69088f",
146         "owner": "a8ccc791-b529-4bc3-bd61-4c87c769af2d",
147         "name": "",
148         "value": "",
149         "unit": "",
150         "position": {
151             "x": 242,
152             "y": 21,
153             "angle": 0
154         },
155     },
156     "isNameHidden": true,
157     "isValueHidden": true
158 },

```

```

154         "ports": [
155             {
156                 "id": "29d080b8-6a4e-4825-afdc-ddd966151b7a",
157                 "owner": "a8ccc791-b529-4bc3-bd61-4c87c769af2d",
158                 "type": "hybrid",
159                 "position": {
160                     "x": 0.0,
161                     "y": 0.5,
162                     "angle": 0
163                 },
164                 "connection": "4c9887ac-67ef-411c-b5c4-5f47169380e2"
165             },
166             {
167                 "id": "67e97f70-85d1-494c-aa9f-dae3628497f3",
168                 "owner": "a8ccc791-b529-4bc3-bd61-4c87c769af2d",
169                 "type": "hybrid",
170                 "position": {
171                     "x": 1.0,
172                     "y": 0.5,
173                     "angle": 0
174                 },
175                 "connection": "9b5f98a7-5851-469d-8645-294a2acbee7a"
176             }
177         ],
178         "position": {
179             "x": 242,
180             "y": 65,
181             "angle": 0
182         }
183     }
184 ],
185     "nodes": [
186         {
187             "id": "37a185f5-81f4-4d20-8421-af89a66adc72",
188             "label": {
189                 "id": "c1274439-57cb-4258-94fb-75921a4d68d1",
190                 "owner": "37a185f5-81f4-4d20-8421-af89a66adc72",
191                 "name": "",
192                 "value": "",
193                 "unit": "",
194                 "position": {
195                     "x": 0,
196                     "y": 0,
197                     "angle": 0
198                 },
199             },

```



```

199         "isNameHidden": true,
200         "isValueHidden": true
201     },
202     "position": {
203         "x": 281,
204         "y": 278,
205         "angle": 0
206     },
207     "connections": [
208         "0e27769a-d84e-4808-935d-6dae427654e5",
209         "4cdab47d-a5fc-4f1b-80be-5705bde6e4c5",
210         "47411d3c-36e3-44a5-8a92-fb11263cf2e1"
211     ],
212     "type": "real"
213 }
214 ],
215 "connections": [
216     {
217         "id": "0e27769a-d84e-4808-935d-6dae427654e5",
218         "start": "c6724690-497d-4518-9dd2-855342a85615",
219         "end": "37a185f5-81f4-4d20-8421-af89a66adc72",
220         "label": {
221             "id": "f64a8aa7-af9a-435d-8eee-deb0aa318fcd",
222             "owner": "0e27769a-d84e-4808-935d-6dae427654e5",
223             "name": "",
224             "value": "",
225             "unit": "",
226             "position": {
227                 "x": 0,
228                 "y": 0,
229                 "angle": 0
230             },
231             "isNameHidden": true,
232             "isValueHidden": true
233         }
234     },
235     {
236         "id": "4cdab47d-a5fc-4f1b-80be-5705bde6e4c5",
237         "start": "0c40ac2b-ccc8-44ba-97c4-aa039134a49e",
238         "end": "37a185f5-81f4-4d20-8421-af89a66adc72",
239         "label": {
240             "id": "720ff297-f1b6-4c9c-ac1e-e52ca2af71d4",
241             "owner": "4cdab47d-a5fc-4f1b-80be-5705bde6e4c5",
242             "name": "",
243             "value": "",
244             "unit": "",
245             "position": {
246                 "x": 0,
247                 "y": 0,

```

```

248         "angle": 0
249     },
250     "isNameHidden": true,
251     "isValueHidden": true
252 }
253 },
254 {
255     "id": "47411d3c-36e3-44a5-8a92-fb11263cf2e1",
256     "start": "99302f6e-e3c6-414e-ab76-cfc925a25ba5",
257     "end": "37a185f5-81f4-4d20-8421-af89a66adc72",
258     "label": {
259         "id": "69a56f63-caed-4da8-a7fe-e35b57e72a59",
260         "owner": "47411d3c-36e3-44a5-8a92-fb11263cf2e1",
261         "name": "",
262         "value": "",
263         "unit": "",
264         "position": {
265             "x": 0,
266             "y": 0,
267             "angle": 0
268         },
269         "isNameHidden": true,
270         "isValueHidden": true
271     }
272 },
273 {
274     "id": "4c9887ac-67ef-411c-b5c4-5f47169380e2",
275     "start": "dd5d220d-f836-4044-9a0b-d47fe18b3cee",
276     "end": "29d080b8-6a4e-4825-afdc-ddd966151b7a",
277     "label": {
278         "id": "ce5710fc-849e-44e8-8efd-1d7826e08726",
279         "owner": "4c9887ac-67ef-411c-b5c4-5f47169380e2",
280         "name": "",
281         "value": "",
282         "unit": "",
283         "position": {
284             "x": 0,
285             "y": 0,
286             "angle": 0
287         },
288         "isNameHidden": true,
289         "isValueHidden": true
290     }
291 },
292 {
293     "id": "9b5f98a7-5851-469d-8645-294a2acbee7a",
294     "start": "871969fe-1cfa-4502-a7e7-5960af667e56",
295     "end": "67e97f70-85d1-494c-aa9f-dae3628497f3",
296     "label": {

```

```
297         "id": "6838bdb1-125f-4dca-870b-376438415930",
298         "owner": "9b5f98a7-5851-469d-8645-294a2acbee7a",
299         "name": "",
300         "value": "",
301         "unit": "",
302         "position": {
303             "x": 0,
304             "y": 0,
305             "angle": 0
306         },
307         "isNameHidden": true,
308         "isValueHidden": true
309     }
310 }
311 ]
312 }
```

Listagem A.1: Mapa de segmentação resultante para o circuito-exemplo.

Anexo B

Guia de utilização do *software*

Este anexo contém um guia das instruções para instalação, compilação e execução do módulo de *software* desenvolvido neste projeto.

B.1 Pré-requisitos

Este módulo de *software* necessita de algumas ferramentas externas para ser compilado com sucesso, podendo o utilizador optar por uma de duas opções: instalar localmente estas ferramentas, ou utilizar o ficheiro de Docker disponibilizado neste projeto que permite gerar uma imagem de um *container* que já inclui todas as dependências requeridas pelo *software*.

B.1.1 Instalação local

As ferramentas que têm de estar instaladas na máquina são as seguintes:

- **CMake**: para gestão do processo de compilação;
- **Compilador C++**: para compilação do *software*;
- **OpenCV**: biblioteca externa de *software* para acesso a algoritmos de visão computacional.

CMake é uma ferramenta multi-plataforma, *open source* e extensível para compilação, teste e distribuição de *software*. Esta ferramenta é muito usada para gestão

do processo de compilação de *software*, utilizando ficheiros de configuração independentes do compilador, e gera *makefiles* nativos e *workspaces* (por exemplo, Unix Makefiles e projetos de Visual Studio, respetivamente) que podem ser usados com o compilador pretendido [61]. A instalação desta ferramenta pode ser feita através do *download* de um binário pré-compilado para o sistema operativo da máquina local (Windows, MacOS ou Linux). Alternativamente, pode também ser feito o *download* do seu código-fonte e compilar CMake a partir deste [62]. A versão de CMake utilizada durante o desenvolvimento deste projeto foi a 3.22.2.

Quanto ao compilador C++, durante o desenvolvimento e teste deste módulo de *software*, foram utilizados os seguintes compiladores, combinados com os respetivos sistemas operativos descritos:

- Microsoft Visual C++ 2022 / Build Tools 19.32.31332, em Windows 10;
- GCC 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1), em Ubuntu 20.04.4 LTS (a partir da imagem Docker).

Assim, é conhecido que este projeto suporta pelo menos esses compiladores, contudo poderá ser compilado com sucesso recorrendo a outros.

Relativamente à biblioteca OpenCV, a sua instalação na máquina pode ser realizada de duas formas: *download* de versões pré-compiladas ou compilar a partir do seu código-fonte. Habitualmente, as versões pré-compiladas de OpenCV são suficientes para responder às necessidades do utilizador. Estas são criadas pela equipa de programadores da biblioteca, providenciando *packages* compilados com parâmetros *default* para Windows, Android e iOS, publicados nas *releases* no seu repositório GitHub ou na plataforma SourceForge. Adicionalmente, também há organizações que mantêm uma distribuição binária própria de OpenCV, estando disponível, por exemplo, em *system packages* de distribuições populares de Linux. Porém, poderá haver situações em que os pacotes existentes não são aplicáveis, consequentemente terá de se realizar um processo de compilação a partir do seu código-fonte [63]. OpenCV 4.5.5 foi a versão utilizada durante a implementação deste *software*.

Para além de OpenCV, o projeto requer outras bibliotecas de *software* externas para a concretização de algumas funcionalidades. No entanto, estas não precisam de ser manualmente instaladas na máquina, visto que são automaticamente *fetched* durante a configuração de CMake, incluindo os seguintes pacotes de *software*:

- **GoogleTest** - *Google Testing and Mocking Framework* [59]:
 - Biblioteca usada para testes unitários e para *mocking* de classes;
 - Só é automaticamente *fetched* se a opção de configuração *BUILD_TESTS* estiver definida como *ON* (opções de compilação são apresentadas na Secção B.2);

- Versão utilizada: 1.11.0.
- **stduuid** - uma implementação C++17 multiplataforma para UUID [57]:
 - Biblioteca usada para geração de UUID para os *IDs* dos elementos do circuito no mapa de segmentação;
 - Versão utilizada: *commit hash*
"3afe7193facd5d674de709fcc44d5055e144d7a" no Git.
- **"JSON for Modern C++"** - biblioteca JSON C++ bastante popular [58]:
 - Biblioteca usada para geração do mapa de segmentação em JSON;
 - Versão utilizada: 3.11.2.

B.1.2 Utilização da plataforma Docker

Docker é uma plataforma muito popular atualmente para desenvolvimento, distribuição, execução e teste de aplicações, baseada em unidades de *software* de empacotamento designadas por *containers*. A imagem de um *container* é um pacote executável de *software* que inclui tudo o que é necessário para correr uma aplicação, que vai desde o código, ambiente de execução, ferramentas e bibliotecas do sistema, até a configurações requeridas. Esta imagem torna-se um *container* no momento em que é executada, criando um processo independente na máquina onde está a correr. Assim, um *container* possibilita o isolamento do *software* do seu ambiente de desenvolvimento, permitindo a sua execução, bem como a sua distribuição, para um outro ambiente pretendido [55, 64].

Este projeto disponibiliza um ficheiro, em *docker/Dockerfile*, para a geração de uma imagem de um *container* de Docker, que inclui todas as dependências requeridas pelo *software*. O *Dockerfile* é simplesmente um *script* com instruções que são utilizadas na criação da imagem do *container*, baseado numa imagem Ubuntu 20.04 no caso deste projeto. O procedimento a realizar para a criação e execução desta imagem é o seguinte:

1. Compilar a imagem usando os comandos descritos na Listagem B.1 a partir de uma linha de comandos (gera uma imagem com o nome *circuitsegmentation*):

```
1 $ cd <project-directory>
2 $ cd docker
3 $ docker build . -t circuitsegmentation
```

Listagem B.1: Comandos para compilação da imagem Docker.

2. Correr a imagem gerada com a criação de um *container* e montagem do diretório deste projeto nesse mesmo *container*, possibilitando assim o acesso ao *software* a partir deste. A Listagem B.2 contém o comando a utilizar neste passo, onde “project-directory” corresponde ao caminho absoluto do diretório do projeto, e */src* é o diretório no qual irá ser montado o projeto (note-se que pode ter outro nome, não está relacionado com o diretório *src* do projeto). Por exemplo, se o *software* se encontra no diretório *c:/Projects/circuit-segmentation*, então “project-directory” deve ser substituído por este caminho no comando apresentado.

```
1 $ docker run -it --name mycontainer -v <project-directory>:/src circuitsegmentation
```

Listagem B.2: Comando para execução do *container* Docker.

O projeto pode agora ser compilado e executado a partir do *container*, sem a instalação manual na máquina das ferramentas descritas na Secção B.1.1. Todavia, nesta opção é necessário a instalação da aplicação Docker no sistema local.

B.2 Compilação e execução do *software*

O *software* deste projeto, tal como mencionado anteriormente, utiliza a ferramenta CMake para gestão do processo de compilação. O CMake aplica as instruções dadas no ficheiro *CMakeLists.txt*, localizado no topo do diretório do *software*. Entre as instruções presentes neste ficheiro, realçam-se a definição de quais os diretórios que devem ser incluídos para a compilação do projeto, assim como a verificação e o *fetch* das bibliotecas requeridas pelo *software*. Além disso, este ficheiro também declara as seguintes opções de configuração de CMake:

- *CMAKE_BUILD_TYPE*: define o tipo de *build* para geradores de *single-configuration* (por exemplo, os geradores de *Makefiles* ou o sistema Ninja). Valores típicos incluem *Debug*, *Release*, *RelWithDebInfo* e *MinSizeRel*. O seu valor *default* é *Debug* [65];
- *CMAKE_CONFIGURATION_TYPES*: seleciona o tipo de *build* para geradores de *multi-configuration* (por exemplo, Visual Studio, Xcode ou Ninja *Multi-Config*). Valores típicos incluem os referidos para *CMAKE_BUILD_TYPE*. O seu valor *default* é *Debug* [66];
- *BUILD_TESTS*: permite a adição ao processo de compilação dos testes unitários. O seu valor *default* é *OFF*;

- *BUILD_COVERAGE*: possibilita a compilação com informações de cobertura de código, devendo ser usada em conjunto com a *BUILD_TESTS* (só para GCC). O seu valor *default* é *OFF*.

Os comandos descritos na Listagem B.3 podem ser utilizados para configuração CMake do projeto (exemplo para configuração *Debug*), onde “project-directory” é o caminho do diretório do projeto, e “.” no comando CMake equivale ao diretório de nível superior que contém o ficheiro *CMakeLists.txt* a usar para configuração. Após a execução destes comandos, é criada uma pasta *build-debug* (pode ter qualquer nome) que contém ficheiros gerados pelo CMake, que serão usados no passo de compilação.

```
1 $ cd <project-directory>
2 $ mkdir build-debug
3 $ cd build-debug
4 $ cmake .. -DCMAKE_BUILD_TYPE=Debug
```

Listagem B.3: Comandos para configuração CMake.

A compilação do *software* pode ser efetuada aplicando o comando *build* de CMake (comando universal para compilação que abstrai a interface de uma ferramenta nativa de *build*) contido na Listagem B.4, onde “.” corresponde ao diretório atual e a opção “-j” define o número máximo de processos paralelos a usar durante a compilação.

```
1 $ cmake --build . -j 4
```

Listagem B.4: Comando para compilação do *software* usando *cmake*.

Por outro lado, é também possível compilar o projeto invocando diretamente a ferramenta de *build*, exemplificado na Listagem B.5 com o comando a usar para Unix Makefiles (“-j” tem a mesma funcionalidade referida previamente para o comando CMake).

```
1 $ make -j 4
```

Listagem B.5: Comando para compilação do *software* usando *make*.

Após a compilação do *software*, um ficheiro executável é criado na pasta *build-debug* e pode ser corrido pela linha de comandos, com a estrutura de comando demonstrada na Listagem B.6. A opção “-i” (ou “--image”) é obrigatória e deve ser providenciada pelo utilizador para que o programa tenha conhecimento da localização do ficheiro de imagem do circuito (“image_path”). Este caminho pode ser relativo, isto é, não necessita de ser fornecido o caminho absoluto da imagem (por exemplo, se o ficheiro de imagem com o nome “my-circuit.png” está localizado no diretório atual, pode ser passado apenas “-i my-circuit.png”). O campo “OPTIONS” corresponde às restantes opções disponíveis para execução do programa, referenciadas na Secção 4.2.1, sendo estas opcionais. Note-se que o ficheiro executável pode estar localizado num diretório distinto do referido no comando, pois depende do gerador de configuração utilizado (para Visual Studio, normalmente está no diretório mencionado no comando, enquanto que os geradores de *Makefiles* não criam o diretório de *Debug*, localizando-se assim em *./src* na pasta *build-debug*).

```
1 $ ./src/Debug/CircuitSegmentation -i <image_path> [OPTIONS]
```

Listagem B.6: Comando para execução do *software*.

Para correr os testes unitários do projeto, podem ser usados os comandos descritos na Listagem B.7, que incluem a configuração CMake, compilação do projeto e o comando *ctest* que executa o conjunto de testes unitários implementados. De salientar que é necessário colocar a opção de configuração CMake *BUILD_TESTS* com valor *ON*, para que a pasta *tests* seja incluída no processo de compilação do *software*.

```
1 $ cd <project-directory>
2 $ mkdir build-debug
3 $ cd build-debug
4 $ cmake .. -DCMAKE_BUILD_TYPE=Debug -DBUILD_TESTS=ON
5 $ cmake --build . -j 4
6 $ ctest
```

Listagem B.7: Comandos para compilação e execução dos testes unitários.

Anexo C

Ficheiros de imagens de processamento

Este anexo descreve os ficheiros das imagens de processamento gerados pelo módulo de *software* durante a sua execução, para as diferentes etapas de processamento da imagem do circuito elétrico. Note-se que estes ficheiros são criados no diretório onde o programa iniciou a sua execução, e somente se o utilizador providenciar a opção “-s” ou “--save-proc” pela linha de comandos. Os ficheiros gerados durante o processamento ativados por esta opção começam sempre por “cs_”, que simboliza *circuit segmentation*.

Tabela C.1: Imagens geradas durante a receção de imagem.

Passo	Ficheiro gerado
Receção da imagem	cs_initial_image.png

Tabela C.2: Imagens geradas durante o pré-processamento.

Passo	Ficheiro gerado
Conversão para escala de cinza	cs_preproc_grayscale.png
Filtragem da imagem	cs_preproc_blur.png
<i>Thresholding</i> adaptativo	cs_preproc_threshold.png
Dilatação	cs_preproc_morph_dilation.png
Esqueletização da imagem	cs_preproc_thinning.png

Tabela C.3: Imagens geradas durante a deteção de conexões.

Passo	Ficheiro gerado
<i>Closing</i>	cs_segment_connections_morph_close.png
<i>Opening</i>	cs_segment_connections_morph_open.png
Interseção	cs_segment_connections_intersection.png
Remoção de elementos	cs_segment_connections_only_conn.png
Deteção de conexões	cs_segment_connections_detected.png

Tabela C.4: Imagens geradas durante a deteção de componentes.

Passo	Ficheiro gerado
Remoção de conexões	cs_segment_components_remove_connections.png
<i>Closing</i>	cs_segment_components_morph_close.png
Deteção de componentes	cs_segment_components_detected.png

Tabela C.5: Imagens geradas durante a atualização de conexões.

Passo	Ficheiro gerado
Remoção de componentes	cs_segment_connections_remove_components.png
Atualização de conexões	cs_segment_connections_updated.png

Tabela C.6: Imagens geradas durante a deteção de nós e atualização de conexões.

Passo	Ficheiro gerado
Deteção de nós e de conexões	cs_segment_nodes_detected_connections_updated.png

Tabela C.7: Imagens geradas durante a deteção de *ports* de componentes.

Passo	Ficheiro gerado
Deteção de <i>ports</i>	cs_segment_components_ports_detected.png

Tabela C.8: Imagens geradas durante a deteção de *labels*.

Passo	Ficheiro gerado
Remoção de elementos	cs_segment_labels_remove_elements.png
<i>Closing</i>	cs_segment_labels_morph_close.png
<i>Opening</i>	cs_segment_labels_morph_open.png
Deteção de <i>labels</i>	cs_segment_labels_detected.png

Tabela C.9: Imagens geradas durante a associação de *labels*.

Passo	Ficheiro gerado
Geração de <i>bounding boxes</i>	cs_segment_labels_associate_boxes_connections_nodes.png

Anexo D

Mensagens de registo

Este anexo demonstra as mensagens registadas pelo módulo de *software* durante a sua execução, para as diferentes etapas de processamento da imagem do circuito elétrico, referentes ao circuito-exemplo utilizado na Secção 4.2. Note-se que estas mensagens apenas são geradas se o utilizador providenciar a opção “-V” ou “--verbose” pela linha de comandos quando inicia a execução do programa.

```
[2022-10-01 22:24:15][INFO] Starting Circuit Segmentation and Modeling: version 1.0.0
[2022-10-01 22:24:15][INFO] Starting image processing
[2022-10-01 22:24:15][INFO] Image file path: circuit-1.jpg
[2022-10-01 22:24:15][INFO] Image received successfully
[2022-10-01 22:24:15][INFO] Starting image preprocessing
```

(a)

```
[2022-10-01 22:28:35][INFO] Starting Circuit Segmentation and Modeling: version 1.0.0
[2022-10-01 22:28:35][INFO] Starting image processing
[ WARN:0@0.040] global C:\build\master_winpack-build-win64-vc14\opencv\modules\imgcodecs\src\
loadsave.cpp (239) cv::findDecoder imread_('circuit-2.jpg'): can't open/read file: check file
path/integrity
[2022-10-01 22:28:35][WARNING] Image cannot be open/read with path: circuit-2.jpg
[2022-10-01 22:28:35][ERROR] Failed during image reception
[2022-10-01 22:28:35][INFO] Ending Circuit Segmentation and Modeling: version 1.0.0
```

(b)

Figura D.1: Mensagens geradas quando a receção da imagem ocorre com (a) sucesso ou (b) insucesso.

```
[2022-10-01 19:07:48][INFO] Starting image preprocessing
[2022-10-01 19:07:48][INFO] Image converted to grayscale
[2022-10-01 19:07:48][INFO] Gaussian blurring applied to the image
[2022-10-01 19:07:49][INFO] Adaptive threshold applied to the image
[2022-10-01 19:07:49][INFO] Morphological dilation applied to the image
[2022-10-01 19:07:50][INFO] Thinning operation applied to the image
[2022-10-01 19:07:50][INFO] Image preprocessing occurred successfully
```

Figura D.2: Mensagens geradas durante o pré-processamento.

```
[2022-10-01 19:07:50][INFO] Starting image segmentation
[2022-10-01 19:07:50][INFO] Detecting connections of the circuit
[2022-10-01 19:07:50][INFO] Morphological closing applied to the image
[2022-10-01 19:07:50][INFO] Morphological opening applied to the image
[2022-10-01 19:07:50][INFO] Intersection between the preprocessed image and the image without connections
[2022-10-01 19:07:50][DEBUG] Contours found in the intersection image: 27
[2022-10-01 19:07:50][INFO] Generated image with only the circuit connections
[2022-10-01 19:07:50][DEBUG] Contours found in the image, to detect connections: 12
[2022-10-01 19:07:50][INFO] Connections found in the circuit: 5
```

Figura D.3: Mensagens geradas durante a detecção de conexões.

```
[2022-10-01 19:07:50][INFO] Detecting components
[2022-10-01 19:07:51][INFO] Morphological closing applied to the image
[2022-10-01 19:07:51][DEBUG] Contours found in the image, to detect components: 9
[2022-10-01 19:07:51][DEBUG] Intersection point between contour and connection at {277, 343}
[2022-10-01 19:07:51][DEBUG] Intersection point between contour and connection at {413, 213}
[2022-10-01 19:07:51][DEBUG] Intersection point between contour and connection at {148, 222}
[2022-10-01 19:07:51][DEBUG] Intersection point between contour and connection at {327, 87}
[2022-10-01 19:07:51][INFO] Components found in the circuit: 4
```

Figura D.4: Mensagens geradas durante a detecção de componentes.

```
[2022-10-01 23:36:27][INFO] Updating connections of the circuit
[2022-10-01 23:36:27][DEBUG] Contours found in the image, to update connections: 20
[2022-10-01 23:36:27][INFO] Connections found in the circuit: 15
```

Figura D.5: Mensagens geradas durante a atualização de conexões.

```
[2022-10-02 11:18:12][INFO] Detecting nodes and update connections
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {277, 342}
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {413, 214}
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {148, 223}
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 3
[2022-10-02 11:18:12][DEBUG] Node position = {281, 278}
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {149, 171}
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {241, 86}
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 2
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {411, 172}
[2022-10-02 11:18:12][DEBUG] Intersection point between component and connection at {328, 87}
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 2
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][DEBUG] Number of intersection points for this connection = 0
[2022-10-02 11:18:12][INFO] Connections detected in the circuit: 5
[2022-10-02 11:18:12][INFO] Nodes detected in the circuit: 1
```

Figura D.6: Mensagens geradas durante a detecção de nós e atualização de conexões.

```
[2022-10-02 18:14:11][INFO] Detecting connection points (ports) of components
[2022-10-02 18:14:11][DEBUG] Checking component with ID 822c6d6e-2df1-4a49-b352-3e2d10612b27
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {277, 342}
[2022-10-02 18:14:11][DEBUG] Port position at {0.420000, 0.000000}
[2022-10-02 18:14:11][DEBUG] Checking component with ID 9fcea773-1ef4-4336-8519-d61d637253c2
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {413, 214}
[2022-10-02 18:14:11][DEBUG] Port position at {0.483333, 1.000000}
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {411, 172}
[2022-10-02 18:14:11][DEBUG] Port position at {0.450000, 0.000000}
[2022-10-02 18:14:11][DEBUG] Checking component with ID 4168bc2f-a04c-4869-a24c-5d358bb42aec
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {148, 223}
[2022-10-02 18:14:11][DEBUG] Port position at {0.444444, 1.000000}
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {149, 171}
[2022-10-02 18:14:11][DEBUG] Port position at {0.460317, 0.000000}
[2022-10-02 18:14:11][DEBUG] Checking component with ID 2e364825-2c5b-42c6-8b10-89b86ca9fbbba
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {241, 86}
[2022-10-02 18:14:11][DEBUG] Port position at {0.000000, 0.488372}
[2022-10-02 18:14:11][DEBUG] Component connected to a connection wire at point {328, 87}
[2022-10-02 18:14:11][DEBUG] Port position at {1.000000, 0.511628}
```

Figura D.7: Mensagens geradas durante a deteção de *ports* de componentes.

```
[2022-10-02 18:14:11][INFO] Updating list of detected components
[2022-10-02 18:14:11][INFO] Detected components in the image: 4
```

Figura D.8: Mensagens geradas durante a atualização de componentes detetados.

```
[2022-10-05 11:20:39][INFO] Detecting labels
[2022-10-05 11:20:40][INFO] Morphological closing applied to the image
[2022-10-05 11:20:40][INFO] Morphological opening applied to the image
[2022-10-05 11:20:40][DEBUG] Contours found in the image, to detect labels: 3
[2022-10-05 11:20:40][INFO] Labels found in the circuit: 3
```

Figura D.9: Mensagens geradas durante a deteção de *labels*.

```
[2022-10-05 11:20:40][INFO] Associating labels to the circuit elements
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 8e553b1e-c9a4-42cc-867d-ce935839cc76 and component 16a8d329-e72a-41f8-b047-932518518e4c = 16.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 8e553b1e-c9a4-42cc-867d-ce935839cc76 and connection 4fe78987-a230-44b3-9be3-f35f56f526ab = 43.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 8e553b1e-c9a4-42cc-867d-ce935839cc76 and node ac0c853b-a9ed-4adb-b1fd-ac1b8c12b314 = 171.049700
[2022-10-05 11:20:40][DEBUG] Label 8e553b1e-c9a4-42cc-867d-ce935839cc76 is associated to the component 16a8d329-e72a-41f8-b047-932518518e4c
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 87a84acb-b886-4355-b6a4-875b67b4ae26 and component 32b39c33-95a0-47f9-adb3-27a578ad1d28 = 27.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 87a84acb-b886-4355-b6a4-875b67b4ae26 and connection 7f2b05e2-77bd-4b05-b5fb-b182cac6b0cd = 57.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 87a84acb-b886-4355-b6a4-875b67b4ae26 and node ac0c853b-a9ed-4adb-b1fd-ac1b8c12b314 = 193.313217
[2022-10-05 11:20:40][DEBUG] Label 87a84acb-b886-4355-b6a4-875b67b4ae26 is associated to the component 32b39c33-95a0-47f9-adb3-27a578ad1d28
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 6e580f0f-5f57-46df-94e1-9b2c0c64d4a1 and component caefc422-9505-4a82-83dd-7c7072ad15b3 = 18.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 6e580f0f-5f57-46df-94e1-9b2c0c64d4a1 and connection 7f2b05e2-77bd-4b05-b5fb-b182cac6b0cd = 37.000000
[2022-10-05 11:20:40][DEBUG] Minimum distance between label 6e580f0f-5f57-46df-94e1-9b2c0c64d4a1 and node ac0c853b-a9ed-4adb-b1fd-ac1b8c12b314 = 221.000000
[2022-10-05 11:20:40][DEBUG] Label 6e580f0f-5f57-46df-94e1-9b2c0c64d4a1 is associated to the component caefc422-9505-4a82-83dd-7c7072ad15b3
[2022-10-05 11:20:40][INFO] Image segmentation occurred successfully
```

Figura D.10: Mensagens geradas durante a associação de *labels*.

```
[2022-10-05 11:20:40][INFO] Generating images with ROI for components
[2022-10-05 11:20:40][INFO] Generating images with ROI for labels
[2022-10-05 11:20:40][INFO] Generation of images with ROI occurred successfully
```

Figura D.11: Mensagens geradas durante a criação de imagens ROI.

```
[2022-10-05 11:20:40][INFO] Generating segmentation map
[2022-10-05 11:20:40][INFO] Writing segmentation map JSON file
[2022-10-05 11:20:40][INFO] Generation of segmentation map file occurred successfully
[2022-10-05 11:20:40][INFO] Ending Circuit Segmentation and Modeling: version 1.0.0
```

Figura D.12: Mensagens geradas durante a criação do mapa de segmentação.