



Universidade Estadual de Campinas  
Faculdade de Engenharia Elétrica e Computação  
Departamento de Engenharia da Computação  
e Automação Industrial

## **Análise de Algoritmos da Transformada *Watershed***

Autor: André Körbes

Orientador: Roberto de Alencar Lotufo

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como  
parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de  
concentração: Engenharia de Computação.

Comissão Examinadora  
Nina Sumiko Tomita Hirata  
Romis Ribeiro de Faissol Attux

Campinas, 23/03/2010

FICHA CATALOGRÁFICA ELABORADA PELA  
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Körbes, André

K841a      Análise de algoritmos da Transformada Watershed  
/ André Körbes. – Campinas, SP: [s.n.], 2010.

Orientador: Roberto de Alencar Lotufo.

Dissertação de Mestrado - Universidade Estadual  
de Campinas, Faculdade de Engenharia Elétrica e de  
Computação.

1. Algoritmos. 2. Morfologia matemática. 3.  
Processamento de Imagens. I. Lotufo, Roberto de  
Alencar. II. Universidade Estadual de Campinas.  
Faculdade de Engenharia Elétrica e de Computação. III.  
Título

Título em Inglês: Analysis of algorithms of the Watershed Transform

Palavras-chave em Inglês: Algorithms, Mathematical morphology, Image processing

Área de concentração: Engenharia de Computação

Titulação: Mestre em Engenharia Elétrica

Banca Examinadora: Nina Sumiko Tomita Hirata, Romis Ribeiro de Faissol Attux

Data da defesa: 23/03/2010

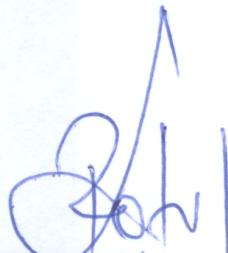
Programa de Pós Graduação: Engenharia Elétrica

## COMISSÃO JULGADORA - TESE DE MESTRADO

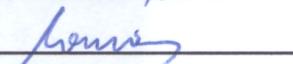
**Candidato:** André Körbes

**Data da Defesa:** 23 de março de 2010

**Título da Tese:** "Análise de Algoritmos da Transformada Watershed"

Prof. Dr. Roberto de Alencar Lotufo (Presidente): 

Profa. Dra. Nina Sumiko Tomita Hirata: 

Prof. Dr. Romis Ribeiro de Faissol Attux: 



# Resumo

A transformada watershed é uma técnica morfológica de segmentação de imagens inspirada na divisão de superfícies em bacias hidrográficas, tendo diversas formas de definição e de algoritmos. Este trabalho realiza uma análise sistemática da literatura de catorze destes algoritmos. Foram consideradas as principais abordagens existentes desde a introdução do primeiro algoritmo rápido por Vincent e Soille em 1991, até os trabalhos de Cousty *et al.* em 2009. Para melhor compreensão da área, as definições de transformada *watershed* são revisitadas, provendo o conjunto de soluções formais possíveis e esperadas dos algoritmos.

Na análise destes algoritmos é fornecido pseudocódigo com notação uniformizada e uma implementação operacional Python permitindo abstrair detalhes de programação. Além disto, três algoritmos foram corrigidos para melhor aderência a definição e especificação. Também são identificadas propriedades tais como o comportamento de varredura dos *pixels*, uso de estratégias em particular, uso de estruturas de dados, entre outras.

A compilação das informações sobre os algoritmos permitiu generalizá-los e classificá-los baseado em paradigmas clássicos da computação, a saber a busca em largura e em profundidade. Ambos são embasados na ordem de visitação dos *pixels* utilizada, sendo a busca em largura semelhante a simulação de inundação enquanto a busca em profundidade simula gotas de água em uma superfície.

Foram também realizados estudos comparativos entre as definições implementadas pelos algoritmos, entre as estratégias utilizadas para tratamento de problemas comuns, entre o desempenho obtido pelos programas Python, e de paralelismo e abordagens utilizadas neste último caso. Desta forma, produziu-se um panorama geral e atualizado dos algoritmos de transformada *watershed*.

**Palavras-chave:** Transformada *watershed*, Análise de Algoritmos.



# Abstract

The watershed transform is a morphological image segmentation technique inspired on the division of surfaces in catchment basins, with several forms of definition and algorithms. This work accomplishes a survey of the literature on fourteen of these algorithms. The main approaches since the introduction of the first fast algorithm by Vincent and Soille in 1991, until the work of Cousty *et al.* in 2009 has been considered. For better understanding of the subject, the watershed definitions are revisited, providing the set of formal solutions that are possible and expected from the algorithms.

On the analysis of the algorithms it is supplied pseudocode with a uniform notation and a Python operational implementation allowing to abstract programming details. Aside, three algorithms were corrected for better adherence to definition and specification. Also some properties such as the scanning behaviour, use of particular strategies, and use of data structures, among others were identified.

The compilation of information of the algorithms allowed to generalise and classify them based on classic paradigms of computing, namely breadth-first and depth-first search. Both are based on the visiting order of the pixels, with the breadth-first similar to a flooding simulation while the depth-first simulates drops of water on a surface.

Comparative studies between the algorithms' implemented definitions, the strategies used for treatment of common issues, the performance of the Python programs, of parallelism and its approaches are provided. In this way, a broad and updated viewpoint of the algorithms of the watershed transforms has been produced.

**Keywords:** Watershed Transform, Algorithms Analysis.



# **Agradecimentos**

Aos meus pais Patrício e Roswitha e ao meu irmão Daniel, pelo suporte e amor constante, sem o qual este trabalho não seria possível.

À minha namorada Greice, companheira, parceira e amiga, pelo seu amor e paciência.

Ao meu orientador, prof. Roberto Lotufo, pelos conhecimentos transmitidos.

A todos amigos, novos e antigos, pelos momentos de descontração proporcionados.

À CAPES, pelo apoio financeiro.



# Sumário

<b>Lista de Figuras</b>	<b>xiii</b>
<b>Lista de Tabelas</b>	<b>xvii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Motivação . . . . .	2
1.2 Objetivos . . . . .	5
1.3 Organização . . . . .	6
<b>2 Definições e Convenções</b>	<b>9</b>
2.1 Convenções . . . . .	9
2.1.1 Imagens, dados e entradas . . . . .	9
2.1.2 Estruturas de Dados . . . . .	10
2.1.3 Notação de Laços . . . . .	12
2.1.4 Operações de <i>Arrowing</i> . . . . .	13
2.2 Definições de Watershed . . . . .	13
2.2.1 Transformada Watershed . . . . .	14
2.2.2 Imersão (Flooding-WT) . . . . .	15
2.2.3 Distância Topográfica (TD-WT) . . . . .	16
2.2.4 Condição Local (LC-WT) . . . . .	21
2.2.5 Watershed por Transformada Imagem-Floresta (IFT-WT) . . . . .	22
2.2.6 Watershed Cut (WC-WT) . . . . .	25
2.2.7 Relações . . . . .	29
<b>3 Algoritmos de Watershed</b>	<b>31</b>
3.1 Algoritmo Vincent e Soille de Imersão . . . . .	31
3.1.1 Implementação Python . . . . .	36
3.2 Algoritmo Fila de Prioridade Beucher e Meyer . . . . .	38
3.2.1 Implementação Python . . . . .	40
3.3 Algoritmo Dijkstra-Moore de Caminhos Mínimos de Meyer . . . . .	41
3.3.1 Implementação Python . . . . .	43
3.4 Algoritmo Hill Climbing de Meyer . . . . .	44
3.4.1 Implementação Python . . . . .	47
3.5 Algoritmo Berge de Caminhos Mínimos de Meyer . . . . .	48

3.5.1	Implementação Python . . . . .	50
3.6	Algoritmo Componentes Conexos de Bieniek e Moga . . . . .	51
3.6.1	Implementação Python . . . . .	54
3.7	Algoritmo Union-Find de Meijster e Roerdink . . . . .	56
3.7.1	Implementação Python . . . . .	59
3.8	Algoritmo IFT de Lotufo e Falcão . . . . .	61
3.8.1	Implementação Python . . . . .	63
3.9	Algoritmo Código de Corrente de Sun, Yang e Ren . . . . .	64
3.9.1	Implementação Python . . . . .	67
3.10	Algoritmo Zona de Empate de Audigier, Lotufo e Couprie . . . . .	69
3.10.1	Implementação Python . . . . .	71
3.11	Algoritmo Tobogã Invariante a Ordem de Lin <i>et al.</i> . . . . .	72
3.11.1	Implementação Python . . . . .	75
3.12	Algoritmo Imersão Invariante a Ordem de Lin <i>et al.</i> . . . . .	77
3.12.1	Implementação Python . . . . .	80
3.13	Algoritmo Caminhos Mínimos de Osma-Ruiz <i>et al.</i> . . . . .	82
3.13.1	Implementação Python . . . . .	86
3.14	Algoritmo Watershed Cut de Cousty <i>et al.</i> . . . . .	88
3.14.1	Implementação Python . . . . .	91
<b>4</b>	<b>Análise Crítica dos Algoritmos</b>	<b>95</b>
4.1	Análise Comparativa de Resultados . . . . .	95
4.1.1	Resolução de Zonas Planas . . . . .	95
4.1.2	Aplicações Práticas . . . . .	111
4.2	Análise Comparativa das Implementações . . . . .	116
4.2.1	Exploração da Imagem . . . . .	117
4.2.2	Endereçamento de Caminhos . . . . .	119
4.2.3	Rotulação de Caminhos . . . . .	119
4.2.4	Descoberta e Rotulação de Mínimos Regionais . . . . .	120
4.2.5	Estruturas de Dados . . . . .	121
4.2.6	Considerações . . . . .	122
4.3	Análise de Desempenho do Tempo de Execução . . . . .	123
4.3.1	Análise de Complexidade . . . . .	126
4.3.2	Considerações . . . . .	128
4.4	Análise de Paralelismo . . . . .	129
<b>5</b>	<b>Considerações Finais</b>	<b>137</b>
5.1	Conclusões . . . . .	137
5.2	Trabalhos Futuros . . . . .	139
<b>Referências bibliográficas</b>		<b>141</b>
<b>A Framework de Processamento de Imagens</b>		<b>145</b>

# Listas de Figuras

1.1	Exemplo de aplicação da transformada watershed com marcadores através de reconstrução morfológica . . . . .	3
1.2	Exemplo de aplicação de transformada watershed na identificação de regiões homogêneas . . . . .	4
1.3	Exemplos de resultados das definições TZ-IFT-WT e WC-WT nas imagens beef e csample . . . . .	5
1.4	Comparação de etapas obtidas em momentos equivalentes em dois tipos de algoritmos. 1 <sup>a</sup> Coluna: Largura (Imersão). 2 <sup>a</sup> Coluna: Profundidade (Caminhos Mínimos). .	7
2.1	Exemplo de operações de inserção e remoção sobre uma estrutura de fila hierárquica, com 4 níveis e prioridade por ordem ascendente . . . . .	12
2.2	Exemplo de compressão de caminhos sobre uma estrutura <i>union-find</i> . (a) Conjuntos com raízes em <b>a</b> e <b>g</b> após operações <i>Link</i> , (b) Conjuntos após operações de <i>Find</i> , com caminhos comprimidos . . . . .	12
2.3	Exemplo de padronização para as operações de <i>Arrowing</i> indicando valores utilizados em dois casos. (a) Intervalo possível, definido para vizinhança-8, (b) <i>Arrowing</i> entre <i>pixels</i> de exemplo . . . . .	14
2.4	Exemplo da intuição da simulação de inundação no perfil de uma superfície . . . . .	15
2.5	Exemplo da intuição da simulação de chuva no perfil de uma superfície . . . . .	16
2.6	Aplicação da definição Flooding-WT sobre uma imagem exemplo, representando o resultado em cada limiar . . . . .	17
2.7	Aplicação da definição TD-WT sobre o grafo LCG de uma imagem exemplo, representando o resultado em cada passo verificando-se as arestas para cada <i>pixel</i> . . . . .	19
2.8	Aplicação do lower completion sobre uma imagem exemplo com zonas planas. Representação dos níveis de cinza de acordo com os valores da imagem em escala [0,25].	21
2.9	Aplicação da definição LC-WT sobre um corte do grafo LCG, feito aleatoriamente para remoção de soluções múltiplas, de uma imagem exemplo, representando o resultado em cada passo verificando-se as arestas para cada <i>pixel</i> , . . . . .	23
2.10	Grafo exemplificando o empate de custos considerando-se W o máximo em um caminho terminado no pixel p . . . . .	24
2.11	Aplicação da definição IFT-WT sobre o grafo MOG de uma imagem exemplo, representando o resultado em cada passo pela propagação do rótulo para os vizinhos conectados pelo MOG de cada <i>pixel</i> já rotulado, resolvendo empates aleatoriamente .	25

2.12	Aplicação da definição IFT-WT sobre o grafo MOG de uma imagem exemplo, representando o resultado em cada passo pela propagação do rótulo para os vizinhos conectados pelo MOG de cada <i>pixel</i> já rotulado, aplicando rótulo TZ em caso de empate . . . . .	26
2.13	Construção do grafo do watershed cut a partir de uma imagem exemplo, aplicando os valores mínimos dos pixels em questão de (a) nas arestas em (b) . . . . .	26
2.14	Subgrafos mínimos em destaque, correspondentes aos mínimos regionais . . . . .	27
2.15	Grafo exemplificando a extensão de componentes conexos . . . . .	28
2.16	Grafo de exemplo completo com subgrafos mínimos e alturas mínimas . . . . .	28
2.17	Aplicação do WC-WT sobre grafo construído a partir de imagem exemplo . . . . .	29
2.18	Gráfico indicando os relacionamentos entre as definições de transformada watershed a respeito das bacias hidrográficas . . . . .	30
3.1	Problema de aderência a definição Flooding-WT pelo algoritmo Imersão. (a) Imagem, (b) resultado da definição Flooding-WT, (c) resultado do algoritmo Imersão (N8) . . . . .	34
3.2	Exemplo de inexistência de linha divisória na imersão de Vincent e Soille. (a) Imagem, (b) resultado da definição Flooding-WT e algoritmo Imersão (N4) . . . . .	35
3.3	Divergência de soluções do algoritmo Imersão. (a) Imagem, (b) varredura raster, (c) varredura anti-raster (N4) . . . . .	35
3.4	Análise de rótulos no algoritmo Imersão. (a) Ordem de visitação em varredura raster, (b) rótulos para (a), (c) ordem de visitação em varredura anti-raster, (d) rótulos para (c) . . . . .	36
3.5	Fila de Prioridade. (a) Imagem, (b) Resultado (N4) . . . . .	40
3.6	Dijkstra-Moore de Caminhos Mínimos. (a) Imagem, (b) Resultado (N4) . . . . .	43
3.7	Cálculo do upstream para uma imagem de exemplo, com o downstream indicado por setas a partir dos pixels de origem. . . . .	46
3.8	Hill Climbing (a) Imagem, (b) Resultado (N4) . . . . .	46
3.9	Berge com custo TD-WT. (a) Imagem, (b) Resultado (N4) . . . . .	50
3.10	Componentes Conexos. (a) Imagem, (b) Resultado em raster, (c) Resultado em anti-raster (N4) . . . . .	54
3.11	Construção do DAG. (a) Imagem sem zonas planas, (b) DAG . . . . .	59
3.12	Union-Find. (a) Imagem, (b) Resultado (N4) . . . . .	59
3.13	IFT. (a) Imagem, (b) Resultado, (c) Floresta resultante com raízes em cinza. (N4) . . . . .	63
3.14	Código de Corrente. (a) Imagem, (b) Resultado (N4) . . . . .	67
3.15	Zona de Empate da IFT. (a) Imagem, (b) Resultado (N4) . . . . .	71
3.16	Tobogã Invariante a Ordem. (a) Imagem, (b) Resultado (N4) . . . . .	75
3.17	Imersão Invariante a Ordem. (a) Imagem, (b) Resultado (N4) . . . . .	80
3.18	Caminhos Mínimos. (a) Imagem, (b) Resultado (N4) . . . . .	86
3.19	Watershed Cut. (a) Imagem, (b) Resultado, (c) Grafo correspondente (N4) . . . . .	91
4.1	Imagens utilizadas para experimento de resolução de zonas planas. (a) Original - f, (b) após remoção de zonas planas - lc. . . . .	106
4.2	Resultado dos algoritmos (a) IFT e (b) Berge-MaxLex aplicando custo combinado sobre imagem com zonas planas f . . . . .	107

---

4.3	Resultados dos algoritmos (a) IFT-Rand e (b) Berge-Max aplicando custo máximo sobre imagem com zonas planas f . . . . .	107
4.4	Resultado dos algoritmos (a) IFT-Rand e (b) Berge-Max aplicando custo máximo sobre imagem sem zonas planas lc . . . . .	108
4.5	Zonas de Empate com (a) custo máximo e lexicográfico sobre imagem com zonas planas f, e com custo máximo sobre (b) a imagem com zonas planas f e (c) sem zonas planas lc . . . . .	109
4.6	Zonas de empate de custo máximo sobre (a) imagem beef e (b) beef sem zonas planas. (c) Zona de empate de custo máximo e lexicográfico sobre imagem beef. (d) Subtração entre (a) e (b), indicando pontos resolvidos por lower completion. (e) Subtração entre (b) e (c), indicando pontos onde apenas o custo lexicográfico resolve o empate .	111
4.7	Zonas de empate de custo máximo sobre (a) imagem csample e (b) csample sem zonas planas. (c) Zonas de empate de custo máximo e lexicográfico sobre imagem csample. (d) Subtração entre (a) e (b). (e) Subtração entre (b) e (c). . . . .	112
4.8	Etapas da aplicação beef para geração da imagem para segmentação por transformada watershed utilizando marcadores indiretamente . . . . .	113
4.9	Comparação de resultados das definições através de algoritmos aplicados na imagem beef preparada por marcadores . . . . .	114
4.10	Etapas da aplicação concrete para segmentação de regiões homogêneas . . . . .	115
4.11	Comparação de resultados das definições de watershed através de algoritmos, na aplicação de identificação de regiões homogêneas . . . . .	116
4.12	Gráfico comparativo do tempo de execução médio dos algoritmos em segundos por número de <i>pixels</i> , para imagens com zonas planas e sem filtragem . . . . .	125
4.13	Gráfico comparativo do tempo de execução médio dos algoritmos em segundos por número de <i>pixels</i> , para imagens sem zonas planas e sem filtragem . . . . .	127
A.1	Exemplo de transformação de (a) imagem em (b) vetor unidimensional . . . . .	146
A.2	Exemplo de tratamento das bordas na transformação de (a) imagem com borda em (b) vetor com borda, onde estes pixels são intercalados . . . . .	149



# **Lista de Tabelas**

4.1	Medições sobre a aplicação de deteção de regiões homogêneas . . . . .	115
4.2	Resumo das características dos algoritmos estudados . . . . .	131
4.3	Mínimos regionais por imagens e por tamanhos . . . . .	132
4.4	Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens com zonas planas não filtradas . . . . .	132
4.5	Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens com zonas planas filtradas . . . . .	132
4.6	Perda de desempenho percentual para os algoritmos, por tamanho de imagem, entre as imagens com zonas planas filtradas e não filtradas . . . . .	133
4.7	Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens sem zonas planas não filtradas . . . . .	133
4.8	Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens sem zonas planas e filtradas . . . . .	134
4.9	Perda de desempenho percentual para os algoritmos, por tamanho de imagem, entre as imagens sem zonas planas filtradas e não filtradas . . . . .	134
4.10	Comparação de desempenho para os algoritmos por tamanho de imagem, entre as imagens sem filtragem com e sem zonas planas . . . . .	135



# **Trabalhos Publicados Pelo Autor**

1. A. Körbes, R. Lotufo. “Analysis of the watershed algorithms based on the Breadth-First and Depth-First exploring methods”. In SIBGRAPI’09, pg. 133-140, Rio de Janeiro, Brazil, Oct. 2009. IEEE Computer Society.
2. A. Körbes, G. B. Vitor, J. V. Ferreira, R. Lotufo. “A proposal for a parallel watershed transform algorithm for real-time segmentation”. In Proceedings of Workshop de Visão Computacional WVC’2009, So Paulo, Brazil, Sep. 2009. Díspónivel em [http://iris.sel.eesc.usp.br/wvc2009/WVC2009\\_CD.rar](http://iris.sel.eesc.usp.br/wvc2009/WVC2009_CD.rar).
3. A. Körbes, R. Lotufo. “On Watershed Transform: Plateau Treatment and Influence of the Different Definitions in Real Applications”. In IWSSIP’2010, accepted, Rio de Janeiro, Brazil, Jun. 2010.
4. R. Lotufo, R. Machado, A. Körbes, R. Ramos. “Adessowiki on-line collaborative scientific programming platform”. In WikiSym ’09: Proceedings of the 5th International Symposium on Wikis and Open Collaboration, pages 1-6, New York, NY, USA, 2009. ACM.



# Capítulo 1

## Introdução

Tarefas de segmentação de imagens são recorrentes em sistemas de visão computacional, onde deseja-se identificar regiões de acordo com certas características, constituindo geralmente uma etapa intermediária de um sistema maior, onde os resultados da segmentação serão utilizados para outros procedimentos. Desta forma, deseja-se que este seja robusto, forneça regiões bem delimitadas e detalhado o suficiente para que sua interpretação seja realizada corretamente. A segmentação de imagens não é um problema trivial, sendo amplamente investigado na literatura através de diversas abordagens e analisado em problemas específicos para obtenção dos resultados desejados. De forma geral, as técnicas existentes são classificadas em duas categorias: detecção de bordas ou descontinuidades e crescimento de regiões [1], sendo a transformada *watershed* classificada na segunda categoria.

A transformada *watershed* propõe uma abordagem morfológica para o problema de segmentação de imagens, interpretando estas como superfícies, onde cada *pixel* corresponde a uma posição e os níveis de cinza determinam as altitudes. A partir desta noção, deseja-se então identificar bacias hidrográficas, definidas por mínimos regionais e suas regiões de domínio. Este conceito, estudado há muito tempo para definições de linhas de divisão da água [2], foi introduzido no estudo de imagens digitais por Digabel e Lantuéjoul [3], e mais tarde, já denominada transformada *watershed* foi utilizada por Beucher e Lantuéjoul [4] com o mesmo objetivo, identificar regiões em uma superfície, detectando seus contornos. Estas propostas introdutórias, juntamente com o trabalho em segmentação morfológica de Meyer e Beucher [5], estabeleceram o uso da transformada *watershed*.

Intuitivamente, a transformada *watershed* trata de encontrar os pontos em uma superfície onde uma gota d'água possa escorrer para dois mínimos regionais diferentes. Esta analogia pode ser também criada inversamente, onde um nível d'água é elevado através de uma superfície, inundando a partir dos mínimos regionais, e, nos pontos onde águas provenientes de mínimos diferentes se tocam ergue-se uma barreira, que constitui a linha de divisão das bacias. Conceitualmente equivalentes, estas noções são formalizadas diversamente, onde as primeiras abordagens buscam imitar diretamente o processo intuitivo [6, 7]. Entretanto, tais conceitos foram demonstrados serem equivalentes a problemas clássicos da computação, e assim, definições mais recentes optam por uma abordagem relacionada a problemas de otimização de custos de caminhos em grafos [8, 9].

Este trabalho trata de uma análise sistemática dos algoritmos da transformada *watershed*, que implementam as diversas definições existentes. Para tal efeito, considerou-se a literatura a partir da introdução da primeira transformada rápida, em 1991, por Vincent e Soille [6], até os trabalhos recentes de Cousty *et al.* [9]. Na mesma linha, outros autores realizaram trabalhos similares, como

Hagyard, Razaz e Atkin [10] com a primeira análise de desempenho comparando os algoritmos de Vincent e Soille [6] e Meyer [7], medindo seus tempos de execução com implementações eficientes, onde, apesar de já existente, o algoritmo de Beucher e Meyer [11] não foi incluído. Roerdink e Meijster revisitaram a área no ano 2000, realizando uma análise extensiva das três definições existentes até o momento e dos seis algoritmos que as implementavam, verificando a corretude destes e seu modo de funcionamento [12]. Após Roerdink e Meijster, mais recentemente, as definições necessitaram de uma revisão, dada a inclusão de novas técnicas, sendo possível então relacioná-las com base em seus resultados teóricos [13]. Neste trabalho os relacionamentos entre as definições são estendidos aos algoritmos, analisando a literatura sistematicamente, de modo a produzir um revisão atualizada sobre o assunto.

Assim, este trabalho resume a literatura de algoritmos de transformada *watershed* buscando ser o mais amplo possível dentro do assunto. Desta forma, catorze algoritmos foram encontrados com a denominação de *watershed* - cobrindo vastamente a área - e utilizando técnicas e conceitos embasados em definições ou nas noções intuitivas de determinação de bacias hidrográficas. Sabe-se da equivalência dos algoritmos de *watershed* com segmentação *fuzzy* conexa, entretanto, este método não foi incluído [14]. De forma a uniformizar este conhecimento, optou-se por reproduzir em pseudocódigo os algoritmos citados, e, através de uma notação comum, buscar equivalência em funcionalidades, aproximando pontos onde os algoritmos executam tarefas similares. Em paralelo, buscou-se também produzir programas operacionais utilizando a linguagem Python [15] e que fossem o mais próximas o possível dos pseudocódigos, utilizando para isto as abstrações necessárias em termos de dimensionalidade e limitações de domínio.

Este trabalho foi desenvolvido utilizando o ambiente Adessowiki<sup>1</sup> como ferramenta de suporte ao gerenciamento dos documentos e programas necessários. Neste ambiente, operado como uma *wiki*, são geradas páginas *web* contendo texto estruturado e código Python executável, capaz de gerar elementos como imagens para compor o resultado final desejado [16]. Além disso, o conteúdo do texto pode ser continuamente mantido e atualizado de forma colaborativa, sendo uma fonte dinâmica de informações sobre a transformada *watershed*.

## 1.1 Motivação

Aplicações da transformada *watershed* são comuns na literatura em diversas áreas do processamento de imagens. Seu uso é em geral associado a algum gradiente, seja morfológico ou não, provendo uma imagem pré-processada adequada à segmentação de regiões via transformada *watershed*. Um problema comum em tais abordagens, conhecido na literatura e tratado de diversas formas, é a supersegmentação, gerando muito mais regiões do que o esperado e/ou desejado para a imagem de entrada. De modo a reduzir estes problemas, as imagens de gradiente podem ser simplificadas, seja utilizando filtros, como por exemplo o fechamento, ou marcadores em uma reconstrução morfológica, diminuindo o número de mínimos regionais, e por consequência o número de regiões identificadas. Uma abordagem própria da transformada *watershed* é o uso de marcadores como entrada, provendo a classe de *watershed* por marcadores, mais eficiente por eliminar uma etapa usual de pré-processamento.

---

<sup>1</sup><http://www.adessowiki.org>

A aplicação clássica da transformada *watershed* é na segmentação de regiões para pós-processamento destas, como medição de área, perímetro, etc. A Fig. 1.1 apresenta uma aplicação onde deseja-se medir a área interna (sem gordura) do bife, equivalente à quantidade de carne neste. Para isto aplica-se sobre a imagem de entrada filtros morfológicos de fechamento e fechamento por área, junto com operações de limiarização, erosão e dilatação, obtendo assim marcadores externos e internos à região desejada [17]. Utilizando estes marcadores processa-se uma reconstrução morfológica na imagem inversa, forçando a criação de apenas dois mínimos regionais. Aplica-se então a transformada *watershed*, detectando duas regiões correspondentes aos marcadores. Pode-se então calcular a área da região desejada e apresentar o contorno desta sobre a imagem original, como na Fig. 1.1. Este constitui um exemplo típico de uso, onde a imagem é filtrada e a transformada é utilizada para detecção da região de interesse.

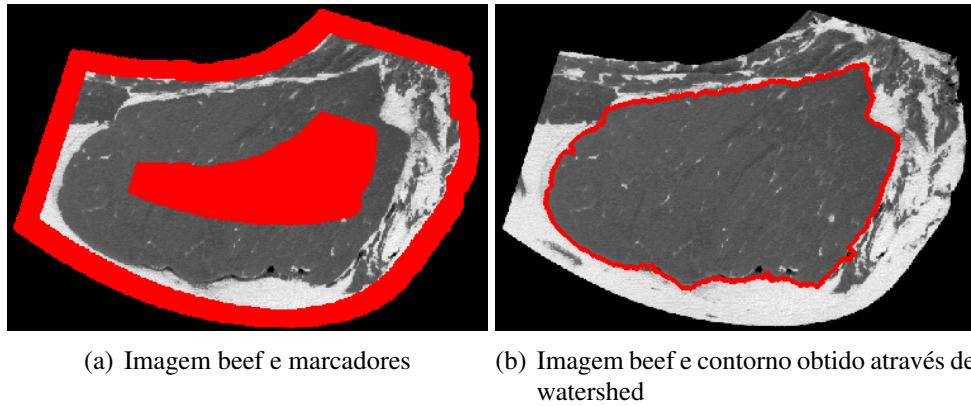


Fig. 1.1: Exemplo de aplicação da transformada watershed com marcadores através de reconstrução morfológica

Apesar do problema de supersegmentação recorrente à transformada *watershed*, este pode ser usado de forma benéfica em algumas aplicações. Um exemplo possível é relacionado à detecção de regiões homogêneas, apresentado na Fig. 1.2 [17]. Nesta aplicação, deseja-se medir a área de tais regiões, sendo aplicado para isto um algoritmo convencional de *watershed* sobre uma imagem de gradiente filtrada por contraste. Neste caso, ocorre severa supersegmentação, devido ao grande número de mínimos regionais. Entretanto, nas regiões de textura homogênea, se formam componentes conexos de área maior que podem ser separados dos demais por um filtro de área. Para correção de ruídos dentro destas regiões, aplica-se um filtro de fechamento por área, que os elimina dentro do critério da aplicação. Pode-se dizer que neste caso a transformada *watershed* é aplicada como um detector de texturas. Na Fig. 1.2 são apresentados os contornos obtidos utilizando esta técnica.

Conforme mencionado anteriormente, a transformada *watershed* é formalizada através de diversas definições com soluções produzidas por diversos algoritmos. Nas aplicações apresentadas nas Figs. 1.1 e 1.2, intencionalmente, estas particularidades não foram especificadas. A razão para isto é a noção geral de que os resultados das várias transformadas *watershed* são equivalentes, não sendo comum na literatura a especificação de qual algoritmo e definição foram utilizados para obtenção de resultados nas aplicações. Assim, as definições **Flooding-WT** [6], **TD-WT** [7], **LC-WT** [18], **IFT-WT** [19], **TZ-IFT-WT** [20] e **WC-WT** [9] são revisitadas neste trabalho, de forma a determinar sua

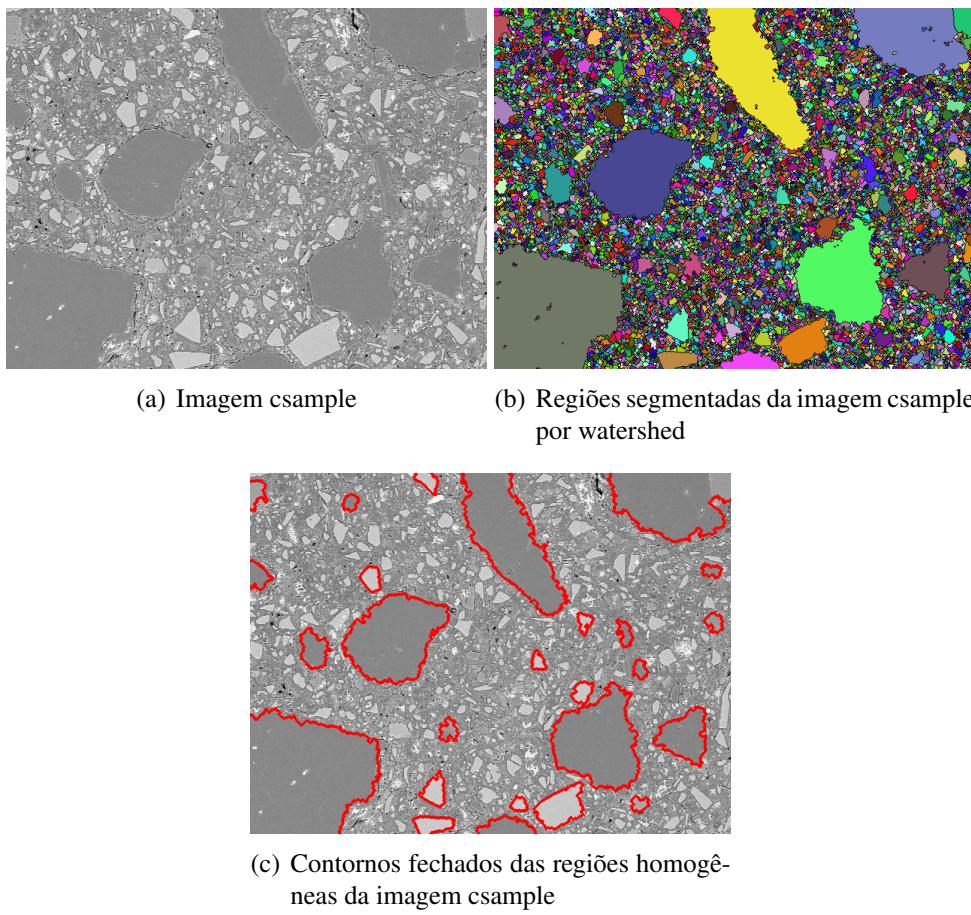


Fig. 1.2: Exemplo de aplicação de transformada watershed na identificação de regiões homogêneas

influência no resultado dos algoritmos. Apesar de as diferenças entre as definições serem contidas na zona de empate [21], com exceção do **WC-WT** e **Flooding-WT**, nem sempre estas diferenças podem ser dadas como equivalentes. A Fig. 1.3 apresenta os dois exemplos de aplicação anteriores utilizando-se transformadas diferentes. No caso das imagens (a) e (b), a diferença de área medida é de 1,1%, e na imagem (c) são detectadas 7 regiões a menos do que em (d). Nesta comparação, as imagens em (a) e (c) correspondem à definição **TZ-IFT-WT** e as imagens (b) e (d) correspondem à definição **WC-WT**. Para muitas aplicações estas diferenças podem ser descartadas, tratadas como ruído ou erro, e consideradas equivalentes. No entanto, casos especiais podem ser influenciados por estes resultados e deve-se então tomar os devidos cuidados.

A principal investigação neste trabalho é relativa à diversidade de algoritmos e como estes se relacionam com as definições da transformada *watershed*. São estudados 14 algoritmos da literatura, por ordem de aparição: Imersão [6], Fila de Prioridade [11], Dijkstra-Moore de Caminhos Mínimos, Hill-Climbing e Berge de Caminhos Mínimos [7], Componentes Conexos [18], Union-Find [22], IFT [19], Código de Corrente [23], Zona de Empate [20], Tobogã Invariante a Ordem e Imersão Invariante a Ordem [24], Caminhos Mínimos [25] e Watershed Cut [9]. Além da relação com as definições, nem sempre explicitadas nas publicações dos algoritmos, o comportamento destes também motivou este

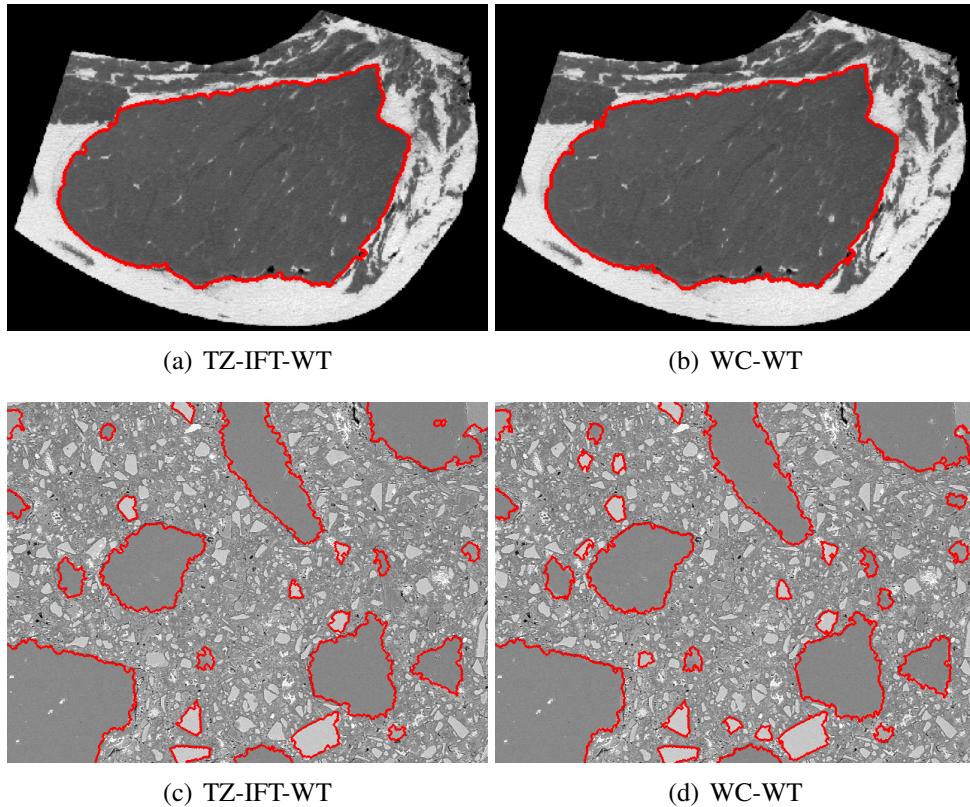


Fig. 1.3: Exemplos de resultados das definições TZ-IFT-WT e WC-WT nas imagens beef e csample

trabalho, implicando no estudo das técnicas de implementação, formas de utilização de estruturas de dados, e a busca por uma classificação mais refinada.

A busca por tal classificação destes algoritmos parte inicialmente dos princípios intuitivos da imersão e da gota d’água. Estes, vistos sob a perspectiva de exploração de vértices em grafos, tornam-se similares às buscas em largura e em profundidade, respectivamente. Entretanto, apesar de serem estratégias fundamentalmente diferentes, os resultados finais produzidos pelos algoritmos são praticamente iguais, com diferenças sutis, conforme mencionado anteriormente e explorado em mais profundidade no Cap. 4. A Fig. 1.4 apresenta os algoritmos de Imersão de Vincent e Soille e de Caminhos Mínimos de Osma-Ruiz, comparando seus resultados parciais de modo a ressaltar a classificação em busca em largura e profundidade.

## 1.2 Objetivos

O objetivo principal deste trabalho é produzir uma revisão sistemática atualizada da literatura sobre a transformada *watershed*, restrita ao método clássico, desconsiderando abordagens por marcadores, hierárquicas, estocásticas, etc., visto que estas são especializações para determinados tipos de problemas. Como parte integrante de tal revisão, tem-se como objetivo a identificação das características principais de cada algoritmo, possibilitando a classificação destes, bem como a generalização

em categorias. Deseja-se também associar cada algoritmo à definição correspondente, muitas vezes não reportada pelos autores originais, ou implementada parcialmente. Nestes últimos casos, os algoritmos são corrigidos em sua especificação, quando não são mudanças que influenciem na ordem destes nem em sua arquitetura geral. Desta forma, deseja-se produzir uma coleção consistente de algoritmos da transformada *watershed*, solidificando conhecimentos esparsos e confusos na literatura do assunto.

Entre as principais comparações realizadas neste estudo, está a classificação de acordo com a visitação e associação com a busca em largura e em profundidade, consideradas técnicas clássicas no estudo de algoritmos. Características como as técnicas de uso de endereços de *pixels*, rotulação de caminhos, descoberta de mínimos regionais, e os usos das diferentes estruturas de dados também mostram-se interessantes para auxiliar na comparação entre os algoritmos. Análises de desempenho, complexidade e possibilidades de paralelismo também são necessárias para se obter uma revisão completa, indicando os algoritmos com tendência a obter melhor desempenho.

### 1.3 Organização

Esta dissertação está organizada da seguinte forma: o Cap. 2 apresenta as convenções e notações utilizadas, além das definições da transformada *watershed*, apresentadas conforme aparição na literatura, utilizando exemplos com imagens numéricas de modo a ressaltar as diferenças. O Cap. 3 revisa os algoritmos da transformada *watershed*, em ordem cronológica, ressaltando as suas características, comportamento e padronizando o pseudocódigo, bem como oferecendo uma implementação Python para cada um deles. No Cap. 4, são feitas análises comparativas entre as características dos algoritmos, dos resultados e técnicas referentes à resolução de zonas planas, de desempenho do programa Python e de paralelismo, incluindo uma breve revisão bibliográfica neste último tópico. No Cap. 5 as contribuições deste trabalho são discutidas, e trabalhos futuros são propostos para sua continuidade. O Apêndice A apresenta o *framework* para processamento de imagens desenvolvido de modo a abstrair problemas comuns na implementação de algoritmos como a transformada *watershed*.

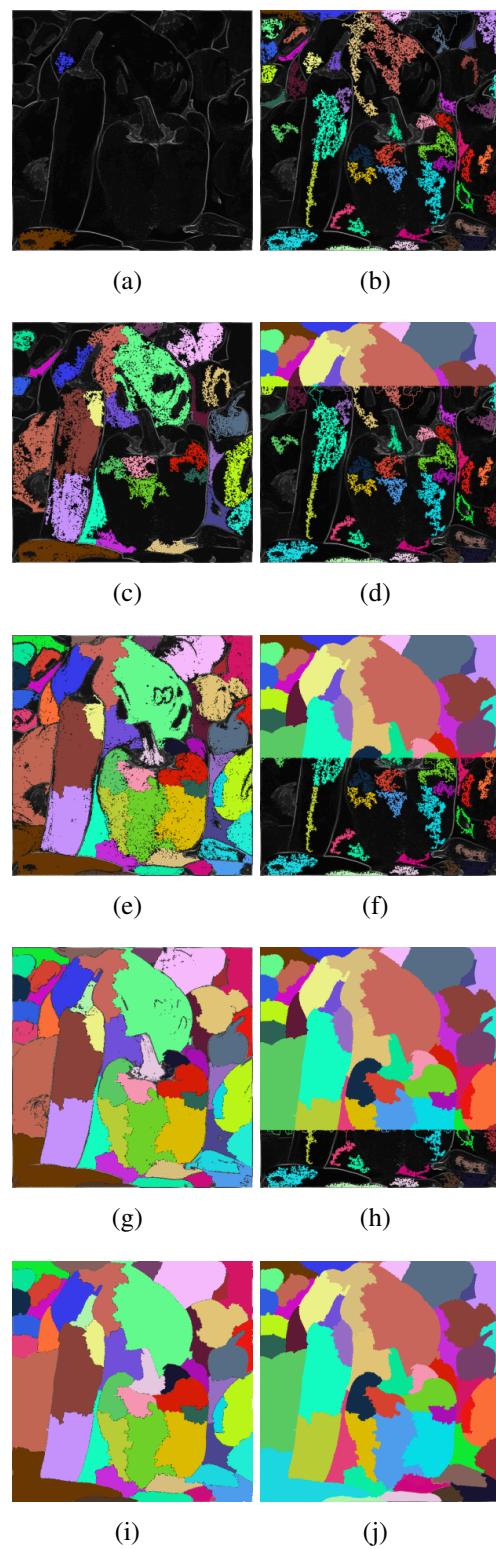


Fig. 1.4: Comparaçāo de etapas obtidas em momentos equivalentes em dois tipos de algoritmos. 1<sup>a</sup> Coluna: Largura (Imersão). 2<sup>a</sup> Coluna: Profundidade (Caminhos Mínimos).



# Capítulo 2

## Definições e Convenções

Neste capítulo, são apresentadas as definições e convenções adotadas para compreensão deste trabalho. São expostas as formas de uso das imagens, além de padrões utilizados nos algoritmos. Em seguida, a transformada *watershed* é introduzida e as definições e equações existentes na literatura são detalhadas. Procura-se com este capítulo solidificar fundamentos necessários para a compreensão dos algoritmos e sua padronização, para posterior especificação individual.

### 2.1 Convenções

Nesta seção são apresentadas as nomenclaturas para as imagens dependendo de seu uso, operações comuns em estruturas de dados, procedimentos nos algoritmos, além dos tipos de laço de repetição **for** utilizados na especificação dos algoritmos. Os algoritmos de transformada *watershed* têm uma característica particular, que permite sua fácil extensão para qualquer dimensão, necessitando apenas de uma relação de ordem para os valores e uma relação de vizinhança entre os *pixels*. Nesta seção define-se a forma como abstraem-se estes elementos das imagens.

#### 2.1.1 Imagens, dados e entradas

Toda imagem  $I$  é considerada uma função  $I : D \rightarrow [h_{min}, h_{max}]$ , onde  $D \subset \mathbb{N}^N$  é o domínio da imagem,  $[h_{min}, h_{max}] \subset \mathbb{R}$  o intervalo de valores possíveis e  $N$  é o número de dimensões da imagem. Denota-se por  $I(p)$  o valor de  $p \in D$ . *Pixels* pertencentes ao domínio são representados com as letras  $p$ ,  $q$ ,  $u$  e  $v$ , conforme necessidade e nesta ordem. No entanto, deve-se tomar o cuidado para que a representação multidimensional seja factível, normalmente vista como: 2D uma imagem em nível de cinza; 3D um sólido - geralmente obtido de imagens médicas - onde as regiões segmentadas serão volumes deste sólido, ou uma sequência de imagens; 4D sólidos ao longo do tempo - também advindos de imagens médicas em geral - com as regiões correspondendo a volumes no tempo.

A entrada para todos os algoritmos é uma imagem  $im$  e a saída é a imagem de rótulos inteiros positivos  $lab$ . Em alguns casos, a imagem de entrada necessita ser pré-processada para remoção de zonas planas, passando a ser mantida em  $lc$ , com mesmo domínio. Nos algoritmos que necessitam de mínimos regionais para realizar seu processamento, estes são fornecidos no conjunto  $M = \{m_1, m_2, \dots, m_n\}$ , onde  $m_i$  são subconjuntos contendo os *pixels* de cada mínimo regional  $i$ .

A maioria dos algoritmos apresentados no Cap. 3 fazem uso de imagens de trabalho, intermediárias, para cálculo de distâncias geodésicas, denotadas *dist*, ou de endereços de outros *pixels*, neste caso *adr*. A relação de vizinhança de um *pixel*  $p$  é dada por  $N(p)$ , onde este é um subconjunto de  $D$ , contendo os *pixels* conectados a  $p$ , de acordo com uma regra pré-estabelecida, como por exemplo vizinhança-4 ou 8 em 2D e vizinhança-6 ou 26 em 3D. Nesta relação ainda pode-se estabelecer duas restrições,  $N^+(p)$  e  $N^-(p)$  que, dada uma ordem de visitação dos *pixels* em  $N(p)$ , incluindo  $p$  (e.g. *raster*, *anti-raster*), indicam respectivamente os *pixels* a serem visitados depois e antes de  $p$ . Em relação aos algoritmos baseados em grafos, utiliza-se para estes a notação tradicional  $G = (V, E, F)$ , onde  $V$  é o conjunto de vértices, correspondente a  $D$ , isto é, cada *pixel* da imagem é um nó do grafo,  $E$  é o conjunto de arestas, construído a partir da relação de vizinhança, e  $F$  é a função de mapeamento de valores nas arestas. As convenções de nomes adotadas aqui são mantidas nas especificações dos algoritmos, procurando assim uniformizá-los e facilitar sua comparação e entendimento.

## 2.1.2 Estruturas de Dados

As estruturas de dados utilizadas nos algoritmos de transformada *watershed* têm um papel fundamental na sua eficiência e em seu comportamento no que diz respeito à ordem de varredura dos *pixels*. São quatro as estruturas utilizadas pelos algoritmos, a fila (FIFO), pilha (LIFO), fila de prioridade com FIFO e conjuntos *union-find*. No caso da fila, pilha e fila de prioridade, para simplicidade, quando o algoritmo em questão utiliza apenas uma instância da estrutura, esta não é incluída nos parâmetros da operação a ser realizada. Nos outros casos, o último parâmetro especifica sobre qual das instâncias deve ser realizada a ação, sendo denotado como opcional por colchetes. Um resumo das operações é apresentado abaixo.

### a. Estrutura de Fila (FIFO)

- *QueuePush( $p$ , [queue])*: Insere na fila *queue* o elemento  $p$ .
- *QueuePop([queue])*: Remove e devolve da fila *queue* o elemento na ordem FIFO.
- *QueueEmpty([queue])*: Devolve verdadeiro se a fila *queue* estiver vazia, falso caso contrário.
- *QueueClear([queue])*: Esvazia a fila.

### b. Estrutura de Pilha (LIFO)

- *StackPush( $p$ , [stack])*: Insere na pilha *stack* o elemento  $p$ .
- *StackPop([stack])*: Remove e devolve da pilha *stack* o elemento na ordem LIFO.
- *StackEmpty([stack])*: Devolve verdadeiro se a pilha *stack* estiver vazia, falso caso contrário.

### c. Estrutura de Fila Hierárquica (fila de prioridade) (*Heap Queue*, *Priority Queue*)

- *HeapQueuePush( $p$ ,  $v$ , [heap])*: Insere na fila *heap* o elemento  $p$  com prioridade  $v$ .

- $\text{HeapQueuePop}([\text{heap}])$ : Remove e devolve da fila  $\text{heap}$  o elemento de maior prioridade.
- $\text{HeapQueueEmpty}([\text{heap}])$ : Devolve verdadeiro se a fila  $\text{heap}$  estiver vazia, falso caso contrário.
- $\text{HeapQueueContains}(p, [\text{heap}])$ : Devolve verdadeiro se o elemento  $p$  estiver contido na fila  $\text{heap}$ .
- $\text{HeapQueueRemove}(p, [\text{heap}])$ : Remove da fila  $\text{heap}$  o elemento  $p$ .

d. Estrutura de conjunto *Union-Find*:

- $\text{MakeSet}(x)$ : Cria um conjunto  $\{x\}$
- $\text{Link}(x,y)$ : Conecta o elemento  $y$  a  $x$
- $\text{Find}(x)$ : Percorre o conjunto a partir de  $x$  até encontrar o elemento representativo deste, comprimindo o caminho, e o devolve
- $\text{Union}(x,y)$ : Cria um novo conjunto a partir da união dos conjuntos cujas raízes são  $x$  e  $y$ , transformando  $x$  na raiz de ambos

A implementação eficiente destas estruturas de dados foge ao escopo deste trabalho, sendo alvo específico da literatura de algoritmos e teoria da computação [26, 27]; no entanto sua compreensão é importante. As estruturas de fila e pilha são bastante conhecidas, sendo listas mantidas em memória com comportamento especial para inserção e retirada de elementos seguindo as políticas FIFO e LIFO respectivamente. A fila de prioridade (ou fila hierárquica), introduzida na transformada *watershed* por Beucher e Meyer [11], possui duas características importantes que determinam a ordem de remoção de elementos, sendo a prioridade individual e o desempate por FIFO. Uma fila de prioridade pode ser vista também como uma coleção de filas FIFO, uma para cada valor possível, onde insere-se os elementos na fila com seu valor respectivo e a remoção é feita a partir da fila que não estiver vazia correspondente ao menor valor possível entre todas. A Fig. 2.1 exemplifica esta estrutura com 4 operações de inserção de elementos  $p$ ,  $q$ ,  $u$  e  $v$  com valores 2, 3, 3 e 1 respectivamente, e 4 remoções, apresentando as suas duas características principais, através da noção de várias filas.

A estrutura de dados *union-find* não é utilizada em sua totalidade em nenhum dos algoritmos, no entanto a sua operação *Find* tem extensa influência, assim como a técnica de compressão de caminhos, sendo importante sua compreensão neste contexto. A estrutura *union-find* consiste em utilizar elementos representativos para identificar conjuntos desconexos, armazenados em um mesmo espaço. Assim, conjuntos são expandidos utilizando-se o elemento representativo - a raiz do conjunto - como elo entre todos os elementos de um conjunto. A operação de conexão no entanto não requer que um dos elementos seja a raiz, e assim, ao conectar dois elementos que não são raizes de conjuntos, formam-se caminhos até esta, pois na conexão o novo elemento passa a ser ligado ao já existente, e consequentemente à raiz do conjunto. O atravessamento destes caminhos, necessário na união de dois conjuntos e para identificar a raiz de um elemento qualquer, realiza então a compressão destes, diminuindo o número de saltos necessários até a raiz em uma próxima visita. A Fig. 2.2 apresenta em (a) dois conjuntos, cujas raízes são os elementos **a** e **g**, e em (b) o estado do conjunto após a execução de uma operação *Find* sobre o elemento **e**.

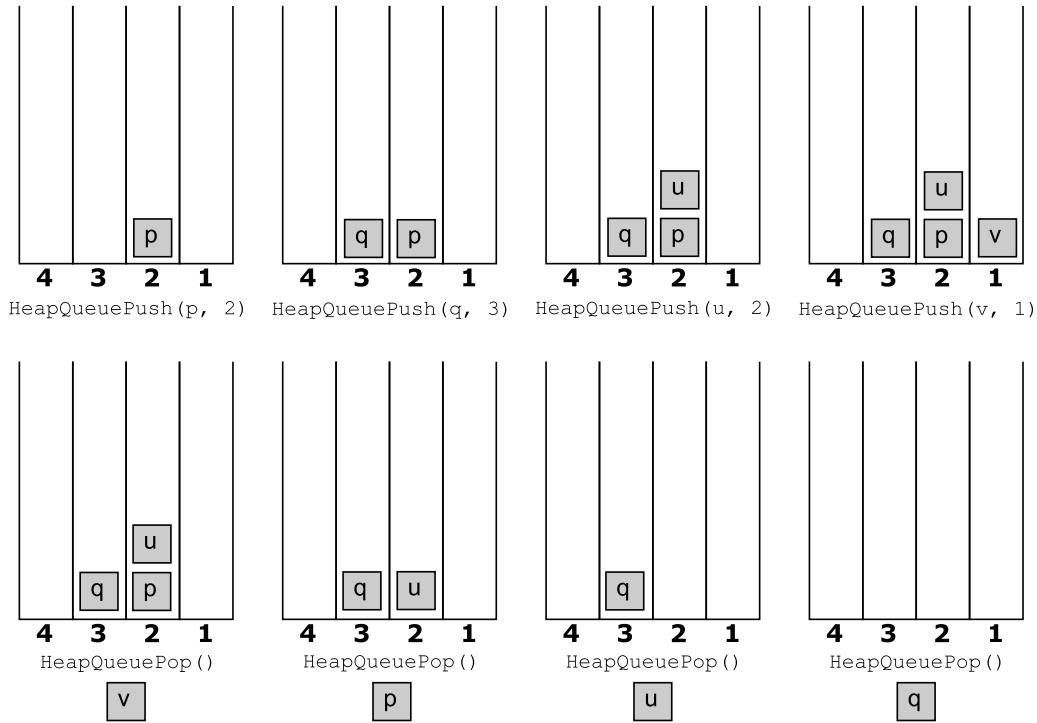


Fig. 2.1: Exemplo de operações de inserção e remoção sobre uma estrutura de fila hierárquica, com 4 níveis e prioridade por ordem ascendente

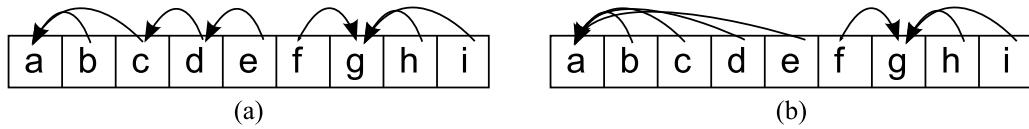


Fig. 2.2: Exemplo de compressão de caminhos sobre uma estrutura *union-find*. (a) Conjuntos com raízes em **a** e **g** após operações *Link*, (b) Conjuntos após operações de *Find*, com caminhos comprimidos

### 2.1.3 Notação de Laços

Na especificação dos algoritmos buscou-se atentar a detalhes de notação para evitar ambiguidades. Esta análise é feita em especial nos laços de repetição **for**, que, quando paralelizáveis, são substituídos por um laço **for all**. Busca-se com esta convenção também resolver ambiguidades comuns em algoritmos onde o laço **for all** é aplicado para denotar o símbolo  $\forall$ , ou apenas um laço **for** sobre todos elementos de um conjunto. A análise destes laços leva em consideração duas condições: ordem, que implica na iteração de um conjunto onde os elementos devem ser analisados segundo alguma regra (e.g. domínio em *raster*, ordem de níveis de cinza); e concorrência, que implica processamento de uma iteração afetar outras.

Desta forma, laços **for** denotam concorrência, e podem ou não implicar em ordem específica, dependendo do tipo de conjunto sendo varrido. Assim, um laço **for** deverá iterar sobre cada elemento

do conjunto de forma sequencial, pois os resultados de uma iteração interferem ou são utilizados nas próximas. A ordem de análise do conjunto depende da sua representação. Conjuntos especificados por uma letra, como a iteração no domínio da imagem  $D$ , não implicam ordem, sendo que os elementos deste podem ser iterados de forma arbitrária. Intervalos, como  $[h_{min}, h_{max}]$  devem ser iterados na ordem do menor para o maior elemento.

Laços **for all** denotam que não há interferência entre as operações e que o conjunto pode ser iterado em qualquer ordem. Desta forma, as operações efetuadas dentro do laço são independentes para cada elemento do conjunto em questão, também denotado na literatura como **parallel for** ou **parfor** [28]. Um laço **for all** tem o mesmo efeito do símbolo  $\forall$ , utilizado em pontos de inicialização de alguns algoritmos, denotando a possibilidade de se efetuar a mesma operação sobre todos os elementos do conjunto paralelamente. Para que isto seja possível, as operações efetuadas não devem influenciar o processamento umas das outras. Nesta análise, considera-se que as operações sobre estruturas de dados são atômicas.

### 2.1.4 Operações de *Arrowing*

As operações de *arrowing* se repetem entre os algoritmos por serem estratégias comuns para construção de caminhos mínimos. No entanto, pequenas variações, como valores de retorno e estratégias de uso ocorrem, e, por esse motivo o corpo destas operações não é explicitado, apenas funcionalidade básica, de forma a facilitar a compreensão dos algoritmos, abstraindo sua especificação nestes.

A operação  $Arrow(p, q)$  tem como objetivo indicar, através de um número, a direção relativa de  $p$  para  $q$ , construindo o caminho de máxima inclinação. O valor devolvido por esta função é padronizado diferentemente dependendo do algoritmo, geralmente um número de 1 a  $N$ , onde  $N$  é o número máximo de vizinhos de cada *pixel*, ocorrendo variação entre os algoritmos no intervalo utilizado. A Fig. 2.3 apresenta uma possível padronização de direções, exemplificando o uso da operação *Arrow*.

Para recuperação do endereço de um *pixel*  $q$  a partir do endereço de um *pixel*  $p$  vizinho e um número  $n$  em um intervalo definido é utilizada a função  $Pointed(p, n)$ . Esta função permite recuperar o endereço do próximo *pixel* em um caminho. Assim, dada uma padronização conforme exemplo da Fig. 2.3, pode-se recuperar o vizinho deste indicado por  $n$ . Na Fig. 2.3 considera-se uma vizinhança-8, onde cada vizinho de  $p$  em (a) é indicado por um número inteiro no intervalo  $[1, 8]$ , e em (b), apresenta-se o posicionamento dos *pixels*  $p$ ,  $q$  e  $u$ , e o direcionamento desejado

Tem-se então que  $Arrow(p, q) = 2$  e  $Arrow(u, q) = 1$ , de acordo com o representado pelas linhas tracejadas. A recuperação destes caminhos é dada por  $Pointed(p, 2) = q$  e  $Pointed(u, 1) = q$ , onde  $q$  é o endereço do *pixel* vizinho do *pixel* endereçado por  $p$ .

## 2.2 Definições de Watershed

Nesta seção é realizada uma breve revisão das definições da transformada *watershed* nas quais os algoritmos estudados neste trabalho se baseiam para produção de seus resultados. Estas definições são: imersão (Flooding-WT) [6], distância topográfica (TD-WT) [7], condição local (LC-WT) [29], transformada imagem-floresta com custo máximo de caminho (IFT-WT) [19], zona de empate da IFT-WT (TZ-IFT-WT) [20] e *watershed cut* (WC-WT) [9]. A abordagem escolhida neste assunto

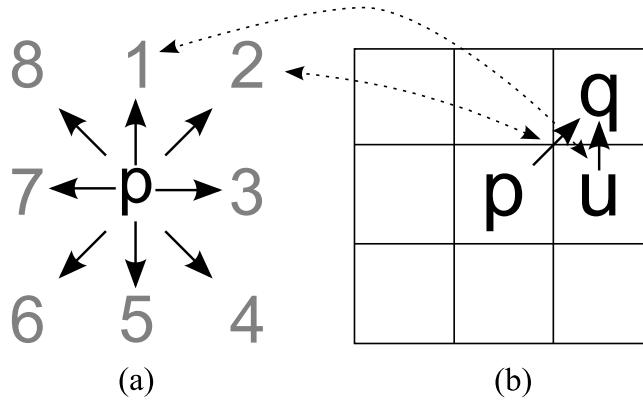


Fig. 2.3: Exemplo de padronização para as operações de *Arrowing* indicando valores utilizados em dois casos. (a) Intervalo possível, definido para vizinhança-8, (b) *Arrowing* entre *pixels* de exemplo

é suficiente para a compreensão do funcionamento dos algoritmos, entretanto um estudo mais aprofundado nas definições e nos seus relacionamentos é apresentado por Audigier [13] e por Roerdink e Meijster [12]. É importante ressaltar o fato de que a existência de diversas definições implica em diferentes espaços de soluções, que podem ser relacionados [21].

### 2.2.1 Transformada Watershed

A transformada *watershed* é baseada na noção de divisão de águas de uma superfície, tal como um terreno, onde deseja-se identificar as bacias de captação dos mínimos regionais. A localização destas linhas d'água pode ser dada intuitivamente de duas formas: elevando um nível d'água uniformemente na superfície e traçando as linhas nos pontos onde águas provenientes de dois mínimos regionais diferentes se tocam, ou localizando os pontos onde uma gota d'água pode escorrer para mínimos diferentes. A primeira noção pode ser visualizada na Fig. 2.4, onde apresenta-se uma superfície e um nível d'água, que eleva-se, identificando os mínimos regionais por cores diferentes e traçando linhas onde as águas se encontram.

A segunda noção tem sua representação na Fig. 2.5, onde, na mesma superfície, são apresentadas sequencialmente gotas d'água sobre esta, traçando linhas divisórias nos pontos onde uma gota poderia escorrer para dois mínimos regionais diferentes. No entanto, em imagens, a localização destes pontos pode não ser trivial, pois não são necessariamente máximos locais na superfície - dois caminhos distintos podem se unir e seguir para um mesmo mínimo regional - e torna-se complicada também a representação gráfica.

Estas noções intuitivas, chamadas princípios de imersão e de gota d'água são fundamentais para a compreensão das definições apresentadas a seguir. A aplicação destes sobre uma imagem é feita de forma a considerar os níveis de cinza destas como os valores de altitude. Entretanto, para a obtenção de contornos de objetos, em geral aplica-se um filtro derivativo sobre a imagem, para realce das diferenças onde estas atingem valores mais altos e sendo o local de posicionamento das linhas do *watershed*. Uma estratégia bastante comum no âmbito da transformada *watershed* é também a filtragem de mínimos regionais para eliminar aqueles menos relevantes, e dessa forma obter regiões

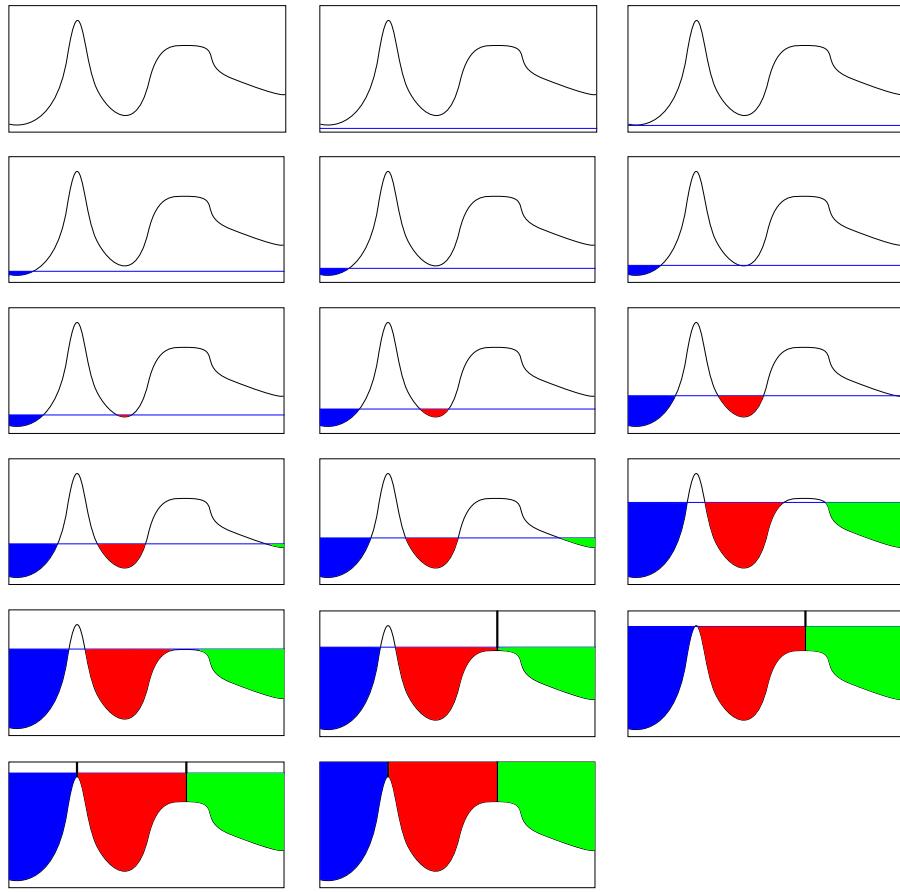


Fig. 2.4: Exemplo da intuição da simulação de inundação no perfil de uma superfície

mais consistentes com o esperado.

### 2.2.2 Imersão (Flooding-WT)

A definição de imersão [6], deste ponto em diante chamada *Flooding-WT*, busca simular o processo de submersão de uma superfície em água, baseando-se em um nível de água ascendente e zonas de influência. Tomando uma imagem  $I$ , define-se uma recursão sobre os níveis de cinza no intervalo  $(h_{min}, h_{max}]$ , onde para cada nível é aplicado um limiar sobre a imagem e são calculadas as zonas de influência do limiar atual em relação as regiões do limiar anterior. As zonas de influência, definidas como  $IZ_A(B)$ , produzem como resultado regiões onde a distância geodésica de um *pixel* em  $A$  em relação a um componente conexo de  $B$  é estritamente menor que a qualquer outro componente conexo. A recursão é inicializada com uma limiarização em  $h_{min}$ . Ao final do processo, os *pixels* que não pertencerem a nenhuma zona de influência formam a linha de *watershed* [6]. Temos então as regiões para um nível  $h$  definidas em  $X_h$ , e os limiares em  $T_h$ . Considere  $\min_h$  os pixels pertencentes a mínimos regionais no nível  $h$ .

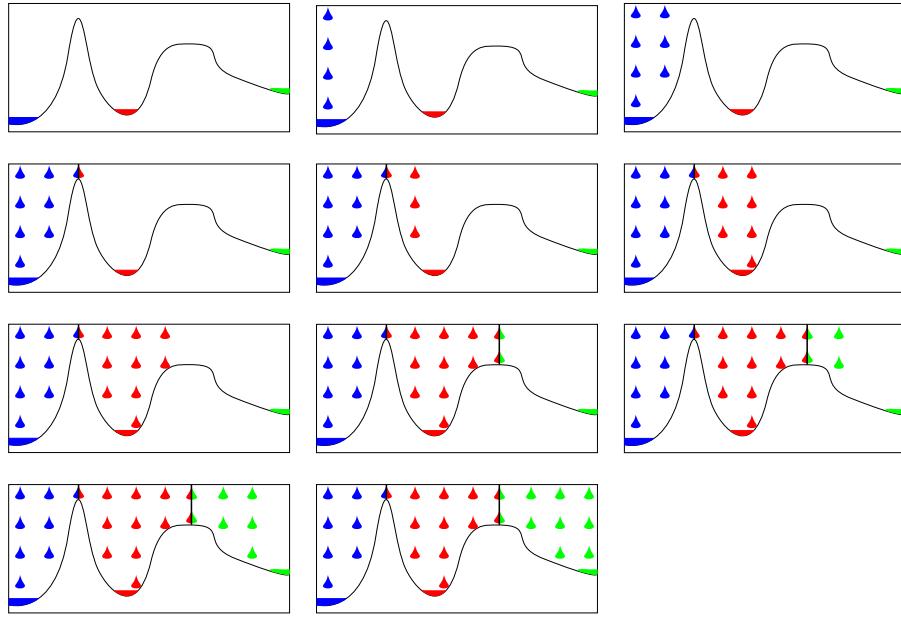


Fig. 2.5: Exemplo da intuição da simulação de chuva no perfil de uma superfície

$$X_{h_{min}} = T_{h_{min}} \quad (2.1)$$

$$\forall h \in (h_{min}, h_{max}], \quad X_h = min_h \cup IZ_{T_h}(X_{h-1}) \quad (2.2)$$

$$W = D_I \setminus X_{h_{max}} \quad (2.3)$$

Esta definição também é vista como um esqueleto de zonas de influência (SKIZ) generalizado para uma imagem em níveis de cinza. A Fig. 2.6 apresenta um exemplo da definição *Flooding-WT*. No caso da aplicação real da definição, deseja-se não somente identificar os *pixels* de *watershed*, mas também os rótulos das regiões, obtidas a partir das zonas de influência calculadas a partir dos mínimos regionais, rotulados estes unicamente na imagem. Nesta figura a definição é apresentada passo-a-passo, para todos os valores de  $h$  na imagem, calculando-se a zona de influência em relação aos níveis anteriores. Os *pixels* que não pertencerem a nenhuma zona de influência são denotados com o rótulo **W**. No entanto, deve-se ressaltar que esta rotulação não é definitiva, sendo que a zona de influência é recalculada e estes *pixels* podem passar a pertencer a outras bacias, como ocorre entre os níveis 3 e 4. O resultado para  $h = 9$  é o resultado final da definição.

### 2.2.3 Distância Topográfica (TD-WT)

A definição de distância topográfica, criada por Meyer [7], cria a noção de custo sobre uma superfície digital considerando quaisquer dois pontos nesta, considerando-se as inclinações entre cada

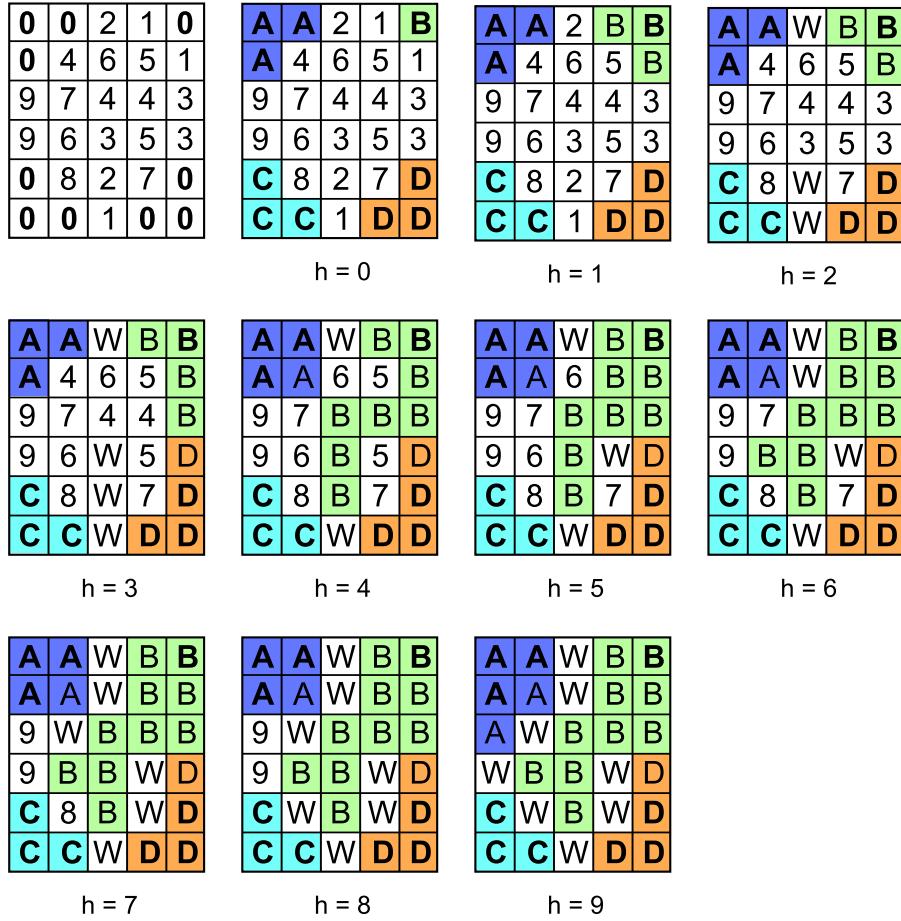


Fig. 2.6: Aplicação da definição Flooding-WT sobre uma imagem exemplo, representando o resultado em cada limiar

ponto e a distância geodésica entre estes. De fato, o nome distância topográfica é inapropriado por não se tratar de uma função do tipo distância, mas sim um custo de caminho. Desta forma, deseja-se identificar os caminhos de custo ótimo (mínimo) entre os pontos da imagem e os mínimos regionais, e assim, nos pontos ambíguos, onde o custo ótimo é igual em relação a dois mínimos diferentes, marca-se uma linha de *watershed*. A formalização desta definição inicia-se pela noção de inclinação  $LS(p)$ . Para simplificação, consideramos que a distância geodésica entre dois *pixels* adjacentes é unitária.

$$LS(p) = \max_{q \in N(p) \cup \{p\}} (I(p) - I(q)) \quad (2.4)$$

O custo para ir de um *pixel*  $p$  a um vizinho  $q$  é definido em  $cost(p, q)$ . Desta forma, temos que o custo será a rampa máxima, ou caminho de máxima inclinação, entre estes dois *pixels*.

$$cost(p, q) = \begin{cases} LS(p) & I(p) > I(q) \\ LS(q) & I(p) < I(q) \\ \frac{LS(p)+LS(q)}{2} & I(p) = I(q) \end{cases} \quad (2.5)$$

A partir desta definição, deduz-se a distância topográfica sobre um caminho  $\pi = \langle p_0, \dots, p_n \rangle$  qualquer como a soma dos custos entre os *pixels* adjacentes neste caminho. A distância topográfica,  $T_I(p, q)$ , entre dois *pixels*  $p = p_0$  e  $q = p_n$  é dada então como a menor distância de todos os caminhos possíveis.

$$T_I(p, q) = \min_{\pi \in [p \leadsto q]} \left( \sum_{i=0}^{n-1} cost(p_i, p_{i+1}) \right) \quad (2.6)$$

Pode-se então definir as bacias de captação dos mínimos regionais da imagem, onde estes são denotados por  $m_i \in M$  sendo  $M$  o conjunto de todos os mínimos, como a minimização da função de distância topográfica entre qualquer  $p$  e os mínimos.

$$CB(m_i) = \{p \in D \mid \forall m_j \in M \setminus \{m_i\} : I(m_i) + T_I(p, m_i) < I(m_j) + T_I(p, m_j)\} \quad (2.7)$$

As linhas de *watershed* são obtidas pelo complemento das bacias, nos *pixels* onde a distância topográfica não é estritamente menor entre dois ou mais mínimos regionais diferentes.

$$W(I) = I \setminus \bigcup_{m_i \in M} CB(m_i) \quad (2.8)$$

Uma definição derivada da distância topográfica, muito importante e útil na compreensão dos algoritmos de *watershed* e sua relação com as definições, é o *downstream*. O *downstream*, representado por  $\Gamma$ , é uma restrição sobre a relação de vizinhança  $N(p)$ , dada por um conjunto onde pertencem apenas aqueles vizinhos com nível de cinza menor que  $p$  e para os quais  $LS(p)$  é máximo, ou seja, aqueles com menor nível de cinza.

$$\Gamma(p) = \{q \in N(p) \mid I(q) < I(p) \wedge I(q) = \min_{\forall u \in N(p)} I(u)\} \quad (2.9)$$

A relação  $\Gamma$  contém os vizinhos de  $p$  para onde segue um caminho de máxima inclinação, sendo o fundamento de diversos algoritmos que realizam esta análise da vizinhança. O inverso desta,  $\Gamma^{-1}$ , chamada de *upstream* define os vizinhos de  $p$  em que este está contido em  $\Gamma$ . Seu cálculo exige a verificação das relações  $\Gamma$  de uma vizinhança expandida nos vizinhos dos vizinhos de  $p$ .

$$\Gamma^{-1}(p) = \{q \in N(p) \mid p \in \Gamma(q)\} \quad (2.10)$$

A Fig. 2.7 apresenta um exemplo da definição TD-WT. Esta definição possui solução única, e pode ser obtida calculando-se os caminhos e distâncias topográficas ou a partir da relação  $\Gamma$ , aplicando rótulo  $\mathbf{W}$  quando os caminhos de máxima inclinação indicados por esta levarem a dois ou mais mínimos regionais diferentes. Na Fig. 2.7 optou-se por apresentar os passos de análise dos caminhos da relação  $\Gamma$ , verificando para cada *pixel* os caminhos que partem deste e denotando-os com setas em vermelho e mantendo as setas já analisadas em verde. Ao analisar os caminhos, aplica-se o rótulo obtido em todos os *pixels* deste. Nesta mesma figura, o segundo quadro apresentado corresponde ao LCG (*Lower Complete Graph*), que apresenta na forma de um grafo o *downstream* de todos os *pixels* [21].

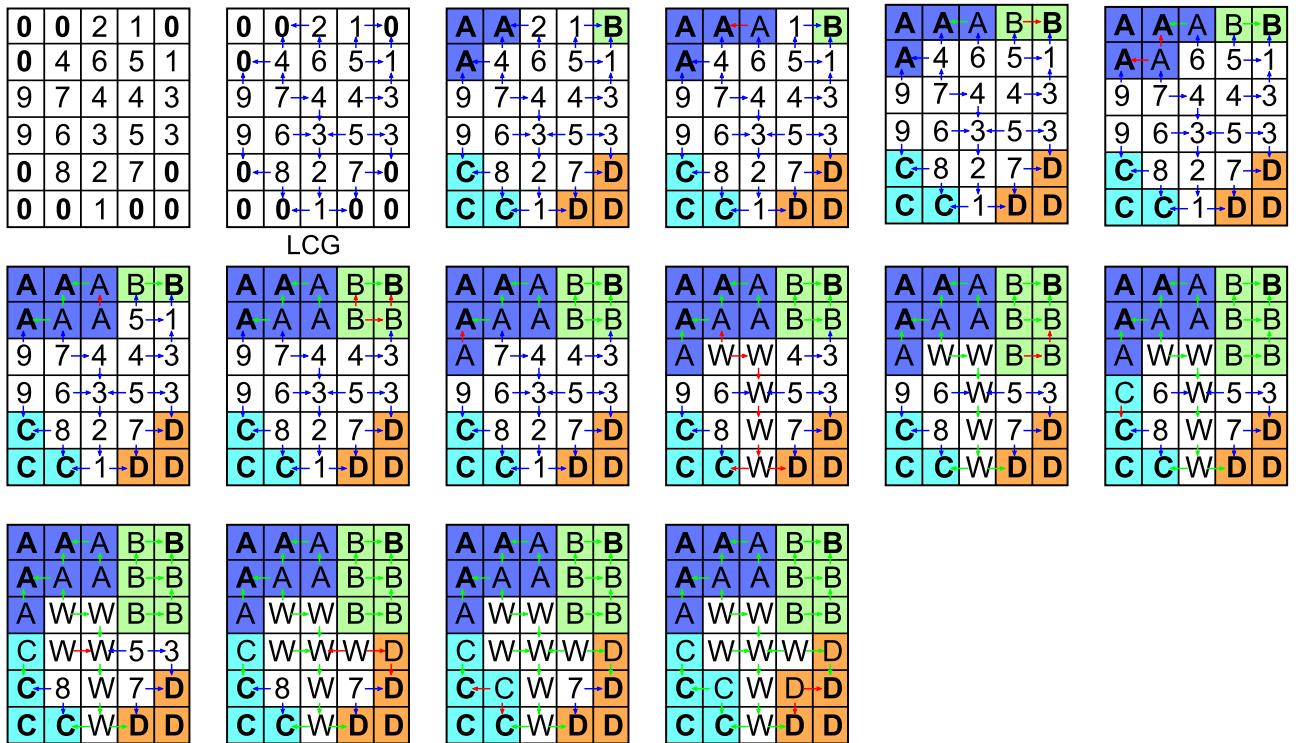


Fig. 2.7: Aplicação da definição TD-WT sobre o grafo LCG de uma imagem exemplo, representando o resultado em cada passo verificando-se as arestas para cada *pixel*

## Zonas Planas

Um aspecto da definição TD-WT se dá em relação às zonas planas das imagens, onde o custo da função é zerado. Este comportamento é justificado, visto que o deslocamento nesta superfície tem o mesmo custo em todas as direções. A solução apontada para este problema é o uso de um algoritmo de remoção de zonas planas, onde todos os *pixels* que não pertencerem a um mínimo regional passam a ter pelo menos um vizinho com nível de cinza inferior. Este processo é chamado de *lower completion*, sendo definido com relação às distâncias geodésicas das bordas das zonas planas. A maior distância encontrada na imagem serve como parâmetro para recalcular os valores dos *pixels*, em conjunto com

distâncias da borda e o nível de cinza original. Desta forma, o Alg. 1 apresenta uma técnica simples para processamento do *lower completion*. Este algoritmo, baseado na proposta de Roerdink e Meijster [12], foi corrigido para tratamento correto de mínimos regionais.

---

**ALGORITMO 1: Lower Completion**


---

**Entrada:**  $im$ : Imagem de níveis de cinza

**Saida:**  $lc$ : Imagem lower-complete

```

1: Initialise
2:   FICTITIOUS-PIXEL  $\leftarrow (-1, -1)$ 
3:   cur_dist  $\leftarrow 0$ 
4: End

5: for all  $p \in D$  do
6:    $lc(p) \leftarrow 0$ 
7:   if  $\exists q \in N(p) \mid im(q) < im(p)$  then
8:     QUEUEPUSH( $p$ )
9:      $lc(p) \leftarrow -1$ 
10:    end if
11: end for

12: cur_dist  $\leftarrow 1$ 
13: QUEUEPUSH(FICTITIOUS-PIXEL)

14: while QUEUEEMPTY() = false do
15:    $p \leftarrow$  QUEUEPOP()
16:   if  $p =$  FICTITIOUS-PIXEL then
17:     if QUEUEEMPTY() = false then
18:       QUEUEPUSH(FICTITIOUS-PIXEL)
19:       cur_dist  $\leftarrow$  cur_dist + 1
20:     end if
21:   else
22:      $lc(p) \leftarrow$  cur_dist
23:     for all  $q \in N(p) \mid im(q) = im(p)$  and  $lc(q) = 0$  do
24:       QUEUEPUSH( $q$ )
25:        $lc(q) \leftarrow -1$ 
26:     end for
27:   end if
28: end while

29: for all  $p \in D$  do
30:   if  $lc(p) = 0$  then
```

```

31:     lc(p) ← cur_dist × im(p)
32:     else
33:         lc(p) ← cur_dist × im(p) + lc(p) - 1
34:     end if
35: end for

```

Devido à complexidade da definição TD-WT em termos de cálculo de custos, normalmente realizado como uma floresta de caminhos mínimos, a opção de implementação mais comum desta é através da relação  $\Gamma$  com procedimentos especiais para propagação em zonas planas. Desta forma, não é utilizado pré-processamento, e em certa medida, a análise é feita apenas nas regiões onde é de fato necessária. A Fig. 2.8 apresenta em (a) um exemplo de imagem onde calcula-se o *lower completion*. Neste caso, a maior distância de borda tem valor 6, sendo o valor de referência utilizado para multiplicar pelos valores dos *pixels* e serem somados às distâncias individuais das bordas. O resultado da remoção de zonas planas é apresentado na Fig. 2.8 (b).

3 3 3 3 3 3 1 1 1 1	23 22 21 20 19 18 8 7 6 7
3 3 4 3 3 3 1 1 0 1	22 23 24 20 19 18 7 6 0 6
3 4 4 4 3 3 1 1 1 1	21 24 25 24 19 18 8 7 6 7
3 3 4 3 3 3 1 1 1 1	20 20 24 20 19 18 9 8 7 8
3 3 3 3 3 3 1 1 1 1	19 19 19 19 19 18 10 9 8 9
3 3 3 3 3 3 1 1 1 1	18 18 18 18 18 18 10 9 8 9
1 1 1 1 1 1 1 1 1 1	8 7 8 9 10 10 9 8 7 8
1 1 1 1 1 1 1 1 1 1	7 6 7 8 9 9 8 7 6 7
1 0 1 1 1 1 1 1 0 1	6 0 6 7 8 8 7 6 0 6
1 1 1 1 1 1 1 1 1 1	7 6 7 8 9 9 8 7 6 7

(a) Imagem com zonas planas

(b) Imagem após

Fig. 2.8: Aplicação do lower completion sobre uma imagem exemplo com zonas planas. Representação dos níveis de cinza de acordo com os valores da imagem em escala [0,25].

## 2.2.4 Condição Local (LC-WT)

Dada a característica da definição TD-WT realizar uma otimização global dos caminhos possíveis na superfície formada pela imagem, seu processamento em blocos paralelos requer diversos passos de sincronização. Buscando simplificar o projeto de um algoritmo paralelo [30], Bieniek e Moga removem a condição de unicidade da solução da definição TD-WT, degenerando-a na definição de condição local, onde apenas informações da vizinhança do *pixel* são utilizadas para determinação do caminho de máxima inclinação a que este pertence [18]. Assim, na definição LC-WT, os *pixels* de

*watershed*, que garantiam a unicidade e resolução de ambiguidades, passam a ser rotulados conforme um dos mínimos em empate.

No entanto, esta alteração tem mais implicações na relação  $\Gamma(p)$ , que passa a ser dada como um caminho único no LCG. A escolha de qual *pixel* vizinho determinará este rótulo é arbitrária dentro do conjunto de opções válidas, sendo todas soluções possíveis. De forma simplificada,  $\Gamma$  passa a conter o *pixel* vizinho com menor valor de cinza e menor que o próprio sendo analisado, ou em caso de empate, um destes, decidido arbitrariamente. As regiões passam a ser definidas pela propagação dos rótulos dos mínimos regionais através dos seguintes princípios, onde  $L$  corresponde ao rótulo da região:

1.  $L(m_i) \neq L(m_j), \forall i \neq j$ , com  $m_k$  sendo os mínimos regionais
2. Para cada *pixel*  $p$  com  $\Gamma(p) \neq \emptyset, \exists q \in \Gamma(p)$  com  $L(p) = L(q)$

A Fig. 2.9 apresenta um exemplo das possíveis soluções para a imagem utilizada. Dado que a definição LC-WT é baseada na definição TD-WT, modificam-se apenas os *pixels* de *watershed*, que passam a receber o rótulo de um de seus vizinhos conectados no LCG. A definição TD-WT também pode ser vista como a transformada zona de empate da LC-WT, onde as múltiplas soluções são indicadas pelos *pixels* de *watershed* [21]. Assim, ao avaliar os caminhos a partir de cada *pixel*, quando estes levam a dois ou mais mínimos regionais diferentes, escolhe-se arbitrariamente um destes, geralmente tomando como base a ordem de análise dos vizinhos, reduzindo então o número de arestas no grafo direcionado (LCG), reduzindo também a complexidade do problema.

## 2.2.5 Watershed por Transformada Imagem-Floresta (IFT-WT)

De forma similar a TD-WT, a definição de transformada *watershed* pela transformada imagem-floresta utiliza uma função de custo de caminho para obtenção dos caminhos ótimos em um grafo onde cada vértice corresponde a um *pixel* e o peso do arco é obtido pela direção em que se atravessa este, sendo utilizado o valor do *pixel* do vértice de fim. A função de custo do caminho utilizada pela IFT-WT, entretanto, é composta de duas componentes, a primeira sendo o máximo do caminho,  $f_{max}$ , e a segunda, utilizada em caso de empate na primeira, o custo lexicográfico,  $f_d$  [19], [8].

$$f_{max}(\langle v_1, v_2, \dots, v_n \rangle) = \max(I(v_2), I(v_3), \dots, I(v_n)) \quad (2.11)$$

$$f_d(\langle v_1, v_2, \dots, v_n \rangle) = \max_{k \in [0, n-1]} (k : C[v_n] = C[v_{n-k}]) \quad (2.12)$$

$$C[v_n] = f_{max}(\langle v_1, v_2, \dots, v_n \rangle) \quad (2.13)$$

A primeira componente, de maior importância, simula a inundação de uma superfície, enquanto a segunda faz com que a inundação em superfícies planas ocorra a mesma velocidade a partir das bordas, vindas de mínimos diferentes. A componente  $f_d$  representa a maior distância em um caminho

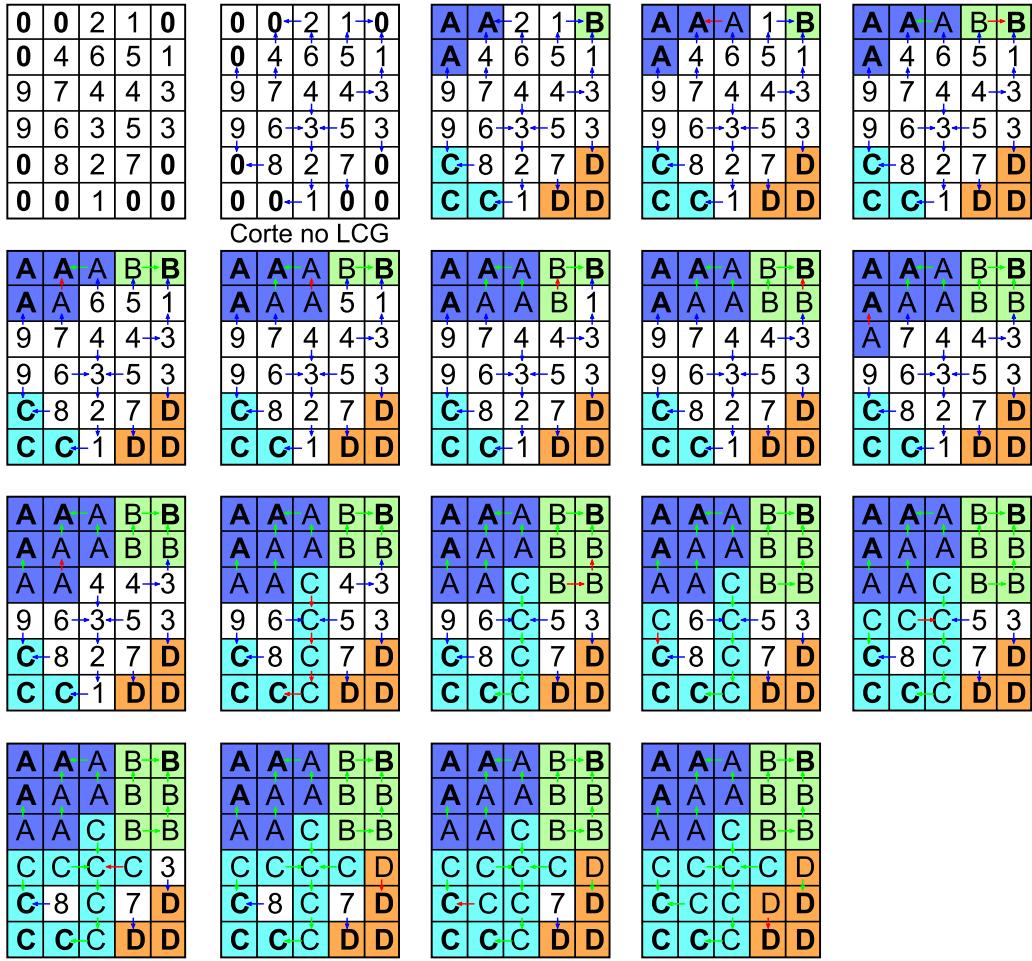
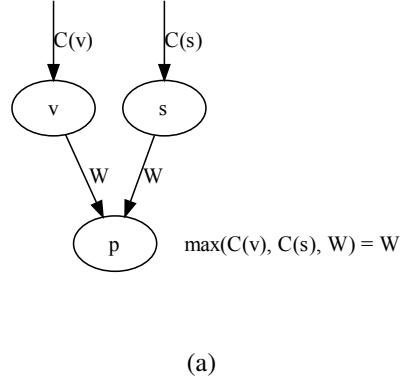


Fig. 2.9: Aplicação da definição LC-WT sobre um corte do grafo LCG, feito aleatoriamente para remoção de soluções múltiplas, de uma imagem exemplo, representando o resultado em cada passo verificando-se as arestas para cada pixel,

onde o valor de máximo se mantém, ou seja, a distância até a borda da zona plana, onde ocorre o empate na primeira componente, ou seja, busca-se maximizar  $k$  que satisfaça a condição  $C[v_n] = C[v_{n-k}]$ . Desta forma, definido o custo do caminho, aplica-se este na transformada imagem-floresta (IFT), utilizando como sementes - raízes da floresta do grafo - os mínimos regionais ou marcadores arbitrários. O resultado desta é uma floresta de caminhos mínimos, onde o custo é o máximo dos arcos. Pode-se dizer então que o resultado é uma minimização de custos máximos.

A definição IFT-WT não apresenta solução única, nem utiliza rótulos *watershed* para representação de pontos distantes igualmente de dois mínimos regionais. Esta é uma consequência do uso do custo máximo, conforme visto na Ref. [19], pois os máximos locais em um grafo podem ser atingidos - e por conseguinte rotulados - por qualquer um de seus vizinhos em empate tanto em custo máximo quanto lexicográfico. A Fig. 2.10 exemplifica este processo, assumindo que  $W > C(v)$  e  $W > C(s)$ . O custo final atribuído a  $p$  será  $W$ , independente de seu antecessor na floresta ser  $v$  ou  $s$ , sendo um custo ótimo, porém possivelmente com rótulos diferentes.



(a)

Fig. 2.10: Grafo exemplificando o empate de custos considerando-se  $W$  o máximo em um caminho terminado no pixel  $p$

Como consequência, temos que as soluções da definição IFT-WT formam um conjunto, denominado  $\Phi$ . Inseridas neste conjunto, todas as soluções são válidas e ótimas do ponto de vista da função de custo definida anteriormente. A Fig. 2.11 apresenta um exemplo de solução para a imagem demonstrativa. Nesta solução, parte-se do grafo que contém todas as soluções de  $\Phi$ , denominado MOG (*Multipredecessor Optimal Graph*), e, a partir das sementes iniciais, verifica-se seus vizinhos, determinando rótulos para estes ou realizando cortes no grafo, para aplicação posterior do rótulo. O MOG, assim como o LCG é construído a partir de uma imagem sem zonas planas, sendo que no MOG, todos os *pixels* vizinhos com nível de cinza inferior pertencem ao conjunto de antecessores de um vértice, enquanto no LCG apenas os vizinhos com nível de cinza mínimo e inferior na vizinhança são antecessores [21]. Procede-se desta forma, expandindo as regiões já rotuladas até que todos os *pixels* sejam analisados, sendo o resultado final das setas em verde formadoras da floresta ótima.

### Zona de Empate (TZ-IFT-WT)

A zona de empate de uma definição com soluções múltiplas qualquer pode ser dita como a solução que unifica as outras, atribuindo um rótulo especial onde há diferenças entre as soluções possíveis como no caso da definição TD-WT em relação a LC-WT. Assim, no caso da IFT-WT, a zona de empate é baseada no conjunto  $\Phi$ , que contém todas as soluções ótimas. Para uma região  $CB_i$ , um *pixel*  $v$  passa a pertencer a esta se e somente se pertencer a esta em todas as soluções possíveis. Em outras palavras, é necessário que em todas as soluções exista um caminho entre o conjunto de sementes  $s_i$  e o *pixel*  $v$  em análise. Desta forma, os *pixels* que não pertencerem a nenhuma região serão contidos em  $T$ , formando a zona de empate [20].

$$CB_i = \{v \in V, \forall F \in \Phi, \exists \pi(s_i, v) \text{ in } F\} \quad (2.14)$$

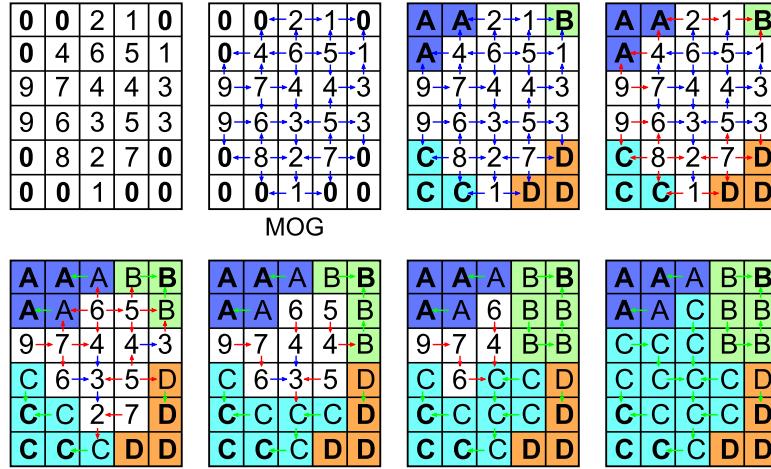


Fig. 2.11: Aplicação da definição IFT-WT sobre o grafo MOG de uma imagem exemplo, representando o resultado em cada passo pela propagação do rótulo para os vizinhos conectados pelo MOG de cada *pixel* já rotulado, resolvendo empates aleatoriamente

$$T = V \setminus \bigcup_i CB_i \quad (2.15)$$

Os *pixels* contidos em  $T$  são representados na imagem resultante com um rótulo especial **TZ**, cujo significado é de permitir mais de uma solução, não devendo ser visto como uma linha divisória entre as regiões. A unificação entre as soluções da IFT-WT permite diversas análises, como a robustez da segmentação, afinamento das linhas, extensão máxima de objetos, além de estabelecer relacionamentos com as outras definições [31], [21], [14]. A Fig. 2.12 apresenta a TZ-IFT-WT da imagem demonstrativa. Assim como no exemplo da IFT-WT, inicia-se pelo MOG e pelos mínimos regionais, porém neste caso, as setas vermelhas indicam os rótulos vizinhos dos *pixels* que estão sendo analisados. Se estes forem consistentes - iguais - o *pixel* em questão o recebe, caso contrário, o *pixel* permite múltiplas soluções, e portanto pertence à zona de empate, e irá propagar este rótulo como outro qualquer.

## 2.2.6 Watershed Cut (WC-WT)

A definição *watershed cut* é baseada em operações de corte de grafo sobre um grafo valorado nas arestas. De modo a melhor compreender seu comportamento, inicia-se pela construção do grafo a partir da imagem. Cada *pixel* da imagem corresponderá a um vértice e será adjacente aos vértices correspondentes aos *pixels* vizinhos. Os valores das arestas podem ser obtidos pelo máximo ou mínimo dos valores entre cada par de *pixels* correspondentes aos vértices, ou pode-se utilizar a diferença absoluta entre estes, sendo a última uma forma simples de cálculo de gradiente [9]. A Fig. 2.13 apresenta em (a) a imagem demonstrativa e em (b) o grafo correspondente a esta, com os valores de aresta obtidos utilizando o mínimo entre os *pixels*.

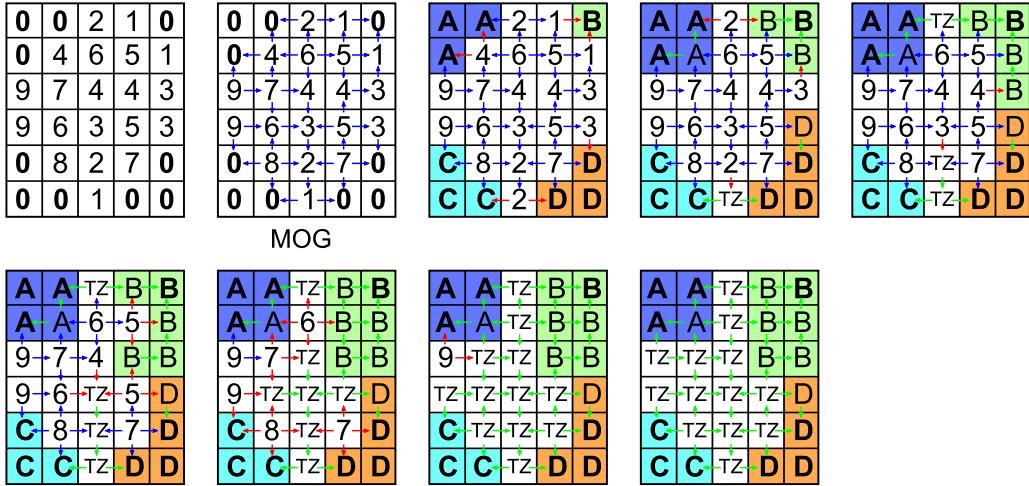
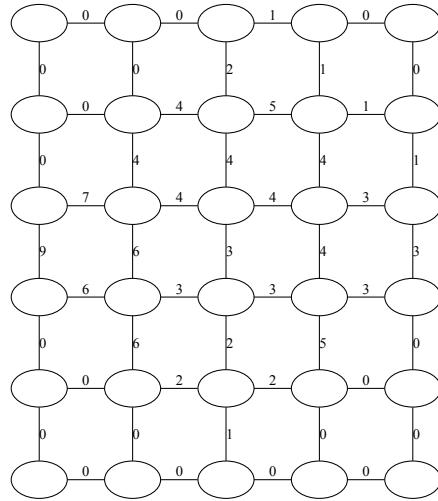


Fig. 2.12: Aplicação da definição IFT-WT sobre o grafo MOG de uma imagem exemplo, representando o resultado em cada passo pela propagação do rótulo para os vizinhos conectados pelo MOG de cada *pixel* já rotulado, aplicando rótulo TZ em caso de empate

0	0	2	1	0
0	4	6	5	1
9	7	4	4	3
9	6	3	5	3
0	8	2	7	0
0	0	1	0	0

(a) Imagem

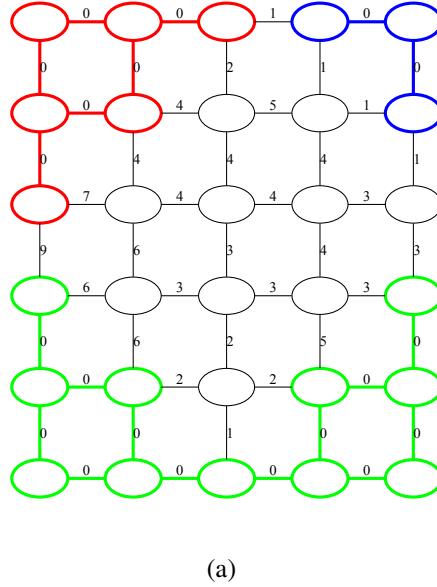


(b) Grafo correspondente

Fig. 2.13: Construção do grafo do watershed cut a partir de uma imagem exemplo, aplicando os valores mínimos dos pixels em questão de (a) nas arestas em (b)

A partir de um dado grafo, seguem duas definições importantes para a compreensão da definição de *watershed cut*. A primeira delas define um subgrafo mínimo, que corresponde a um mínimo regional. Assim, dado um grafo  $G$  com função de valores  $F$ , um subgrafo  $X$  será um mínimo de  $F$  com valor  $k$  se: (1)  $X$  é conexo, (2)  $k$  é o valor de qualquer aresta de  $X$  e (3) qualquer aresta

adjacente a  $X$  tem valor estritamente maior que  $k$  [9]. O conjunto de subgrafos mínimos em  $F$  é denotado por  $M(F)$ . Esta definição é extremamente importante no contexto da segmentação por transformada *watershed*, pois por consequência impõe o número de regiões em que a imagem será dividida. Dada sua importância, a Fig. 2.14 apresenta o grafo da Fig. 2.13 com seus subgrafos mínimos em destaque.



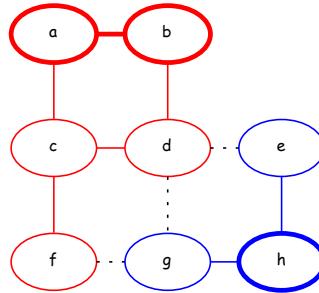
(a)

Fig. 2.14: Subgrafos mínimos em destaque, correspondentes aos mínimos regionais

Visto que são identificados três subgrafos mínimos na Fig. 2.14, logicamente serão obtidas três regiões como resultado da aplicação da definição de *watershed cut*. É importante notar a definição diferente de mínimo regional para o *watershed cut*, que pode implicar na redução do número de regiões segmentadas na imagem. Isto nos leva a segunda importante definição, de extensão, revisitada a partir da proposta original de Bertrand [32, 9]. Sendo  $X$  e  $Y$  dois subgrafos de  $G$ ,  $Y$  é uma extensão de  $X$  se  $X \subseteq Y$  e cada componente conexo de  $Y$  contém exatamente um componente conexo de  $X$ . Intuitivamente, a extensão implica em um subgrafo passar a ser adjacente de mais nós vizinhos mantendo-se desconexo de outros componentes. A Fig. 2.15 apresenta um exemplo de extensão, onde os componentes conexos são apresentados com cores diferentes, o subgrafo  $X$  é ressaltado em negrito, e arestas que pertencem a  $G$  mas não pertencem a  $Y$  são mostradas em pontilhado.

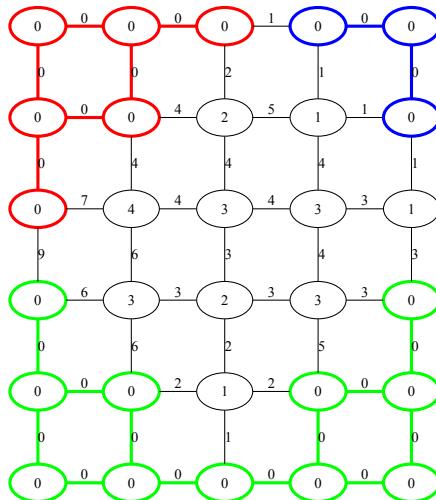
Para completar as definições necessárias ao *watershed cut*, atribui-se também um valor a cada vértice do grafo, denominado altura mínima, denotado por  $F^\ominus(x)$ , sendo o valor mínimo entre as arestas conectadas ao vértice  $x$  [9]. Podendo ser calculado no momento da construção do grafo, este valor é utilizado diversas vezes na obtenção dos conjuntos de vértices no algoritmo. A Fig. 2.16 apresenta o grafo de exemplo completo com os valores de altura mínima.

Assim, o *watershed cut* é definido sobre um grafo valorado  $G = (V, E, F)$ , sendo  $S$  um subconjunto de  $E$ , e seu complemento  $\bar{S}$ .  $S$  satisfaz o princípio da gota d'água - e por consequência é um *watershed cut* - se  $S$  for uma extensão de  $M(F)$  e: [9]



(a)

Fig. 2.15: Grafo exemplificando a extensão de componentes conexos



(a)

Fig. 2.16: Grafo de exemplo completo com subgrafos mínimos e alturas mínimas

- Para qualquer aresta  $u = \{x_0, y_0\} \in S$  há dois caminhos descendentes iniciando em  $x_0$  e  $y_0$  levando a mínimos diferentes e contidos em  $\bar{S}$
- $F(u) \geq F(\{x_0, x_1\})$  ( $F(u) \geq F(\{y_0, y_1\})$ ), se os caminhos não forem triviais ( $n(\pi) > 1$ )

Desta forma,  $S$  deverá conter todas as arestas que dividem as regiões do grafo em componentes (subgrafos) conexos. As linhas de divisão entre as regiões são definidas sobre arestas, e  $u$  é uma aresta conectando quaisquer dois pontos de divisa entre regiões. Ou seja,  $S$  é um corte em  $G$ , contendo as arestas que devem ser removidas de modo a se obter um *watershed cut*. No entanto, a definição não

fornecer uma solução única, sendo que podem existir vários conjuntos  $S$  que satisfaçam as condições acima. A Fig. 2.17 mostra em (a) o grafo após a operação de corte e em (b) o resultado da rotulação aplicada na imagem. As arestas pertencentes ao conjunto  $S$  são mostradas em pontilhado, e separam as regiões da imagem, satisfazendo o princípio da gota d'água exposto acima. Na imagem (b) não são mostrados os mínimos regionais hachurados conforme os exemplos das definições anteriores, visto que seriam todos *pixels* vizinhos, contrariando a noção de divisão em regiões de acordo com os componentes conexos mínimos, comum às outras definições.

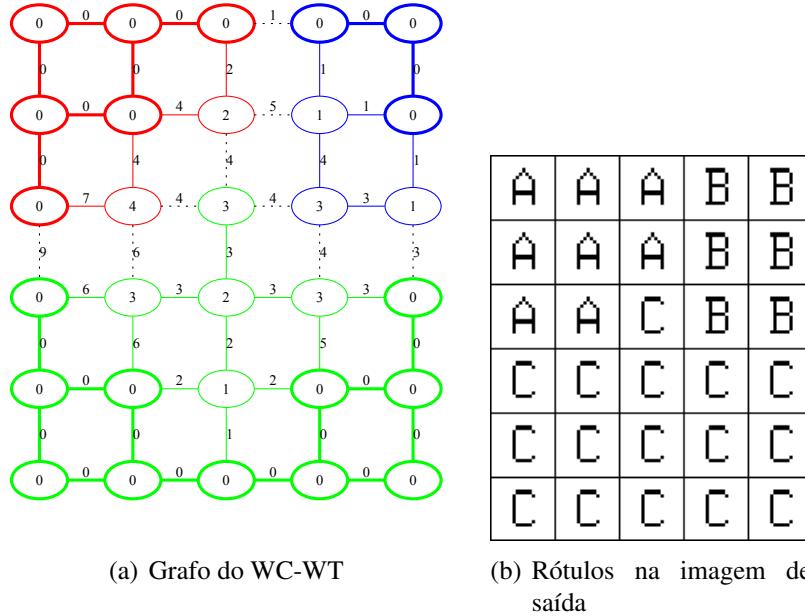


Fig. 2.17: Aplicação do WC-WT sobre grafo construído a partir de exemplo

### 2.2.7 Relações

As definições da transformada *watershed* apresentadas acima podem ser relacionadas conforme o trabalho de Audigier [13]. Para estabelecer estas relações as imagens foram modeladas utilizando dois tipos de grafo em particular, o grafo MOG e LCG. Estes grafos permitem representar as soluções possíveis para a função de custo máximo e para a relação  $\Gamma$ . É importante também compreender o sentido de uma transformada zona de empate: através desta se obtém uma solução única para uma definição que fornece múltiplas soluções. A partir destes conceitos pode-se relacionar as definições TD-WT, LC-WT, IFT-WT e TZ-IFT-WT. As principais relações estabelecidas são [21]:

1. Qualquer LC-WT é também um IFT-WT.
2. TD-WT é a transformada zona de empate de LC-WT.
3. As regiões definidas por TZ-IFT-WT são um subconjunto das regiões correspondentes de TD-WT.

Pode-se a partir destes itens, considerando-se apenas as regiões definidas pelas transformadas, organizar as informações de relacionamentos em um gráfico, apresentado na Fig. 2.18. A transformação por zona de empate apresentada no gráfico implica que *pixels*, que podem assumir diferentes rótulos dada uma mesma definição, passam a receber um rótulo especial, como o rótulo **W** e **TZ** para as definições TD-WT e TZ-IFT-WT respectivamente. Neste gráfico não são incluídas as definições Flooding-WT e WC-WT, pois, no caso da primeira, não há relações estabelecidas, e no caso da segunda, esta é definida como uma operação de corte de grafo, possuindo relações próprias, onde ocorrem equivalências entre as definições [33].

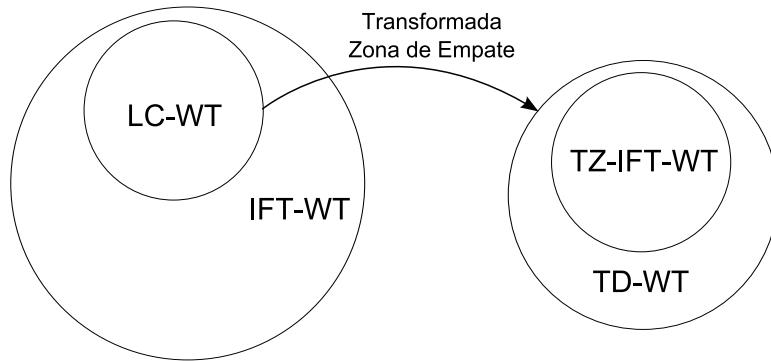


Fig. 2.18: Gráfico indicando os relacionamentos entre as definições de transformada watershed a respeito das bacias hidrográficas

Os algoritmos apresentados no Cap. 3 são todos relacionados com uma das definições da transformada *watershed* apresentadas aqui. Desta forma, pode-se estender o gráfico dos relacionamentos diretamente à estes. No próximo capítulo são detalhados os algoritmos de transformada *watershed* disponíveis na literatura, apresentando-os no contexto das definições formais, e ressaltando suas características principais.

# Capítulo 3

## Algoritmos de Watershed

Neste capítulo são avaliados 14 algoritmos de transformada *watershed* disponíveis na literatura. Esta revisão busca uniformizar as informações sobre estes, como sua motivação para criação, qual definição implementa, corretamente ou não, funcionamento geral, caracterização como busca em largura ou profundidade, detalhes específicos de implementação, além da descrição dos algoritmos utilizando pseudocódigo com notação similar entre estes, facilitando a compreensão. Os algoritmos são detalhados por ordem de aparição na literatura, a partir das primeiras transformadas rápidas conhecidas. Com esta revisão do estado da arte procura-se reunir as técnicas utilizadas de forma a facilitar a compreensão de suas implicações e usos.

### 3.1 Algoritmo Vincent e Soille de Imersão

O algoritmo de imersão, proposto por Vincent e Soille, é a primeira transformada *watershed* rápida conhecida na literatura, tendo seus princípios estabelecidos pelo paradigma de busca em largura com resultados determinados pela definição Flooding-WT [6]. Sua implementação é baseada em um algoritmo de quatro passos. O primeiro passo é uma ordenação dos *pixels* pelo seu nível de cinza. Este passo é fundamental para determinar a velocidade de execução do algoritmo, pois permite o acesso direto aos *pixels* no mesmo nível, no entanto pode-se utilizar implementações relativamente simples que garantem esta mesma propriedade, como o uso de dicionários. Em seguida, itera-se pelos *pixels* nos níveis de cinza na imagem, mascarando-os e inserindo em uma estrutura de dados aqueles com vizinhos rotulados. A análise dos *pixels* na estrutura constitui o terceiro passo, onde os rótulos são determinados e propagados nas zonas planas. Por último, *pixels* que não foram rotulados constituem novos mínimos regionais, e assim seus componentes conexos recebem novos rótulos. O Alg. 2 apresenta a imersão de Vincent e Soille.

---

#### ALGORITMO 2: Imersão

---

**Entrada:** *im*: Imagem de níveis de cinza

**Saida:** *lab*: Imagem de rótulos

1: **Initialise**

```

2: MASK ← -2
3: INIT ← -1
4: WSHED ← 0
5: FICTITIOUS-PIXEL ← -1
6:  $\forall p \in D, \text{lab}(p) \leftarrow \text{INIT}$ 
7: basins ← 0
8: cur_dist ← 0 // Imagem de trabalho, para armazenar distâncias
   geodésicas
9:  $\forall p \in D, \text{dist}(p) \leftarrow 0$ 
10: End

// 1º Passo
11: Ordene os pixels por seu nível de cinza, com mínimo  $h_{min}$  e máximo  $h_{max}$ 

// 2º Passo
12: for  $h \in [h_{min}, h_{max}]$  do // Imersão em cada nível de cinza
13:   for all  $p \in D \mid \text{im}(p) = h$  do
14:     lab( $p$ ) ← MASK // Rotula como não processado os pixels neste
   nível
15:     if  $\exists q \in N(p) \mid (\text{lab}(q) > 0 \text{ or } \text{lab}(q) = \text{WSHED})$  then
16:       dist( $p$ ) ← 1 // Inicializa a distância para o pixel ...
17:       QUEUEPUSH( $p$ ) // ... e usa a fila para processá-lo
18:     end if
19:   end for

20:   cur_dist ← 1
21:   QUEUEPUSH(FICTITIOUS-PIXEL)

// 3º Passo
22: while true do
23:   p ← QUEUEPOP()
24:   if  $p = \text{FICTITIOUS-PIXEL}$  then // Processou todos os pixels deste
   nível de cinza a esta distância
25:     if QUEUEEMPTY() = true then // Verifica a condição de parada
26:       break
27:     else
28:       QUEUEPUSH(FICTITIOUS-PIXEL) // Reinsere a condição de
   parada
29:       cur_dist ← cur_dist + 1 // Após processar todos os pixels
   vizinhos imediatos de regiões já rotuladas (linha 17), aumenta a
   distância geodésica
30:     p ← QUEUEPOP()

```

```

31:         end if
32:     end if
33:     for  $q \in N(p)$  do
34:         if  $\text{dist}(q) < \text{cur\_dist}$  and ( $\text{lab}(q) > 0$  or  $\text{lab}(q) = \text{WSHED}$ ) then // Verifica se o vizinho está rotulado e se a distância geodésica é menor que a atual (mais próximo da descida do plateau, processado previamente)
35:             if  $\text{lab}(q) > 0$  then
36:                 if  $\text{lab}(p) = \text{MASK}$  or  $\text{lab}(p) = \text{WSHED}$  then // Se o pixel ainda não foi processado ou pertence a watershed, atribui o label do mínimo, isto afina a linha de watershed
37:                      $\text{lab}(p) \leftarrow \text{lab}(q)$ 
38:                 else if  $\text{lab}(p) \neq \text{lab}(q)$  then // Se os rótulos forem diferentes (dois mínimos atingiram um ponto) o pixel é de watershed
39:                      $\text{lab}(p) \leftarrow \text{WSHED}$ 
40:                 end if
41:             else if  $\text{lab}(p) = \text{MASK}$  then
42:                  $\text{lab}(p) \leftarrow \text{WSHED}$  // Propagação do rótulo watershed
43:             end if
44:             else if  $\text{lab}(q) = \text{MASK}$  and  $\text{dist}(q) = 0$  then // Propaga o processamento dentro de um plateau
45:                  $\text{dist}(q) \leftarrow \text{cur\_dist} + 1$ 
46:                 QUEUEPUSH( $q$ )
47:             end if
48:         end for
49:     end while

    // 4º Passo
50:     for  $p \in D$  |  $\text{im}(p) = h$  do // Novos mínimos não são enfileirados, são processados à parte
51:          $\text{dist}(p) \leftarrow 0$ 
52:         if  $\text{lab}(p) = \text{MASK}$  then
53:             basins  $\leftarrow$  basins + 1
54:             QUEUEPUSH( $p$ )
55:              $\text{lab}(p) \leftarrow \text{basins}$  // Propaga o label para os pixels vizinhos não processados usando a fila
56:             while QUEUEEMPTY() = false do
57:                  $q \leftarrow \text{QUEUEPOP}()$ 
58:                 for all  $u \in N(q)$  |  $\text{lab}(u) = \text{MASK}$  do
59:                     QUEUEPUSH( $u$ )
60:                      $\text{lab}(u) \leftarrow \text{basins}$ 
61:                 end for

```

```

62:      end while
63:      end if
64:    end for
65: end for

```

Para implementar estes passos, o algoritmo utiliza uma estrutura de dados FIFO, onde o primeiro elemento inserido será o primeiro elemento a ser removido, correspondendo às funções de estrutura de fila apresentadas no Cap. 2. Além das imagens de entrada e saída, também utiliza-se uma imagem de trabalho, onde são armazenadas as distâncias geodésicas. O processo de análise dos *pixels* enfileirados se inicia por uma verificação de condição de parada, representada por um *pixel* fictício, cujo significado é indicar que todos *pixels* a uma determinada distância geodésica a partir da borda da zona plana já foram processados. No caso de a fila estar vazia, o processamento desta deve ser interrompido, caso contrário, os *pixels* seguintes devem ser processados após incrementar a distância atual. Após esta verificação, avalia-se a vizinhança do *pixel* e decide-se qual será seu rótulo, além de inserir na fila para serem processados os *pixels* vizinhos no mesmo nível de cinza e que não estiverem nesta. Entretanto, os resultados desta implementação podem não condizer com o esperado, de acordo com a definição. A Fig. 3.1 apresenta um contra-exemplo simples, onde em (a) é apresentada uma imagem com dois mínimos regionais hachurados, sendo que em (c) a linha produzida pelo algoritmo é afinada em relação a linha da definição vista em (b). Os resultados nestes exemplos foram obtidos utilizando vizinhança-8. Este problema é conhecido na literatura, e pode ser visto também nas Refs. [12, 31].

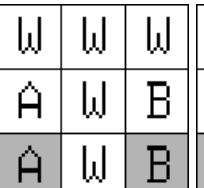
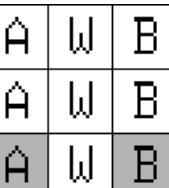
		
(a)	(b)	(c)

Fig. 3.1: Problema de aderência a definição Flooding-WT pelo algoritmo Imersão. (a) Imagem, (b) resultado da definição Flooding-WT, (c) resultado do algoritmo Imersão (N8)

Além desta questão, o algoritmo de Vincent e Soille nem sempre irá produzir linhas dividindo as regiões segmentadas, sendo uma consequência da definição adotada utilizando zonas de influência como critério. Assim, pode-se exemplificar esta condição com uma região plana onde as distâncias dos pixels aos mínimos regionais correspondentes são estritamente menores umas das outras, ou seja, as zonas de influência conterão todos os pixels desta região e não haverá linha divisória. A Fig. 3.2 apresenta um exemplo de acordo com o descrito.

Outro ponto importante a ser considerado neste algoritmo é referente à unicidade das soluções produzidas. A definição Flooding-WT produz uma solução única, mas como visto na Fig. 3.1, a implementação do algoritmo não adere à definição. Desta forma, uma das consequências é o fato de

	(a)		(b)
--	-----	--	-----

Fig. 3.2: Exemplo de inexistência de linha divisória na imersão de Vincent e Soille. (a) Imagem, (b) resultado da definição Flooding-WT e algoritmo Imersão (N4)

o resultado do algoritmo ser dependente da ordem de visitação dos pixels, especialmente na relação de vizinhança estabelecida. A Fig. 3.3 apresenta um exemplo onde a imagem foi lida em varredura *raster* e *anti-raster* - correspondente a uma rotação de  $180^\circ$  - e o algoritmo executado sobre elas. Desconsiderando-se diferenças entre rótulos, resultado da ordem de descoberta dos mínimos regionais estar invertida, a importância deste exemplo está nos *pixels* que recebem dois rótulos diferentes, em especial o *pixel* de valor 7 e posição (2,3).

	(a)		(b)		(c)
--	-----	--	-----	--	-----

Fig. 3.3: Divergência de soluções do algoritmo Imersão. (a) Imagem, (b) varredura raster, (c) varredura anti-raster (N4)

Esta diferença de rotulação se deve às regras adotadas para afinar a linha de *watershed*, que permite aos *pixels* rotulados como *watershed* serem rotulados novamente na avaliação de sua vizinhança. A Fig. 3.4 apresenta a forma de rotulação do *pixel* de valor 7 na imagem (a) da Fig. 3.3. Considerando-se a sequência de leitura da vizinhança apresentada na Fig. 3.4 (a) e (c), o primeiro caso visitaria os *pixels* na ordem **W, B, B, A** e o segundo caso na ordem **A, B, W, A**. Desta forma, no primeiro, conforme o algoritmo, o rótulo atribuído será **W**, e no segundo o rótulo será **A**, pois o rótulo **W** atribuído inicialmente será alterado na análise do último vizinho. Esta política, que permite a alteração de um rótulo **W** - responsável por afinar as linhas - tem como consequência o problema de soluções múltiplas dependentes da ordem de análise dos vizinhos.

A solução dos problemas citados acima - aderência a definição, posicionamento da linha de divisão e unicidade de resultado - não é trivial. No entanto, pode-se modificar o algoritmo de modo a solucionar o segundo problema e garantir a existência de linhas separando as bacias de captação da imagem. Esta solução, adotada por ferramentas de processamento de imagens como ImageJ [34] e SDC Morphology Toolbox [17], tem de forma geral, efeitos benéficos, pois garante a separação não apenas por rótulos mas também por contornos e, avaliando-se qualitativamente os resultados, é próxima do original, mantendo a essência da segmentação por imersão. As modificações efetuadas no Alg. 2 dizem respeito a avaliação da distância geodésica, que deixa de ser estritamente menor

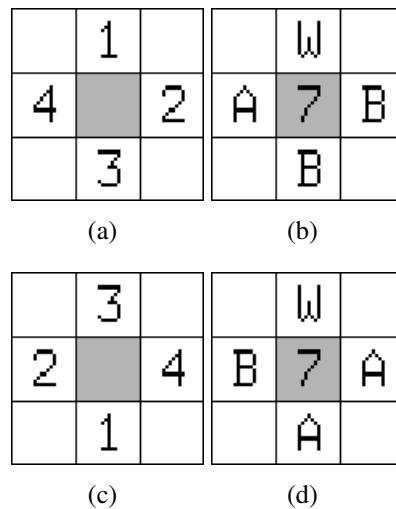


Fig. 3.4: Análise de rótulos no algoritmo Imersão. (a) Ordem de visitação em varredura raster, (b) rótulos para (a), (c) ordem de visitação em varredura anti-raster, (d) rótulos para (c)

para menor ou igual (linha 34) e a propagação do rótulo **WSHED** (linha 36), condição que é retirada. Deve-se ressaltar que estas alterações não tornam o algoritmo aderente a definição, e assim, seus resultados continuam a divergir do esperado e também não são únicos para uma mesma imagem.

### 3.1.1 Implementação Python

```
from common import *

#constants
MASK = -2
INIT = -1
WSHED = 0
FICTITIOUS\_PIXEL = -1

def immersion(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = INIT
    basins = 0
    cur\_dist = 0

    dist = ws.makeWorkCopy(0)
```

```

queue = wsQueue()

# "sorting"
levels = dict()
for p in D:

    if levels.has_key(im[p]):
        levels[im[p]].append(p)
    else:
        levels[im[p]] = [p]

# watershed
H = sorted(levels.keys())
for h in H:
    for p in levels[h]:
        lab[p] = MASK
        for q in N(p):

            if lab[q] > 0 or lab[q] == WSHED:
                dist[p] = 1
                queue.push(p)
                break

        cur\_dist = 1
        queue.push(FICTITIOUS\_PIXEL)

    while True:

        p = queue.pop()
        if p == FICTITIOUS\_PIXEL:
            if queue.empty():
                break
            else:
                queue.push(FICTITIOUS\_PIXEL)
                cur\_dist += 1
                p = queue.pop()

        for q in N(p):

            if dist[q] < cur\_dist and (lab[q] > 0 or lab[q] == WSHED):
                if lab[q] > 0:
                    if lab[p] == MASK or lab[p] == WSHED:
                        lab[p] = lab[q]
                    elif lab[p] != lab[q]:
                        lab[p] = WSHED
                    elif lab[p] == MASK:
                        lab[p] = WSHED
                elif lab[q] == MASK and dist[q] == 0:
                    dist[q] = cur\_dist + 1
                    queue.push(q)

```

```

for p in levels[h]:
    dist[p] = 0
    if lab[p] == MASK:
        basins += 1
        queue.push(p)
        lab[p] = basins
    while not queue.empty():
        q = queue.pop()
        for u in N(q):
            if lab[u] != MASK:
                continue

            queue.push(u)
            lab[u] = basins

return ws.end()

```

## 3.2 Algoritmo Fila de Prioridade Beucher e Meyer

O algoritmo de fila de prioridade para a transformada *watershed* foi proposto por Beucher e Meyer [11] como um algoritmo simples de simulação de inundação. Mais tarde, foi mostrado que o uso de filas de prioridade produz uma segmentação ótima considerando a função de máximo como custo do caminho entre cada *pixel* e os mínimos regionais [19]. Além disso, o uso da política FIFO de desempate na fila de prioridade implementa implicitamente o custo lexicográfico, cujo resultado mais visível é a divisão de zonas planas de acordo com a distância da borda destas. Com estas propriedades, o algoritmo de Beucher e Meyer produz o mesmo conjunto de resultados que qualquer algoritmo de floresta de caminhos mínimos que utilize o custo do caminho com dois componentes, o máximo e o custo lexicográfico. Desta forma, o conjunto de resultados passíveis de obtenção através do algoritmo de fila de prioridade é dado pela definição IFT-WT.

O procedimento proposto funciona em dois passos, sendo um de inicialização da fila e outro de trabalho. Entretanto, a inicialização requer um conjunto de sementes - *pixels* selecionados e rotulados da imagem - que podem ser obtidos através de um algoritmo de detecção de mínimos regionais, ou através de marcadores pré-processados ou selecionados por usuários. A inicialização consiste então em enfileirar estas sementes com seu custo correspondendo ao seu nível de cinza e rotulá-las de acordo com uma função de rótulos ou pelos componentes conexos formados. O processamento trata de remover o *pixel* com maior prioridade da fila - aquele com o menor custo e inserido por primeiro entre aqueles com o mesmo custo - e então analisar cada um de seus vizinhos ainda não rotulados, inserindo-os na fila com custo correspondente ao seu nível de cinza, e propagar seu rótulo. Itera-se desta forma até que a fila esteja vazia, indicando que todos os *pixels* da imagem foram processados. O Alg. 3 apresenta a transformada *watershed* por fila de prioridade de Beucher e Meyer.

ALGORITMO 3: Fila de Prioridade

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$  e conjunto de mínimos  $M$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D, lab(p) \leftarrow MASK$ 
3: End
4: // Passo 1: Insere os mínimos regionais na fila
5: for all  $p \in m_i \in M$  do
6:   HEAPQUEUEPUSH( $p, im(p)$ )
7:    $lab(p) \leftarrow i$ 
8: end for
9: // Passo 2: Processa os pixels na ordem da fila
10: while HEAPQUEUEEMPTY() = false do
11:    $p \leftarrow HEAPQUEUEPOP()$ 
12:   for all  $q \in N(p) | lab(q) = MASK$  do
13:      $lab(q) \leftarrow lab(p)$ 
14:     HEAPQUEUEPUSH( $q, im(q)$ )
15:   end for
16: end while

```

Analizando-se o Algoritmo 3, nota-se que o controle da simulação de imersão é completamente realizado pela fila de prioridade, dado que nenhuma relação de ordem entre os níveis de cinza dos *pixels* é explícita no algoritmo. A caracterização deste procedimento como uma transformada *watershed* por busca em largura é dada pela política FIFO em cada nível de cinza, sendo que a visitação destes é feita por uma fronteira que se expande, visitando todos na fronteira atual antes de visitar a próxima. O uso dos mínimos regionais como sementes garante o comportamento de busca em largura de forma geral, no entanto o uso de marcadores arbitrários pode resultar na formação de caminhos diferentes, visto que a fronteira em expansão pode encontrar *pixels* com custo menor que o atual, fazendo com que estes tenham maior prioridade do que o atual e interrompendo a análise deste nível até que todos os custos inferiores sejam processados. Este comportamento de fato é incorreto, e a correção deste é feita armazenando-se o custo máximo do caminho e o utilizando na inserção na fila de prioridade.

Como dito anteriormente, este algoritmo adere à definição IFT-WT , e portanto não tem solução única, sendo dependente de ordem de varredura e visitação. A programação deste algoritmo tem seu desempenho extremamente ligado à eficiência da fila de prioridade. Um estudo mais aprofundado desta foge ao escopo deste trabalho, sendo alvo de diversos outros, todavia é importante ressaltar a necessidade da política FIFO como regra de desempate, não considerada em diversas implementações (e.g. *priority\_queue* da biblioteca STL para C++ e *heapq* para Python), sendo necessário utilizar funções de comparação personalizadas que levam em conta um valor referente a ordem de inserção, armazenado junto ao custo. Ainda em relação a seu desempenho, este algoritmo apresenta uma solução elegante, separando problemas como a detecção dos mínimos regionais e avaliação de custos em algoritmos distintos. A Fig. 3.5 apresenta um exemplo de resultado obtido utilizando o algoritmo de fila de prioridade.

<table border="1"> <tr><td>8</td><td>6</td><td>6</td><td>6</td></tr> <tr><td>0</td><td>3</td><td>7</td><td>5</td></tr> <tr><td>2</td><td>4</td><td>0</td><td>2</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>9</td></tr> </table>	8	6	6	6	0	3	7	5	2	4	0	2	2	4	6	9	<table border="1"> <tr><td>A</td><td>A</td><td>A</td><td>B</td></tr> <tr><td>A</td><td>A</td><td>B</td><td>B</td></tr> <tr><td>A</td><td>B</td><td>B</td><td>B</td></tr> <tr><td>A</td><td>A</td><td>B</td><td>B</td></tr> </table>	A	A	A	B	A	A	B	B	A	B	B	B	A	A	B	B
8	6	6	6																														
0	3	7	5																														
2	4	0	2																														
2	4	6	9																														
A	A	A	B																														
A	A	B	B																														
A	B	B	B																														
A	A	B	B																														
(a)	(b)																																

Fig. 3.5: Fila de Prioridade. (a) Imagem, (b) Resultado (N4)

Uma versão deste algoritmo, também mencionada por Beucher e Meyer, degenera a otimalidade da definição IFT-WT e produz *pixels* de linha entre as regiões da imagem. Este algoritmo foi descrito por Meyer, incluindo uma condição na extração dos elementos da fila, se dois ou mais vizinhos tiverem rótulos diferentes, o *pixel* será rotulado como *watershed* [35]. Esta implementação, apesar de não aderir a nenhuma definição estabelecida, é encontrada no pacote OpenCV [36] e na biblioteca Milena [37], como parte da ferramenta Olena [38].

### 3.2.1 Implementação Python

```

from common import *

# constants
MASK = -2

def hierarchicalQueue(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = MASK

    # find minima
    M = findMinima(im, N, D)

    queue = wsHeapQueue()

    for m in xrange(len(M)):
        for p in M[m]:
            queue.push(p, im[p])
            lab[p] = m+1

    while not queue.empty():
        p = queue.pop()

```

```

for q in N(p):
    if lab[q] != MASK:
        continue

    lab[q] = lab[p]
    queue.push(q, im[q])

return ws.end()

```

### 3.3 Algoritmo Dijkstra-Moore de Caminhos Mínimos de Meyer

No mesmo trabalho em que Meyer apresenta a definição TD-WT, são propostos três algoritmos para implementação desta [7]. O primeiro destes, tratado nesta seção, é baseado nos trabalhos de Dijkstra [39] e Moore [40] para construção de florestas de caminhos mínimos. Ao usar estes algoritmos, Meyer modifica sua função de custo, que deixa de ser apenas uma soma, para indicar a inclinação de um caminho, e assim construir os caminhos de máxima inclinação e menor custo a partir dos mínimos regionais de uma imagem. Sua definição produz resultados únicos, possibilitados pela marcação de *pixels* de divisão. Entretanto, estes não são previstos no algoritmo proposto, e desta forma, Roerdink e Meijster [12] propõe uma modificação, reproduzida no Alg. 4, de forma a implementar a definição TD-WT completamente.

---

#### ALGORITMO 4: Dijkstra-Moore

---

**Entrada:**  $im$ : Imagem de níveis de cinza sem zonas planas, com domínio  $D$  e mínimos regionais

$$m_i \in M$$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:   WSHEd  $\leftarrow 0$ 
3:    $\forall p \in D, lab(p) \leftarrow 0, dist(p) \leftarrow \infty$ 
4: End

5: for all  $p \in m_i \in M$  do
6:    $lab(p) \leftarrow i$ 
7:    $dist(p) \leftarrow im(p)$ 
8: end for

9: while  $D \neq \emptyset$  do
10:   $p \leftarrow u : u \in D$  and  $dist(u) = min_{v \in D} dist(v)$ 

11:   $D \leftarrow D \setminus \{p\}$ 
12:  for all  $q \in N(p)$  do
13:    if  $dist(p) + COST(p, q) < dist(q)$  then

```

```

14:     dist(q) ← dist(p) + COST(p,q)
15:     lab(q) ← lab(p)
16:     else if dist(p) + COST(p, q) = dist(q) and lab(q) ≠ lab(p) then
17:         lab(q) ← WSHED
18:     end if
19: end for
20: end while

21: Procedure COST( $p, q : q \in N(p)$ )
22:      $LS(p) \leftarrow \max_{\forall u \in N(p) | im(u) < im(p)} (im(p) - im(u))$ 
23:      $LS(q) \leftarrow \max_{\forall u \in N(q) | im(u) < im(q)} (im(q) - im(u))$ 

24:     if im(p) > im(q) then
25:         return  $LS(p)$ 
26:     else if im(q) > im(p) then
27:         return  $LS(q)$ 
28:     else if im(p) = im(q) then
29:         return  $\frac{LS(p)+LS(q)}{2}$ 
30:     end if
31: End

```

Os algoritmos de Dijkstra e Moore são clássicos na literatura de teoria de grafos na abordagem do problema de florestas de caminhos mínimos [41], portanto seu funcionamento geral é bastante conhecido, sendo aplicado sobre o grafo inerente à imagem. Todavia o cálculo da função de custo e a política de rotulação dos *pixels* necessitam de mais explicações. Assim, o algoritmo inicia rotulando os mínimos regionais, previamente detectados por outro procedimento e inicializando os valores da matriz **dist** com os níveis de cinza de cada *pixel* pertencente a um mínimo regional, de onde são selecionados os *pixels* e armazenadas as distâncias topográficas até os mínimos. A matriz **dist** é utilizada para realizar a função de uma fila hierárquica, onde primeiro são processados os elementos com menor custo. Desta forma, a cada iteração processa-se e retira-se do conjunto contendo todos os *pixels* da imagem aquele com o menor valor na matriz **dist** (linhas 10 e 11). Em seguida, avalia-se a vizinhança do *pixel* selecionado, verificando se o custo oferecido é menor que o custo atual destes, e então propagando o novo custo e rótulo. Nesta operação, o algoritmo tem funcionamento idêntico ao algoritmo de Dijkstra. No entanto, se o custo oferecido for igual ao custo atual do *pixel* vizinho e estes possuírem rótulos diferentes, caracteriza-se uma ambiguidade de caminhos de máxima inclinação, e assim este *pixel* é rotulado como *watershed* (linhas 16 e 17).

O algoritmo de Dijkstra reconhecidamente mantém similaridades com o algoritmo de busca em largura clássico [26]. Sua adaptação para a transformada *watershed*, utilizando uma função de custo crescente e positiva que garanta as mesmas propriedades do algoritmo original, será caracterizada como uma transformada *watershed* por busca em largura. Esta similaridade é dada na forma que a fronteira de expansão cresce de forma próxima à uniforme, dependendo dos custos dos vizinhos destas. Pode-se dizer que em comparação com a definição clássica - avaliação de todos vértices a

distância  $k$  antes dos vértices a distância  $k + 1$  - esta é mantida, no entanto  $k$  passa a ter o valor da função de custo.

A programação deste algoritmo depende de outros dois, para remoção de zonas planas, discutido na Sec. 2.2.3 e detecção de mínimos regionais. É necessário também decidir a forma de aplicação da operação de extração do menor elemento, feita avaliando-se a matriz **dist** a cada iteração ou utilizando-se uma fila de prioridade, sendo a segunda forma mais recomendada por reduzir o número de operações e mantendo em uma estrutura apenas os dados necessários, e proposta originalmente para este fim por Dial [42]. A Fig. 3.6 apresenta um exemplo de resultado obtido utilizando o algoritmo Dijkstra-Moore de transformada *watershed*.

8	6	6	6
0	3	7	5
2	4	0	2
2	4	6	9

(a)

A	A	W	B
A	A	B	B
A	B	B	B
A	A	B	B

(b)

Fig. 3.6: Dijkstra-Moore de Caminhos Mínimos. (a) Imagem, (b) Resultado (N4)

### 3.3.1 Implementação Python

```

from common import *

# constants
MASK = -2
WSHED = 0

def dijkstraMoore(im, offsets):

    # initialise variables
    lc = lowerComplete(im, offsets)
    ws = wsImage(lc)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    def cost(p, q):
        LSp = im[p] - min(im[N(p)])
        LSq = im[q] - min(im[N(q)])
        if im[p] > im[q]:

```

```

        return LSp
    elif im[q] > im[p]:
        return LSq
    else:
        return (LSp + LSq)/2.0

lab[:] = MASK
dist = ws.makeWorkCopy(inf)

for m in xrange(len(M)):
    for p in M[m]:
        lab[p] = m+1
        dist[p] = im[p]

# make the domain object a set
D = list(D)

while len(D) > 0:
    p = dist.argmin()

    D.remove(p)
    for q in N(p):

        c = cost(p,q)
        if dist[p] + c < dist[q] and lab[q] == MASK:
            dist[q] = dist[p] + c
            lab[q] = lab[p]
        elif dist[p] + c == dist[q] and lab[q] != lab[p]:
            lab[q] = WSHED

    dist[p] = inf

return ws.end()

```

### 3.4 Algoritmo Hill Climbing de Meyer

Assim como o algoritmo Dijkstra-Moore, Meyer propôs em conjunto à definição TD-WT o algoritmo *Hill Climbing*, utilizando os conceitos de *downstream* e *upstream* [7]. Estes conjuntos processam intrinsecamente a distância topográfica, e portanto neste algoritmo esta não é calculada explicitamente. Da mesma forma que o algoritmo Dijkstra-Moore, este não foi proposto prevendo a unicidade das soluções, e também foi revisto por Roerdink e Meijster [12], sendo reproduzido aqui detalhando a forma de construção do conjunto *upstream*, e implementando completamente a definição TD-WT. O Alg. 5 apresenta o pseudocódigo da transformada *watershed* por *Hill Climbing*.

ALGORITMO 5: Hill Climbing
----------------------------

**Entrada:**  $im$ : Imagem de níveis de cinza sem zonas planas, com domínio  $D$  e mínimos regionais  $m_i \in M$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:   WSHED  $\leftarrow 0$ 
3:   MASK  $\leftarrow -1$ 
4:    $\forall p \in m_i \in M, lab(p) \leftarrow i$ 
5:    $\forall p \in D \setminus M, lab(p) \leftarrow MASK$ 
6:    $S \leftarrow \{p \in D : \exists q \in N(p), im(p) \neq im(q)\}$ 
7: End

8: while  $S \neq \emptyset$  do
9:    $p \leftarrow u : u \in S$  and  $im(u) = \min_{v \in S} im(v)$ 
10:   $S \leftarrow S \setminus \{p\}$ 

11:  $\Gamma^{-1}(p) \leftarrow \{q \mid q \in N(p) \text{ and } im(p) = \min_{u \in N(q)} im(u)\}$ 

12: for all  $q \in \Gamma^{-1}(p) \cap S$  do
13:   if  $lab(q) = MASK$  then
14:      $lab(q) \leftarrow lab(p)$ 
15:   else if  $lab(q) \neq lab(p)$  then
16:      $lab(q) \leftarrow WSHED$ 
17:   end if
18: end for
19: end while

```

O algoritmo opera calculando para cada *pixel* o seu *upstream* e propagando seu rótulo para este. Quando dois rótulos diferentes se encontram, o *pixel* é marcado como *watershed*. Este processamento é iniciado a partir das bordas dos mínimos regionais e ordenado pelo nível de cinza dos *pixels*, processando todos a um nível antes de processar o próximo. Uma vez que o *pixel* foi processado, ele não é mais visitado. Há duas etapas importantes neste algoritmo, que são a seleção do pixel a ser processado e o cálculo do *upstream*.

A primeira etapa consiste na seleção entre todos os *pixels* que ainda não foram processados aquele com o menor nível de cinza. Esta operação é equivalente a inserir inicialmente todos os *pixels* em uma fila de prioridade e executar uma remoção da frente desta a cada iteração. No Alg. 5 a forma escolhida de representação desta operação foi através de um subconjunto da imagem original e buscando o elemento de menor nível de cinza, e então removendo-o do subconjunto.

A segunda etapa, de cálculo do *upstream*, envolve a análise dos vizinhos do *pixel* selecionado e dos vizinhos destes. Isto é necessário pois o *upstream* é o inverso do *downstream*. Desta forma, o cálculo do *upstream* só é possível determinando-se o *downstream* de cada vizinho, que, por consequência, necessita da análise dos vizinhos destes. A Fig. 3.7 apresenta um exemplo de cálculo do *upstream*

para o *pixel* de coordenadas (3,3). Em (a) é mostrada a imagem com o *pixel* em análise hachurado. Em (b) o *downstream* é apresentado com setas e os *pixels* de destino destas são hachurados. Em (c) são hachurados os *pixels* que pertencem ao *upstream* do *pixel* em análise, ou seja, aqueles cujo *downstream* contém o *pixel* (3,3), com as setas em destaque em (b).

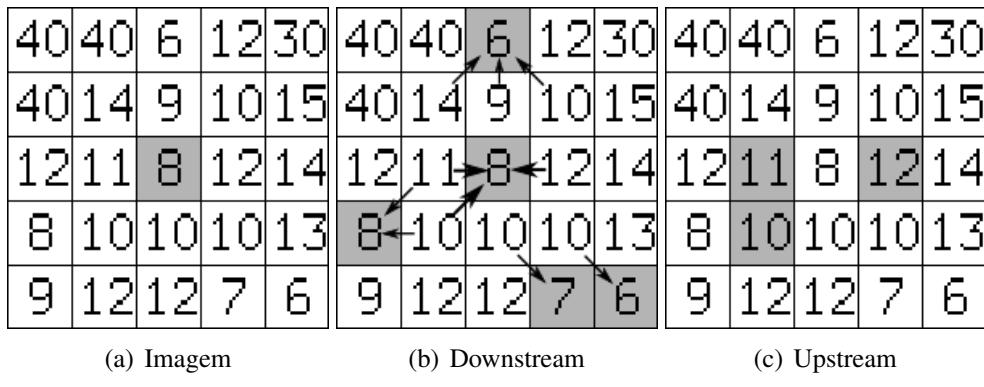


Fig. 3.7: Cálculo do upstream para uma imagem de exemplo, com o downstream indicado por setas a partir dos pixels de origem.

Para o algoritmo *Hill Climbing* também verifica-se a caracterização da busca em largura, com base no mesmo princípio para isto no algoritmo Dijkstra-Moore. Sendo todos os *pixels* a um determinado nível de cinza processados antes do próximo: o custo destes será definitivo e novamente a distância  $k$  pode ser vista como o custo mínimo a partir de um mínimo regional, e todos os *pixels* a um determinado custo serão processados antes dos *pixels* a custo  $k + 1$ . A programação deste algoritmo deve levar em consideração a necessidade de um algoritmo de remoção de zonas planas e de detecção de mínimos regionais, assim como uma técnica eficiente para armazenamento e identificação do *pixel* de menor valor dado um subconjunto da imagem. Assim como no caso do algoritmo Dijkstra-Moore, onde deseja-se o menor valor dentre um conjunto de distâncias topográficas, uma fila de prioridade é recomendada para implementação deste. A Fig. 3.8 apresenta um exemplo de resultado do algoritmo *Hill Climbing*.

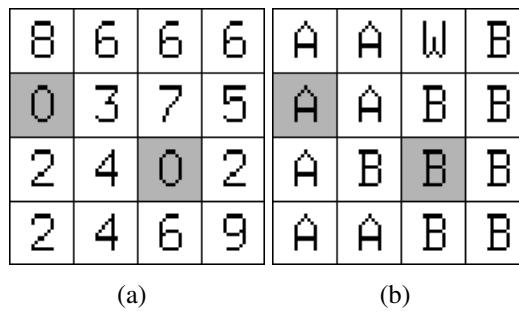


Fig. 3.8: Hill Climbing (a) Imagem, (b) Resultado (N4)

### 3.4.1 Implementação Python

```

from common import *

# constants
WSHED = 0
MASK = -2

def hillClimbing(im, offsets):

    # initialise variables
    lc = lowerComplete(im, offsets)
    ws = wsImage(lc)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = MASK

    # find minima
    M = findMinima(im, N, D)

    # expands xrange into a list
    S = list(D)

    # label minima
    for m in xrange(len(M)):
        for p in M[m]:
            lab[p] = m+1
            # removes inner pixels from S
            hmax = max(im[N(p)])
            if hmax == im[p]:
                S.remove(p)

    Sc = dict()
    while len(S) > 0:
        p = S[(im[S]).argmin()]
        S.remove(p)
        Sc[p] = True

        upstream = list()
        for q in N(p):
            if im[p] == min(im[N(q)]):
                upstream.append(q)
        for q in upstream:
            if Sc.has\_key(q):
                continue

            if lab[q] == MASK:
                lab[q] = lab[p]
            elif lab[q] != lab[p]:
                lab[q] = WSHED

```

```
return ws.end()
```

### 3.5 Algoritmo Berge de Caminhos Mínimos de Meyer

O terceiro algoritmo proposto por Meyer [7] é baseado no algoritmo de construção de florestas de caminhos mínimos de C. Berge [43]. Dado que o algoritmo de Dijkstra-Moore constrói uma SPF (*Shortest Path Forest*), qualquer algoritmo com esta capacidade, utilizando a função de custo de distância topográfica, também irá produzir como resultado uma transformada *watershed*. Todavia, assim como nos algoritmos Dijkstra-Moore e Hill-Climbing, o original de Meyer não prevê o uso de *pixels* de *watershed* para garantir a implementação correta da definição TD-WT e unicidade da solução. Desta forma, Roerdink e Meijster [12] revisam-no e este é reproduzido no Alg. 6.

#### ALGORITMO 6: Berge

**Entrada:**  $im$ : Imagem de níveis de cinza sem zonas planas, com domínio  $D$  e mínimos regionais  $m_i \in M$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D, lab(p) \leftarrow 0, dist(p) \leftarrow \infty$ 
3:    $\forall p \in m_i \in M, lab(p) \leftarrow i, dist(p) \leftarrow im(p)$ 
4:   stable  $\leftarrow$  true
5:   WSHEd  $\leftarrow 0$ 
6: End
7: repeat
8:   stable  $\leftarrow$  true
9:   for  $p \in D^+$  do
10:    PROPAGATE( $p, N^+(p)$ )
11:   end for
12:   for  $p \in D^-$  do
13:    PROPAGATE( $p, N^+(p)$ )
14:   end for
15: until stable = true

16: Procedure PROPAGATE( $p, Q$ )
17:   for all  $q \in Q$  do
18:     if  $dist(p) + COST(p, q) < dist(q)$  then
19:        $dist(q) \leftarrow dist(p) + COST(p, q)$ 
20:        $lab(q) \leftarrow lab(p)$ 
21:       stable  $\leftarrow$  false
```

```

22:      else if dist(p) + COST(p, q) = dist(q) and lab(q) ≠ lab(p) and lab(q) ≠ WSHED
23:          lab(q) ← WSHED
24:          stable ← false
25:      end if
26:  end for
27: End

28: Procedure COST( $p, q : q \in N(p)$ )
29:      $LS(p) \leftarrow \max_{\forall u \in N(p) \mid im(u) < im(p)} (im(p) - im(u))$ 
30:      $LS(q) \leftarrow \max_{\forall u \in N(q) \mid im(u) < im(q)} (im(q) - im(u))$ 

31:     if im(p) > im(q) then
32:         return LS(p)
33:     else if im(q) > im(p) then
34:         return LS(q)
35:     else if im(p) = im(q) then
36:         return  $\frac{LS(p) + LS(q)}{2}$ 
37:     end if
38: End

```

O algoritmo de Berge para transformada *watershed* é um caso em particular na classificação utilizada neste trabalho, pois não possui comportamento de calcular os resultados guiado por largura ou profundidade. Sua varredura pode ser feita de forma aleatória, até a estabilização dos resultados, determinado pela variável **stable** no Alg. 6. Entretanto, de modo a obter resultados estáveis mais rapidamente (menos varreduras), Roerdink e Meijster sugerem o uso de varreduras sequenciais em *raster* e *anti-raster* representadas no algoritmo por  $D^+$  e  $D^-$ . Para cada *pixel*, propaga-se seu resultado atual para seus vizinhos ainda não visitados -  $N^+$  - relativos à ordem de varredura atual. A estabilização do algoritmo se dá quando nenhum *pixel* tiver seu rótulo modificado, sendo que, a cada nova iteração necessária, todos os *pixels* devem ser reprocessados.

Em uma implementação em *software*, este comportamento pode ser indesejado, por repetir diversas vezes as operações, mas, em dispositivos com memória de leitura sequencial rápida, onde o acesso a esta se torna muito eficiente dado um sentido, o algoritmo pode apresentar resultados superiores a outros ou mesmo ser a única opção. Em particular, este algoritmo também é interessante justamente por sua varredura aleatória, que remove condições intrínsecas de outros, como o custo lexicográfico intrínseco às estruturas de filas, passando a depender única e exclusivamente da função de custo utilizada na propagação. A Fig. 3.9 apresenta um exemplo de resultado do algoritmo Berge.

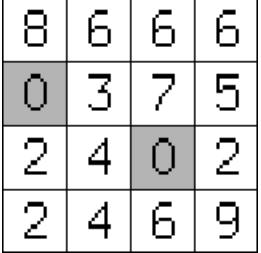
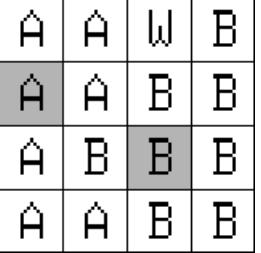
	
(a)	(b)

Fig. 3.9: Berge com custo TD-WT. (a) Imagem, (b) Resultado (N4)

### 3.5.1 Implementação Python

```

from common import *

# constants
MASK = -2
WSHED = 0

def berge(im, offsets):

    # initialise variables
    lc = lowerComplete(im, offsets)
    ws = wsImage(lc)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    def cost(p, q):
        LSp = im[p] - min(im[N(p)])
        LSq = im[q] - min(im[N(q)])
        if im[p] > im[q]:
            return LSp
        elif im[q] > im[p]:
            return LSq
        else:
            return (LSp + LSq)/2.0

    def propagate(p, Q, cmp):
        s = True
        for q in Q:
            if not cmp(q, p):
                continue

            c = cost(p,q)
            if dist[p] + c < dist[q]:

```

```

        dist[q] = dist[p] + c
        lab[q] = lab[p]
        s = False
    elif dist[p] + c == dist[q] and lab[q] != lab[p] and lab[q] != WSHED:
        lab[q] = WSHED
        s = False

    return s

lab[:] = MASK
dist = ws.makeWorkCopy(inf)
stable = False

D = list(D)

for m in xrange(len(M)):
    for p in M[m]:
        lab[p] = m+1
        dist[p] = im[p]

while not stable:
    stable = True
    for p in D:
        stable = stable and propagate(p, N(p), lambda u,v: u > v)

    for p in reversed(D):
        stable = stable and propagate(p, N(p), lambda u,v: u < v)

return ws.end()

```

## 3.6 Algoritmo Componentes Conexos de Bieniek e Moga

O algoritmo de componentes conexos de Bieniek e Moga se origina de estudos sobre a paralelização da transformada *watershed*, iniciados por Moga *et al.* [44] com a arquitetura para implementação distribuída e continuados por Bieniek *et al.* com a definição LC-WT e a proposta de um algoritmo paralelo [30]. No entanto, a formalização do algoritmo de componentes conexos viria somente com a união destes trabalhos [18, 29]. Desta forma, apresenta-se consistentemente a definição LC-WT, derivada da definição TD-WT, e um algoritmo capaz de implementá-la, com possibilidade de paralelização. O Alg. 7 apresenta a proposta de Bieniek e Moga.

### ALGORITMO 7: Componentes Conexos

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:   PLATEAU  $\leftarrow \infty$ 
3:    $\forall p \in D, \text{lab}(p) \leftarrow \text{MASK}$ 
4: End

5: // Passo 1
6: for all  $p \in D$  do
7:    $q \leftarrow p$ 
8:   for  $u \in N(p)$  and  $\text{im}(u) < \text{im}(p)$  do
9:     if  $\text{im}(u) < \text{im}(q)$  then
10:       $q \leftarrow u$ 
11:    end if
12:   end for
13:   if  $q \neq p$  then
14:      $\text{adr}(p) \leftarrow q$ 
15:   else
16:      $\text{adr}(p) \leftarrow \text{PLATEAU}$ 
17:   end if
18: end for

19: // Passo 2
20: for all  $p \in D \mid \text{adr}(p) = \text{PLATEAU}$  do
21:   if  $\exists q \in N(p) \mid \text{adr}(q) \neq \text{PLATEAU}$  and  $\text{im}(p) = \text{im}(q)$  then
22:     QUEUEPUSH(q)
23:   end if
24: end for
25: while QUEUEEMPTY() = false do
26:    $p \leftarrow \text{QUEUEPOP}()$ 
27:   for all  $q \in N(p) \mid \text{adr}(q) = \text{PLATEAU}$  and  $\text{im}(p) = \text{im}(q)$  do
28:      $\text{adr}(q) \leftarrow \text{adr}(p)$ 
29:     QUEUEPUSH(q)
30:   end for
31: end while

32: // Passo 3
33: for  $p \in D \mid \text{adr}(p) = \text{PLATEAU}$  do
34:    $\text{adr}(p) \leftarrow p$ 
35:   for  $q \in N^-(p) \mid \text{im}(p) = \text{im}(q)$  do
36:      $u \leftarrow \text{FIND}(p)$ 
37:      $v \leftarrow \text{FIND}(q)$ 
38:      $\text{adr}(u) \leftarrow \text{adr}(v) \leftarrow \min(u,v)$ 
39:   end for

```

```

40: end for

41: // Passo 4
42: basins ← 1
43: for  $p \in D$  do
44:   r ← FIND(p)
45:   adr(p) ← r
46:   if lab(r) = MASK then
47:     lab(r) ← basins
48:     basins ← basins + 1
49:   end if
50:   lab(p) ← lab(r)
51: end for

52: Procedure FIND(p)
53:   q ← p
54:   while adr(q) ≠ q do
55:     q ← adr(q)
56:   end while
57:   u ← p
58:   while adr(u) ≠ q do
59:     v ← adr(u)
60:     adr(u) ← q
61:     u ← v
62:   end while
63:   return q
64: End

```

A definição LC-WT tem, por sua natureza, múltiplas soluções, e, desta forma, a resposta do algoritmo - que não tem componentes estocásticos - depende da forma de varredura da imagem e da vizinhança. Todavia, todas as respostas são válidas de acordo com a LC-WT. O Alg. 7 opera em quatro passos básicos, utilizando **adr** como matriz auxiliar, armazenando endereços de *pixels*. No primeiro passo, cada *pixel* armazena em **adr** o endereço de seu vizinho mínimo, ou uma constante em caso deste não existir, indicando uma zona plana. O segundo passo consiste em resolver o endereço dos *pixels* de zona plana que não são mínimos regionais. Isto é feito propagando o endereço a partir da borda destas zonas, uniformemente, através de uma fila. O terceiro passo trata da resolução dos mínimos regionais, que também são zonas planas, passam a ser representados por um *pixel*, aquele com o menor endereço entre todos do componente conexo, sendo armazenado em **adr** para os outros. No último passo, os caminhos gerados na matriz **adr** são percorridos, rotulando-se o *pixel* avaliado de acordo com o final do caminho, correspondente a um mínimo regional. Na visitação é utilizada a técnica de compressão de caminhos, de modo que cada *pixel* neste passa a apontar diretamente para o final do caminho, otimizando a busca por seu rótulo no momento da sua avaliação.

Analizando-se o comportamento deste algoritmo, os três passos iniciais apenas constróem os caminhos entre qualquer *pixel* e seu mínimo regional correspondente. O último passo, responsável pela rotulação, atravessa esses caminhos até encontrar o fim deles, caracterizando uma busca em profundidade. A Fig. 3.10 apresenta um exemplo de resultados do algoritmo componentes conexos, com as duas possibilidades de soluções, geradas por varredura em *raster* e *anti-raster*. A diferença entre as duas soluções corresponde ao *pixel* que na definição TD-WT é rotulado como *watershed*, como pode ser visto entre outras na Fig. 3.6.

8	6	6	6	A	A	B	B	B	B	B	A
0	3	7	5	A	A	B	B	B	B	A	A
2	4	0	2	A	B	B	B	B	A	A	A
2	4	6	9	A	A	B	B	B	B	A	A
(a)				(b)				(c)			

Fig. 3.10: Componentes Conexos. (a) Imagem, (b) Resultado em raster, (c) Resultado em anti-raster (N4)

### 3.6.1 Implementação Python

```

from common import *

# constants
MASK = -2
PLATEAU = -1

def connectedComponents(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = MASK
    adr = ws.makeWorkCopy(0)

    queue = wsQueue()

    def find(p):
        q = p
        while adr[q] != q:
            q = adr[q]
        u = p
        for i in range(N):
            if adr[i] == q:
                adr[i] = u
        return u
    return lab

```

```

while adr[u] != q:
    v = adr[u]
    adr[u] = q
    u = v
return q

# step 1
for p in D:
    q = p
    for u in N(p):
        if im[u] < im[q]:
            q = u

        if q != p:
            adr[p] = q
        else:
            adr[p] = PLATEAU

# step 2
for p in D:
    if adr[p] != PLATEAU:
        continue

    for q in N(p):
        if adr[q] == PLATEAU or im[q] != im[p]:
            continue

        queue.push(q)

while not queue.empty():
    p = queue.pop()
    for q in N(p):
        if adr[q] != PLATEAU or im[q] != im[p]:
            continue

        adr[q] = p
        queue.push(q)

# step 3
for p in D:
    if adr[p] != PLATEAU:
        continue

    adr[p] = p

    for q in N(p):
        if q > p or im[q] != im[p]:
            continue

```

```

    u = find(p)
    v = find(q)
    adr[u] = adr[v] = min(u, v)

    # step 4
    basins = 1
    for p in D:
        r = find(p)
        adr[p] = r
        if lab[r] == MASK:
            lab[r] = basins
            basins += 1
        lab[p] = lab[r]

    return ws.end()

```

### 3.7 Algoritmo Union-Find de Meijster e Roerdink

Após a proposta da definição TD-WT por Meyer [7], com algoritmos fracamente definidos e inadequados para obter os resultados de acordo com a definição [12], Meijster e Roerdink propõem um algoritmo baseado na definição de *downstream* e que implementa corretamente tal definição [22]. O algoritmo Union-Find utiliza operações de compressão de caminho e verificação de unicidade de rótulos sobre um dígrafo acíclico (DAG - *directed acyclic graph*), construído sobre uma imagem sem zonas planas. Desta forma, são utilizados de fato três algoritmos em separado, com suas saídas concatenadas: remoção de zonas planas, construção do DAG e resolução de rótulos. Assim, o Alg. 8 constrói o dígrafo acíclico da imagem sem zonas planas e o Alg. 9 efetua a resolução de rótulos por *union-find*.

#### ALGORITMO 8: DAG

**Entrada:**  $im$ : Imagem de níveis de cinza

**Saida:**  $sln$ : Dígrafo representado sob forma de matriz indexado pelo pixel

```

1: for  $p \in D$  do
2:   if  $\Gamma(p) = \emptyset$  then
3:      $\Gamma(p) \leftarrow \{q \in N(p) \mid im(q) = \min_{u \in N(p)} im(u), im(q) < im(p)\}$ 
4:     if  $\Gamma(p) = \emptyset$  then
5:        $\Gamma(p) \leftarrow p$ 
6:       if  $\exists q \in N(p) \mid im(q) = im(p)$  then
7:         QUEUEPUSH(p)
8:         while QUEUEEMPTY() = false do
9:           q = QUEUEPOP()
10:          for  $u \in N(q) \mid im(u) = im(p)$  do

```

```

11:           if  $\Gamma(u) = \emptyset$  then
12:                $\Gamma(u) = p$ 
13:               QUEUEPUSH(u)
14:           end if
15:       end for
16:   end while
17:   end if
18: end if
19: end if
20:  $i \leftarrow 0$ 
21: for all  $q \in \Gamma(p)$  do
22:      $sln[p,i] \leftarrow q$ 
23:      $i \leftarrow i + 1$ 
24: end for
25: end for

```

## ALGORITMO 9: Union-Find

**Entrada:**  $sln$ : Dígrafo representado sob forma de matriz indexado pelo pixel**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $W \leftarrow -1$ 
3:    $N \leftarrow 4$ 
4:    $\forall p \in D, lab(p) \leftarrow MASK$ 
5:   basins  $\leftarrow 1$ 
6: End

7: for  $p \in D$  do
8:    $r \leftarrow FIND(p)$ 
9:   if  $r \neq W$  then
10:    if  $lab(r) = MASK$  then
11:       $lab(r) \leftarrow basins$ 
12:       $basins \leftarrow basins + 1$ 
13:    end if
14:     $lab(p) \leftarrow lab(r)$ 
15:  else
16:     $lab(p) \leftarrow WSSED$ 
17:  end if
18: end for

```

```

1: Procedure FIND( $p$ )
2:    $rep \leftarrow 0$ 
3:    $i \leftarrow 1$ 
4:   while  $i \leq N$  and  $rep \neq W$  do
5:     if  $sln[p,i] \neq p$  and  $sln[p,i] \neq W$  then
6:        $sln[p,i] \leftarrow FIND(sln[p,i])$ 
7:     end if
8:     if  $i = 1$  then
9:        $rep \leftarrow sln[p,1]$ 
10:    else if  $sln[p,i] \neq rep$  then
11:       $rep \leftarrow W$ 
12:      for all  $j \in [1, N]$  do
13:         $sln[p,j] \leftarrow W$ 
14:      end for
15:    end if
16:     $i \leftarrow i + 1$ 
17:   end while
18:   return  $rep$ 
19: End

```

A construção do DAG é feita definindo um vértice para cada *pixel* e conectando estes de acordo com seu *downstream*. Esta regra, calculada na linha 3, é válida para todos os *pixels* da imagem, pois esta não contém zonas planas, exceto aqueles de mínimo regional, onde  $\Gamma(p)$  será vazio para todo o componente conexo, que recebe então tratamento especial. Nestas regiões, o primeiro *pixel* descoberto como pertencente a ela ( $\Gamma(p) = \emptyset$ ) passa a ser o seu representante, e propaga-se seu endereço para seus vizinhos de componente, modificando seus valores de  $\Gamma$  antes de serem visitados, o que exige a condição de teste na linha 2. As linhas 20 a 24 tratam de transformar o *downstream* na forma de armazenamento de grafo na matriz *sln* adotada. A Fig. 3.11 apresenta em (a) a imagem de exemplo sem zonas planas utilizada ao longo deste capítulo. Sobre esta é calculado o DAG mostrado em (b) de acordo com o Alg. 8.

A etapa de resolução dos rótulos a partir do DAG, apresentada no Alg. 9, busca, a partir de cada vértice, seguir os caminhos formados por suas arestas direcionadas até um vértice sem arestas de saída, indicando ser o representante de um mínimo regional. Esta operação, chamada de *Find*, advém do algoritmo de gerenciamento de conjuntos desconexos de Tarjan [27], chamado de *Union-Find*, cujo nome foi trazido para o contexto da transformada *watershed*, dada sua importância. Ao encontrar dois rótulos diferentes atingíveis a partir do mesmo vértice, este passa a ser identificado como um *pixel* de *watershed* (linhas 10 e 11 na operação **Find**). É importante também notar o caráter recursivo desta operação, com a compressão dos caminhos, evitando a repetição de percursos.

Da mesma forma que o Alg. Componentes Conexos (Alg. 7), este também pode ser caracterizado como uma busca em profundidade, justamente pelo comportamento da operação **Find** na resolução dos rótulos. Em particular, pode-se dizer que o Alg. 7 é um caso particular deste, onde cada vértice pode possuir apenas uma aresta de saída [12]. O resultado final da resolução de rótulos sobre o DAG

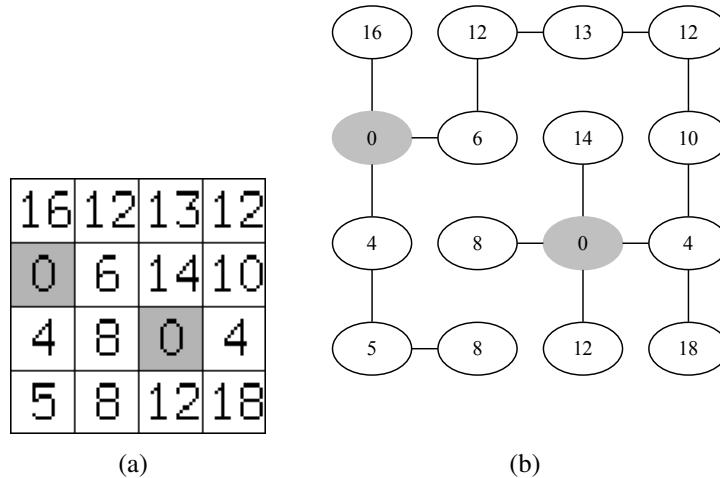


Fig. 3.11: Construção do DAG. (a) Imagem sem zonas planas, (b) DAG

da Fig. 3.11 (b) é apresentado na Fig. 3.12, igualando-se aos outros algoritmos que implementam a definição TD-WT. É importante ressaltar que o resultado do DAG deve ser idêntico ao LCG, exceto nos mínimos regionais, onde os vértices no LCG não têm arestas, e no DAG as arestas levam a um representante do componente.

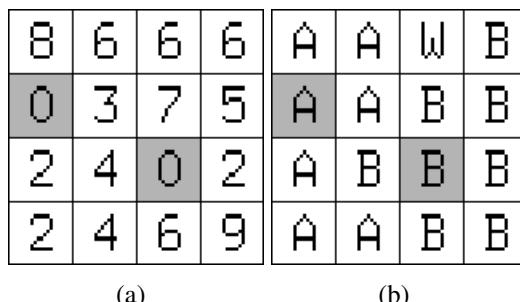


Fig. 3.12: Union-Find. (a) Imagem, (b) Resultado (N4)

### 3.7.1 Implementação Python

```
from common import *
# constants
WSHED = 0
W = -1
```

```

MASK = -2

def unionFind(im, offsets):
    lc = lowerComplete(im, offsets)
    sln = digraph(lc, offsets)

    # initialise variables
    ws = wsImage(lc)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = MASK
    basins = 1

    def find(p):
        rep = 0
        i = 0
        gamma = sln[p]
        while i < len(gamma) and rep != W:
            if gamma[i] != p and gamma[i] != W:
                gamma[i] = find(gamma[i])

            if i == 0:
                rep = gamma[0]
            elif gamma[i] != rep:
                rep = W
                for j in range(len(gamma)):
                    gamma[j] = W
            i += 1

        return rep

    for p in D:
        r = find(p)
        if r != W:
            if lab[r] == MASK:
                lab[r] = basins
                basins += 1

            lab[p] = lab[r]
        else:
            lab[p] = WSHED

    return ws.end()

def digraph(im, offsets):
    # initialise variables

```

```

ws = wsImage(im)
N, im, lc, D = ws.begin(offsets)

sln = dict()

# scan the image
for p in D:

    if sln.has\key(p):
        continue # is part of the minimal plateau

    gamma = list()
    minvalue = min(im[N(p)])
    plateau = False
    for q in N(p):

        if im[q] == minvalue and im[q] < im[p]:
            gamma.append(q)

        if im[q] == im[p]:
            plateau = True

    if len(gamma) == 0:
        if plateau:
            # minimal plateau
            queue = wsQueue()
            queue.push(p)
            gamma.append(p) # make the first one the representative
            while not queue.empty():
                q = queue.pop()
                for u in N(q):

                    if im[u] == im[p]:
                        if not sln.has\key(u):
                            sln[u] = [p]
                            queue.push(u)

                else:
                    gamma.append(p) # one-pixel minimum

    sln[p] = gamma

return sln

```

## 3.8 Algoritmo IFT de Lotufo e Falcão

A transformada imagem-floresta (IFT - *Image Foresting Transform*) é em sua essência uma generalização dos algoritmos de Dijkstra [39] e Moore [40] com a implementação baseada em filas de prioridade proposta por Dial [42] para fins de cálculo de uma floresta de caminhos mínimos [19]. Diversas aplicações para este algoritmo são possíveis variando-se seus parâmetros [8], mas, no con-

texto da transformada *watershed*, utiliza-se a função de custo de máximo [19]. No entanto, o uso da fila hierárquica com política FIFO implica no uso indireto do custo lexicográfico, tornando assim o custo a ser minimizado para cada *pixel* baseado em duas componentes. Assim, de acordo com o formalizado na seção 2.2.5, o Alg. 10 implementa a definição IFT-WT.

---

**ALGORITMO 10: IFT**


---

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$ ,  $S$ : marcadores

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D$ ,  $done(p) \leftarrow \text{false}$ 
3:    $\forall p \notin S$ ,  $C_1(p) \leftarrow \infty$ ,  $lab(p) \leftarrow \text{MASK}$ ,  $parent(p) \leftarrow \text{MASK}$ 
4:    $\forall p \in S$ ,  $C_1(p) \leftarrow im(p)$ ,  $lab(p) \leftarrow \lambda(p)$ ,  $parent(p) \leftarrow p$ , HEAPQUEUEPUSH( $p$ ,  $im(p)$ )
5: End

6: while HEAPQUEUEEMPTY() = false do
7:    $p \leftarrow$  HEAPQUEUEPOP()
8:    $done(p) \leftarrow \text{true}$ 
9:   for all  $q \in N(p) \mid done(q) = \text{false}$  do
10:     $c \leftarrow max(C_1(p), im(q))$ 
11:    if  $c < C_1(q)$  then
12:      if HEAPQUEUECONTAINS( $q$ ) then
13:        HEAPQUEUEREMOVE( $q$ )
14:      end if
15:       $C_1(q) \leftarrow c$ 
16:       $lab(q) \leftarrow lab(p)$ 
17:       $parent(q) \leftarrow p$ 
18:      HEAPQUEUEPUSH( $q$ ,  $C_1(q)$ )
19:    end if
20:   end for
21: end while
```

O Alg. 10, como mencionado anteriormente, é uma generalização, e, portanto, assemelha-se muito à abordagem de Dijkstra e aos Algs. Dijkstra-Moore de Caminhos Mínimos (Alg. 4) e Fila de Prioridade (Alg. 3). Sua principal diferença está na utilização explícita de uma função de custo, apresentada como o máximo do caminho para obter os resultados da definição IFT-WT, mas que poderia ser substituída pela distância topográfica para produzir soluções de acordo com a definição TD-WT. Por ser um algoritmo de propósito mais geral, este também necessita gerenciar a remoção e reinserção na fila de prioridade, dependendo do custo encontrado (linhas 12 e 13). No entanto, como visto na Fig. 2 da ref. [19], para a função de custo máximo, esta remoção não é necessária, e, assim, o algoritmo pode ser degenerado no Alg. Fila de Prioridade (Alg. 3, Seção 3.2).

De fato, estes passam a apresentar o mesmo comportamento, efetuando uma busca em largura com critério dependente do custo do caminho. Ainda em relação ao Alg. 10, este faz uso da função  $\lambda$ , que representa alguma rotulação utilizada para os *pixels*, gerada pelos componentes conexos ou arbitrariamente por um usuário, por exemplo. Como resultado, além da segmentação da imagem, é produzida a floresta, onde cada *pixel* tem apenas um antecessor, denotado por  $parent(p)$ , e cada região é uma árvore.

A implementação do algoritmo IFT depende de um algoritmo de detecção de mínimos regionais e de uma fila de prioridade com política FIFO. No entanto, seu uso mais comum é com marcadores escolhidos e rotulados pelo usuário: sendo assim, seu desempenho grandemente dependente da fila. O algoritmo IFT implementa a correção comentada no Alg. 3 para tratar o uso de marcadores arbitrários na transformada *watershed*. A Fig. 3.13 apresenta um dos possíveis resultados e a floresta gerada para este.

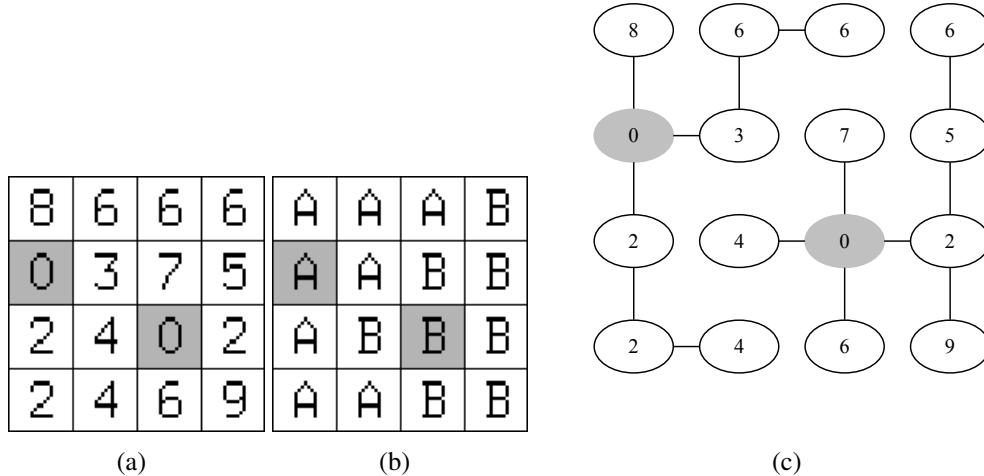


Fig. 3.13: IFT. (a) Imagem, (b) Resultado, (c) Floresta resultante com raízes em cinza. (N4)

### 3.8.1 Implementação Python

```

from common import *

# constants
MASK = -2

def ift(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)
  
```

```

# find minima
M = findMinima(im, N, D)

# create the working images
done = ws.makeWorkCopy(False)
c1 = ws.makeWorkCopy(inf)
par = ws.makeWorkCopy(MASK)

lab[:] = MASK

queue = wsHeapQueue()

for m in xrange(len(M)):
    for p in M[m]:
        c1[p] = im[p]
        lab[p] = m+1
        par[p] = p
        queue.push(p, im[p])

while not queue.empty():
    p = queue.pop()
    done[p] = True
    for q in N(p):
        if done[q]:
            continue

        c = max(c1[p], im[q])
        if c < c1[q]:
            if c1[q] < inf:
                if queue.contains(q, c1[q]):
                    queue.remove(q, c1[q])
            c1[q] = c
            lab[q] = lab[p]
            par[q] = p
            queue.push(q, c1[q])

return ws.end()

```

### 3.9 Algoritmo Código de Corrente de Sun, Yang e Ren

O algoritmo de código de corrente para transformada *watershed*, proposto por Sun, Yang e Ren [23], estende a noção comum desta técnica de detecção de contornos para uso na indicação dos caminhos de máxima inclinação. De fato, o algoritmo é baseado na proposta de Bieniek e Moga [29], executando passos muito semelhantes em objetivo, e, como este, produz resultados condizentes com a definição LC-WT. Todavia, apesar da semelhança, o Alg. 11, apresentado abaixo, pode ser modificado para produzir outras informações úteis mais facilmente, como por exemplo a área [23].

**ALGORITMO 11:** Código de Corrente**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$ **Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D : out(p) \leftarrow MASK, in(p) \leftarrow 0$ 
3:   basins  $\leftarrow 1$ 
4: End

5: // Passo 1
6: for  $p \in D$  do
7:    $q \leftarrow u \in N(p) : im(u) = \min_{v \in N(p)} im(v)$  and  $im(u) < im(p)$ 
8:   if  $q \neq \emptyset$  then
9:      $out(p) \leftarrow ARROW(p,q)$ 
10:     $in(q) \leftarrow in(q) \cup POINTIN(p,q)$ 
11:   end if
12: end for

13: // Passo 2
14: for all  $p \in D | out(p) \neq MASK$  do
15:   if  $\exists q | q \in N(p), out(q) = MASK$  and  $im(p) = im(q)$  then
16:     QUEUEPUSH(p)
17:   end if
18: end for
19: while QUEUEEMPTY() = false do
20:    $p \leftarrow QUEUEPOP()$ 
21:   for  $q \in N(p) | im(p) = im(q)$  and  $out(q) = MASK$  do
22:      $out(q) \leftarrow ARROW(q,p)$ 
23:      $in(p) \leftarrow in(p) \cup POINTIN(q,p)$ 
24:     QUEUEPUSH(q)
25:   end for
26: end while

27: // Passo 3
28: for  $p \in D | out(p) = MASK$  do
29:   for  $q \in N(p) | im(p) = im(q)$  do
30:      $out(q) \leftarrow ARROW(q,p)$ 
31:      $in(p) \leftarrow in(p) \cup POINTIN(q,p)$ 
32:     STACKPUSH(q)
33:   end for
34:   while STACKEMPTY() = false do
35:      $u \leftarrow STACKPOP()$ 

```

```

36:   for  $v \in N(u) \mid im(u) = im(v)$  do
37:     if  $out(v) = MASK$  and  $v \neq p$  then
38:        $out(v) \leftarrow ARROW(v,u)$ 
39:        $in(u) \leftarrow in(u) \cup POINTIN(v,u)$ 
40:       STACKPUSH(v)
41:     end if
42:   end for
43: end while
44: end for

45: // Passo 4
46: for  $p \in D \mid out(p) = MASK$  do
47:   STACKPUSH(p)
48:   while STACKEMPTY() = false do
49:      $u \leftarrow STACKPOP()$ 
50:     lab( $u$ )  $\leftarrow basins$ 
51:     for all  $v \in N(u) \mid POINTIN(v, u) \in in(u)$  do
52:       lab( $v$ )  $\leftarrow basins$ 
53:       STACKPUSH(v)
54:     end for
55:   end while
56:   basins  $\leftarrow basins + 1$ 
57: end for

58: Procedure POINTIN( $p,q$ )
59:   return  $ARROW(q,p)$ 
60: End

```

O principal ponto na compreensão do Alg. 11 é o uso das matrizes **out** e **in**, que armazenam a direção do vizinho mínimo e dos vizinhos para os quais o *pixel* em questão é mínimo, respectivamente. Estas matrizes correspondem aos conceitos de *downstream* e *upstream*. Assim, os três primeiros passos efetuam o cálculo destes valores para todos os *pixels* da imagem, escolhendo um representante para cada componente conexo de mínimo regional e direcionando os outros para este. Desta forma, constrói-se implicitamente um grafo direcionado e formam-se caminhos de todos os *pixels* para um mínimo regional. No último passo são gerados os rótulos para os representantes e propaga-se este rótulo seguindo o caminho a partir do *upstream*. Esta última etapa, onde o algoritmo torna-se similar ao Alg. *Hill Climbing* (Alg. 5), caracteriza-o como uma busca em profundidade, pois utiliza uma estrutura de pilha para visitação dos *pixels* de *upstream*, assim percorrendo o caminho o máximo possível antes de iniciar outro.

É importante também ressaltar as funções **Arrow** e **PointIn** utilizadas no Alg. 11. A primeira, descrita na seção 2.1.4 é convencionada com valores de retorno em  $[0, N]$ . A segunda faz uso da primeira para indicar a direção contrária do caminho, invertendo-se os parâmetros de entrada. Desta

forma o endereço da direção também é invertido, gerando o número corretamente. O Alg. 11 utiliza estes valores em **in** indicando-os como um conjunto, porém a programação desta forma seria muito dispendiosa em termos de uso de memória. Assim, Sun, Yang e Ren [23] utilizam os endereços dos vizinhos como *flags* binárias na memória, e então torna-se possível, por exemplo, com um byte, indicar que um *pixel* pertence ao *downstream* de até oito vizinhos seus e utilizando operações rápidas para isto. A Fig. 3.14 apresenta um exemplo do uso do algoritmo código de corrente.

8	6	6	6
0	3	7	5
2	4	0	2
2	4	6	9

(a)

A	A	A	B
A	A	B	B
A	B	B	B
A	A	B	B

(b)

Fig. 3.14: Código de Corrente. (a) Imagem, (b) Resultado (N4)

### 3.9.1 Implementação Python

```
from common import *

# constants
MASK = -2

def chainCode(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    wout = ws.makeWorkCopy(MASK)
    win = ws.makeWorkCopy(0)
    basins = 1
    queue = wsQueue()
    stack = wsStack()

    def pointOut(p, q):
        # finds the offset
        c = q - p
        # query the neighbourhood for the offset
        idx = ws.neighbour.query(c)
        if idx == -1:
            raise Exception("Neighbour not found")
```

```

return idx

def pointIn(p,q):
    idx = pointOut(q,p)
    return 1 << idx

# step 1
for p in D:

    minValue = min(im[N(p)])
    minpx = None

    for q in N(p):

        if im[q] < im[p] and im[q] == minValue:
            minpx = q

        if not minpx is None:
            wout[p] = pointOut(p, minpx)
            win[minpx] |= pointIn(p, minpx)

# step 2
for p in D:
    if wout[p] == MASK:
        continue

    for q in N(p):

        if wout[q] == MASK and im[q] == im[p]:
            queue.push(p)
            break

    while not queue.empty():
        p = queue.pop()
        for q in N(p):

            if im[p] == im[q] and wout[q] == MASK:
                wout[q] = pointOut(q, p)
                win[p] |= pointIn(q, p)
                queue.push(q)

# step 3
for p in D:
    if wout[p] != MASK:
        continue

    for q in N(p):
        if im[q] != im[p]:
            continue

        wout[q] = pointOut(q, p)

```

```

win[p] |= pointIn(q, p)
stack.push(q)

while not stack.empty():
    u = stack.pop()
    for v in N(u):

        if im[u] == im[v] and wout[v] == MASK and v != p:
            wout[v] = pointOut(v, u)
            win[u] |= pointIn(v, u)
            stack.push(v)

# step 4
for p in D:
    if wout[p] != MASK:
        continue

    stack.push(p)
    while not stack.empty():
        u = stack.pop()
        lab[u] = basins
        for v in N(u):

            # checks if v is in in-list of u
            if pointIn(v, u) & win[u] > 0:
                lab[v] = basins
                stack.push(v)

    basins += 1

return ws.end()

```

## 3.10 Algoritmo Zona de Empate de Audigier, Lotufo e Couprie

O algoritmo de zona de empate, proposto por Audigier, Lotufo e Couprie [20], baseia-se na transformada *watershed* por IFT, vista na seção 3.8, de forma a unificar suas soluções, provendo um algoritmo de transformada *watershed* com função de custo de caminho máximo e solução única, coerente com a definição TZ-IFT-WT, apresentada na Sec. 2.2.5. Para realizar corretamente o cálculo dos *pixels* de zona de empate, o Alg. IFT (Alg. 10) é modificado, de modo a explicitamente armazenar os custos lexicográficos, necessários para determinar as situações em que o *pixel* receberá o rótulo **TZ**. O Alg. 12 apresenta as modificações comentadas.

ALGORITMO 12: Zona de Empate

**Entrada:** *im*: Imagem de níveis de cinza com domínio *D*, *S*: marcadores

**Saida:** *lab*: Imagem rotulada

```

1: Initialise
2:    $\forall p \in D, C_2(p) \leftarrow 0$ ,  $\text{done}(p) \leftarrow \text{false}$ 
3:    $\forall p \notin S, C_1(p) \leftarrow \infty$ ,  $\text{lab}(p) \leftarrow \text{MASK}$ ,  $\text{parent}(p) \leftarrow \text{MASK}$ 
4:    $\forall p \in S, C_1(p) \leftarrow \text{im}(p)$ ,  $\text{lab}(p) \leftarrow \lambda(p)$ ,  $\text{parent}(p) \leftarrow p$ ,  $\text{HEAPQUEUEPUSH}(p, \text{im}(p))$ 
5: End

6: while  $\text{HEAPQUEUEEMPTY}() = \text{false}$  do
7:    $p \leftarrow \text{HEAPQUEUEPOP}()$ 
8:    $\text{done}(p) \leftarrow \text{true}$ 
9:   for all  $q \in N(p) \mid \text{done}(q) = \text{false}$  do
10:     $c \leftarrow \max(C_1(p), \text{im}(q))$ 
11:    if  $c < C_1(q)$  then
12:      if  $\text{HEAPQUEUECONTAINS}(q)$  then
13:         $\text{HEAPQUEUEREMOVE}(q)$ 
14:      end if
15:       $C_1(q) \leftarrow c$ 
16:       $\text{lab}(q) \leftarrow \text{lab}(p)$ 
17:       $\text{parent}(q) \leftarrow p$ 
18:       $\text{HEAPQUEUEPUSH}(q, C_1(q))$ 
19:      if  $c = C_1(p)$  then
20:         $C_2(q) \leftarrow C_2(p) + 1$ 
21:      end if
22:      else if  $c = C_1(q)$  and  $\text{lab}(q) \neq \text{lab}(p)$  then
23:        if  $c = C_1(p)$  then
24:          if  $C_2(q) = C_2(p) + 1$  then
25:             $\text{lab}(q) \leftarrow \text{TIE-ZONE}$ 
26:          end if
27:        else
28:           $\text{lab}(q) \leftarrow \text{TIE-ZONE}$ 
29:        end if
30:      end if
31:    end for
32: end while

```

As modificações em relação ao Alg. 10 são efetuadas nas linhas 19 a 29, onde a imagem  $C_2$  armazena o custo lexicográfico e este é avaliado quando o custo máximo não satisfizer a condição de unicidade. A condição testada na linha 22 verifica se dois caminhos atingiram um *pixel* com mesmo custo máximo e que estes têm rótulos diferentes, indicando possíveis soluções diferentes. Em seguida, verifica-se na linha 23 se esta condição foi atingida dentro de uma zona plana, que caso contrário, indica automaticamente um empate. Se, dentro desta zona plana, os custos lexicográficos forem iguais (linha 24) então constitui-se uma outra forma de empate, baseada na segunda componente do

custo do caminho. Resumidamente, a zona de empate ocorre quando custo máximo e lexicográfico são iguais para um *pixel*, estando este em zona plana ( $C_2 > 0$ ) ou não ( $C_2 = 0$ ).

Apesar das modificações efetuadas, o comportamento do algoritmo não é modificado, dado que este é regido pela fila de prioridade, mantendo assim sua característica de busca em largura, herdada do Alg. 10. O rótulo **TZ**, correspondente às zonas de empate e responsável pela unicidade da solução do algoritmo, não deve ser interpretado como uma linha de *watershed*, mas como um identificador de ambiguidade, onde mais de uma solução é possível mas todas são válidas [21]. A Fig. 3.15 apresenta um exemplo de resultado produzido pelo algoritmo de zona de empate. Deve-se ressaltar que a zona de empate não garante a separação entre regiões. Este caso pode ser visto em uma zona plana onde no encontro de regiões não há *pixel* onde o custo lexicográfico entre em empate.

The figure consists of two 4x4 grids labeled (a) and (b). Grid (a) contains numerical values: Row 1: 8, 6, 6, 6; Row 2: 0, 3, 7, 5; Row 3: 2, 4, 0, 2; Row 4: 2, 4, 6, 9. Grid (b) contains labels: Row 1: A, A, TZ, B; Row 2: A, A, TZ, B; Row 3: A, TZ, B, B; Row 4: A, A, TZ, TZ. The labels A, B, and TZ represent different regions or states assigned by the algorithm.

Fig. 3.15: Zona de Empate da IFT. (a) Imagem, (b) Resultado (N4)

### 3.10.1 Implementação Python

```
from common import *

# constants
MASK = -2
TIE_ZONE = 0

def tieZone(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    # create the working images
    done = ws.makeWorkCopy(False)
    c1 = ws.makeWorkCopy(inf)
    c2 = ws.makeWorkCopy(0)
```

```

par = ws.makeWorkCopy(MASK)
lab[:] = MASK

queue = wsHeapQueue()

for m in xrange(len(M)):
    for p in M[m]:
        c1[p] = im[p]
        lab[p] = m+1
        par[p] = p
        queue.push(p, im[p])

while not queue.empty():
    p = queue.pop()
    done[p] = True
    for q in N(p):
        if done[q]:
            continue

        c = max(c1[p], im[q])
        if c < c1[q]:
            if c1[q] < inf:
                if queue.contains(q, c1[q]):
                    queue.remove(q, c1[q])
            c1[q] = c
            lab[q] = lab[p]
            par[q] = p
            queue.push(q, c1[q])
            if c == c1[p]:
                c2[q] = c2[p] + 1
            elif c == c1[q] and lab[q] != lab[p]:
                if c == c1[p]:
                    if c2[q] == c2[p] + 1:
                        lab[q] = TIE\_ZONE
                else:
                    lab[q] = TIE\_ZONE

return ws.end()

```

### 3.11 Algoritmo Tobogã Invariante a Ordem de Lin *et al.*

O algoritmo de tobogã invariante a ordem, de Lin *et al.*, propõe um método de resolução de ambiguidades referentes a ordem de processamento dos *pixels* de uma imagem [24]. Para isso, o rótulo **RIDGE** é utilizado quando dois caminhos de inclinação máxima a partir de um mesmo *pixel* levam a dois rótulos diferentes. Esta proposta remete à definição TD-WT [7] e ao algoritmo Union-Find para *watershed* de Meijster e Roerdink [22]. No entanto, a equivalência direta entre estas propostas só ocorre quando a imagem tem suas zonas planas removidas. Isto se deve à natureza da definição TD-WT, sabidamente problemática nestas zonas, por ter custo zero no caminho [12]. Todavia, o al-

goritmo de Lin *et al.* não tem como requisito o processo de remoção de zonas planas, tratando este problema implicitamente através do cálculo das distâncias geodésicas até as bordas. Entretanto, o processo de rotulação não utiliza esta informação na avaliação dos rótulos, causando portanto a diferença. A correção deste problema é simples, bastando restringir o conjunto de *pixels* em que o rótulo será avaliado, aderindo então à definição TD-WT. O Alg. 13 apresenta o pseudocódigo corrigido restringindo a avaliação de rótulos no conjunto  $S_{min}$ , contendo os elementos de  $S$  onde a distância é mínima dentro deste.

---

**ALGORITMO 13:** Tobogã Invariante a Ordem

---

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$

**Saida:**  $lab$ : Imagem rotulada

1: **Initialise**

2:      $\forall p \in D : lab(p) \leftarrow \text{MASK}$

3: **End**

    // Passo 1: Para cada pixel com vizinho menor, cria a lista de rampas máximas

4: **for all**  $p \in D$  **do**

5:      $h \leftarrow im(p)$

6:      $min \leftarrow min_{q \in N(p)} im(q)$

7:     **if**  $h > min$  **then**

8:          $sliding(p) \leftarrow \{q \in N(p) \mid im(q) = min\}$

9:         QUEUEPUSH( $p$ )

10:         $dist(p) \leftarrow 0$

11:     **end if**

12: **end for**

    // Passo 2: Propagação das bordas para dentro de plateaus

13: **while** QUEUEEMPTY() = **false** **do**

14:      $p \leftarrow \text{QUEUEPOP}()$

15:      $d \leftarrow dist(p) + 1$

16:      $h \leftarrow im(p)$

17:     **for all**  $q \in N(p) \mid im(q) = h$  **do**

18:         **if**  $sliding(q) = \emptyset$  **then**

19:              $sliding(q) \leftarrow \{p\}$

20:              $dist(q) \leftarrow d$

21:             QUEUEPUSH( $q$ )

22:         **else if**  $dist(q) = d$  **then**

23:              $sliding(q) \leftarrow sliding(q) \cup \{p\}$

24:         **end if**

25:     **end for**

26: **end while**

    // Passo 3: Rotulação dos mínimos

27:  $basins \leftarrow 1$

```

28: for  $p \in D$  |  $\text{sliding}(p) = \emptyset$  and  $\text{lab}(p) = \text{MASK}$  do
29:    $\text{lab}(p) \leftarrow \text{basins}$ 
30:    $\text{basins} \leftarrow \text{basins} + 1$ 
31:    $\text{QUEUEPUSH}(p)$ 
32:    $h = \text{im}(p)$ 
33:   while  $\text{QUEUEEMPTY}() = \text{false}$  do
34:      $q \leftarrow \text{QUEUEPOP}()$ 
35:     for all  $u \in N(q)$  |  $\text{im}(u) = h$  do
36:       if  $\text{lab}(u) = \text{MASK}$  then
37:          $\text{lab}(u) \leftarrow \text{lab}(p)$ 
38:          $\text{QUEUEPUSH}(u)$ 
39:       end if
40:     end for
41:   end while
42: end for
43:   // Passo 4: Rotulação dos pixels por busca em profundidade
44:   for  $p \in D$  do
45:      $\text{RESOLVE}(p)$ 


---


        // Desce o caminho da gota recursivamente, até resolver todos os vizinhos
1: Procedure  $\text{RESOLVE}(p)$ 
2:   if  $\text{lab}(p) = \text{MASK}$  then
3:      $S \leftarrow \text{sliding}(p)$ 
4:     for  $q \in S$  do
5:        $\text{RESOLVE}(q)$ 
6:     end for
7:      $d_{min} \leftarrow \min_{q \in S} \text{dist}(q)$ 
8:      $S_{min} \leftarrow \{q \in S : \text{dist}(q) = d_{min}\}$ 
9:     if  $\forall q \in S_{min}, \text{lab}(q) = \alpha$  then
10:       $\text{lab}(p) \leftarrow \alpha$ 
11:    else
12:       $\text{lab}(p) \leftarrow \text{WSHED}$ 
13:    end if
14:  end if
15: End

```

O Alg. 13 consiste de quatro passos básicos: criação da lista de rampas máximas para cada *pixel* com ao menos um vizinho em nível de cinza menor, propagação uniforme das rampas nas zonas planas, rotulação dos mínimos regionais, resolução dos rótulos dos *pixels* com base nas rampas máximas. A caracterização deste algoritmo como busca em profundidade se dá na última etapa, de resolução dos rótulos, onde os caminhos das rampas máximas são percorridos até a localização

de um *pixel* já rotulado. Um dos complicadores para programação deste algoritmo é justamente a lista de rampas máximas, necessária para cada *pixel*. Entretanto, algumas estratégias podem ser adotadas, como o uso de bits como sinalizadores de quais vizinhos do *pixel* pertencem à lista, como proposto por Sun, Yang e Ren [23] para o armazenamento do *upstream*. De forma a processar zonas planas uniformemente, uma fila é utilizada para ordenamento dos próximos *pixels* que devem ser processados.

De forma geral, o funcionamento do Alg. 13 corresponde à construção de um grafo direcionado acíclico, formando caminhos de máxima inclinação até mínimos regionais, e então determinando se todos os caminhos partindo de um vértice terminam em um mesmo mínimo regional, ou seja, se o rótulo de seus caminhos é único. A partir desta avaliação é definido se o *pixel* será rotulado conforme um mínimo ou como linha de *watershed*. Este processo une em apenas um algoritmo o método Union-Find [22, 12] que depende de três algoritmos, otimizando o acesso aos *pixels*. Aplicando o Algoritmo 13, modificado da versão original, obtém-se resultados únicos por imagem, independentes de ordem de varredura, aderentes à definição TD-WT e respeitando suas propriedades. A Fig. 3.16 apresenta um exemplo de resultado obtido usando o algoritmo de tobogã invariante a ordem.

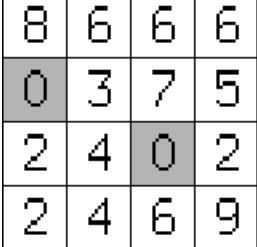
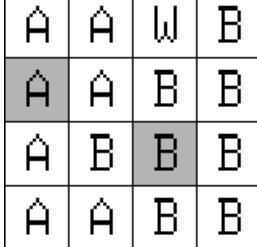
	
(a)	(b)

Fig. 3.16: Tobogã Invariante a Ordem. (a) Imagem, (b) Resultado (N4)

### 3.11.1 Implementação Python

```
from common import *

#constants
MASK = -2
RIDGE = 0

def orderInvariantToboggan(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    lab[:] = MASK
```

```

queue = wsQueue()
dist = ws.makeWorkCopy(0)
sliding = dict()

def resolve(p):
    if lab[p] == MASK:

        S = sliding[p]

        for q in S:
            resolve(q)

        single\_label = True
        alfa = None

        dmin = min(dist[S])

        # identifies unique label on S
        for u in S:
            if dist[u] != dmin:
                continue

            if alfa is None:
                alfa = lab[u]
            else:
                if alfa != lab[u]:
                    single\_label = False

        if single\_label:
            lab[p] = alfa
        else:
            lab[p] = RIDGE

# step 1
for p in D:

    h = im[p]
    hmin = min(im[N(p)])

    if h > hmin:
        S = list()
        for q in N(p):
            if im[q] != hmin:
                continue
            S.append(q)
        sliding[p] = S
        queue.push(p)
        dist[p] = 0

```

```

# step 2
while not queue.empty():
    p = queue.pop()
    d = dist[p] + 1
    h = im[p]

    for q in N(p):
        if im[q] != h:
            continue

        if not sliding.has\_key(q):
            sliding[q] = [p]
            dist[q] = d
            queue.push(q)
        elif dist[q] == d:
            sliding[q].append(p)

# step 3
basins = 1
for p in D:
    if sliding.has\_key(p) or lab[p] != MASK:
        continue

    lab[p] = basins
    basins += 1

    queue.push(p)
    h = im[p]

    while not queue.empty():
        q = queue.pop()
        for u in N(q):
            if im[u] != h:
                continue

            if lab[u] == MASK:
                lab[u] = lab[p]
                queue.push(u)

# step 4
for p in D:
    resolve(p)

return ws.end()

```

### 3.12 Algoritmo Imersão Invariante a Ordem de Lin *et al.*

Em paralelo ao algoritmo de tobogã, Lin *et al.* propõe um algoritmo de imersão invariante a ordem utilizando os mesmos conceitos de resolução de ambiguidades com o rótulo **RIDGE** [24]. A

diferença entre as propostas se dá na ordem de processamento, onde, neste algoritmo, os *pixels* são visitados de acordo com seu nível em ordem ascendente, levando à noção de imersão. De modo a uniformizar a divisão das zonas planas, estas são visitadas a partir das bordas calculando sua distância geodésica utilizando uma fila, caracterizando um algoritmo de busca em largura a partir dos mínimos regionais. O algoritmo de imersão foi projetado para obter os mesmo resultados do tobogã [24], e, assim, apesar de remeter à definição TD-WT, não a implementa completamente, ignorando a distância geodésica na resolução dos rótulos. Similarmente ao algoritmo tobogã, a correção deste problema é feita restringindo o conjunto  $S$  no conjunto  $S_{min}$ , onde os elementos de  $S_{min}$  são os elementos  $S$  com valor de distância mínimo no conjunto, aderindo então à definição TD-WT. O Algoritmo 14 apresenta a implementação do procedimento de imersão invariante a ordem em pseudocódigo.

---

**ALGORITMO 14:** Imersão Invariante a Ordem

---

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D : lab(p) \leftarrow MASK$ 
3: End

4: Ordene os pixels por seu valor de nível de cinza, com mínimo  $h_{min}$  e máximo  $h_{max}$ 

5: basins  $\leftarrow 1$ 
6: for  $h \in [h_{min}, h_{max}]$  do
7:   // Passo 1: Simulação de inundação
8:   for all  $p \in D \mid im(p) = h$  do
9:      $min \leftarrow min_{\forall q \in N(p)} im(q)$ 
10:    if  $h > min$  then
11:      // Verifica todos os vizinhos que estão no nível mínimo
        encontrado
12:       $S \leftarrow \{q \mid q \in N(p) \text{ and } im(q) = min\}$ 
13:       $d_{min} \leftarrow min_{\forall q \in S} dist(q)$ 
14:       $S_{min} \leftarrow \{q \in S : dist(q) = d_{min}\}$ 
15:      if  $\forall q \in S_{min}, lab(q) = \alpha$  then
16:         $lab(p) \leftarrow \alpha$  // Se o rótulo for único na menor distância,
          atribui para o pixel
17:      else
18:         $lab(p) \leftarrow WSHED$  // Se o rótulo não for único, cria uma
          barreira
19:      end if
20:      QUEUEPUSH(p)
21:       $dist(p) \leftarrow 0$ 
22:    end if
23:  end for

```

```

24:           // Passo 2: Crescimento de regiões com base na distância
   geodésica dentro de zona plana
25:   while QUEUEEMPTY() = false do
26:       p ← QUEUEPOP()
27:       d ← dist(p) + 1
28:       for all q ∈ N(p) | im(q) = h do
29:           if lab(q) = MASK then
30:               lab(q) ← lab(p)
31:               dist(q) ← d
32:               QUEUEPUSH(q)
33:           else if lab(p) ≠ lab(q) and dist(q) = d then
34:               lab(q) ← WSHED
35:           end if
36:       end for
37:   end while

38:   // Passo 3: Rotulação dos mínimos
39:   for p ∈ D | im(p) = h and lab(p) = MASK do
40:       lab(p) ← basins
41:       basins ← basins+ 1
42:       QUEUEPUSH(p)
43:       while QUEUEEMPTY() = false do
44:           q ← QUEUEPOP()
45:           for all u ∈ N(q) | im(u) = h and lab(u) = MASK do
46:               lab(u) ← lab(p)
47:               QUEUEPUSH(u)
48:           end for
49:       end while
50:   end for
51: end for

```

O Alg. 14 consiste de três passos básicos dependentes da ordenação dos *pixels*: simulação da inundação, verificando os *pixels* no nível de cinza em processamento, e rotulando-os de acordo com sua vizinhança; propagação das regiões nas zonas planas com base na distância geodésica até a borda; rotulação dos novos mínimos regionais. O primeiro passo é tratado como uma simulação de inundação, pois é nesta etapa em que os rótulos dos *pixels* de níveis de cinza menores passam para o nível sendo processado no momento, dando a noção do nível de água sendo elevado. O segundo passo trata da noção da imersão em zonas planas, onde a água é propagada uniformemente das bordas para o interior destas regiões, em um mesmo nível. Por último, os rótulos são gerados avaliando os níveis de cinza iterativamente, e descobrindo aquelas regiões em que nenhum *pixel* tem nível de cinza superior a um de seus vizinhos, ou seja, são novos mínimos regionais.

Assim como o algoritmo de tobogã ordenado, a imersão ordenada constrói implicitamente um

grafo direcionado acíclico, porém partindo dos mínimos regionais, ou seja, do fim dos caminhos de máxima inclinação. A avaliação dos rótulos é feita da mesma forma, analisando os possíveis caminhos a partir de um *pixel*, porém esta pode ser feita no momento da inundação, pois estes já são conhecidos, não sendo necessário manter listas de vizinhos. Por, de fato, construirão representações equivalentes para os caminhos, os Algs. 14 e 13 obtém os mesmos resultados, e, por sua vez, coerentes com a definição TD-WT. Assim, a Fig. 3.17 apresenta o mesmo resultado da Fig. 3.16, porém utilizando o algoritmo de imersão invariante a ordem e também coerente com os outros algoritmos de TD-WT, dada a unicidade de solução da definição.

8	6	6	6
0	3	7	5
2	4	0	2
2	4	6	9

(a)

A	A	W	B
A	A	B	B
A	B	B	B
A	A	B	B

(b)

Fig. 3.17: Imersão Invariante a Ordem. (a) Imagem, (b) Resultado (N4)

Em relação ao desempenho comparativo entre os algoritmos de imersão e tobogã ordenados, a imersão depende da ordenação dos *pixels* por seu nível de cinza, processo que também necessita de uma estrutura de dados especial para armazenamento dos *pixels* em cada nível. A ordenação, assim como no Alg. Imersão (Alg. 2), causa o maior problema de desempenho, sendo que o processamento restante é dependente desta, e por sua vez, de sua eficiência.

### 3.12.1 Implementação Python

```

from common import *

#constants
MASK = -2
RIDGE = 0

def orderInvariantImmersion(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    dist = ws.makeWorkCopy(0)

    lab[:] = MASK
    basins = 1

```

```

queue = wsQueue()

# "sorting"
levels = dict()
for p in D:

    if levels.has\_key(im[p]):
        levels[im[p]].append(p)
    else:
        levels[im[p]] = [p]

# watershed
H = sorted(levels.keys())
for h in H:
    # step 1
    for p in levels[h]:

        hmin = min(im[N(p)])
        if h > hmin:
            S = list()
            single\_label = True
            alfa = None
            for q in N(p):
                if im[q] != hmin:
                    continue

            S.append(q)

            dmin = min(dist[S])
            for q in S:
                if dist[q] != dmin:
                    continue

                if alfa is None:
                    alfa = lab[q]
                else:
                    if alfa != lab[q]:
                        single\_label = False

                if single\_label:
                    lab[p] = alfa
                else:
                    lab[p] = RIDGE
                    queue.push(p)
                    dist[p] = 0

    # step 2
    while not queue.empty():
        p = queue.pop()

```

```

d = dist[p] + 1
for q in N(p):
    if im[q] != h:
        continue

    if lab[q] == MASK:
        lab[q] = lab[p]
        dist[q] = d
        queue.push(q)
    elif lab[p] != lab[q] and dist[q] == d:
        lab[q] = RIDGE

# step 3
for p in levels[h]:
    if lab[p] != MASK:
        continue

    lab[p] = basins
    basins += 1
    queue.push(p)
    while not queue.empty():
        q = queue.pop()
        for u in N(q):
            if im[u] != h or lab[u] != MASK:
                continue

            lab[u] = lab[p]
            queue.push(u)

return ws.end()

```

### 3.13 Algoritmo Caminhos Mínimos de Osma-Ruiz *et al.*

Osma-Ruiz *et al.* propõe um algoritmo para computação mais eficiente de caminhos mínimos, buscando encontrar os caminhos de inclinação máxima em cada *pixel*, implementando implicitamente a definição LC-WT [25]. Seu incremento em eficiência é analisado em relação ao algoritmo de código de corrente de Sun, Yang e Ren [23], pois são propostas similares em relação ao funcionamento e produzem os mesmo resultados em relação à definição. A implementação de Osma-Ruiz *et al.* procura ser o mais eficiente possível em relação à visitação de *pixels*, formando caminhos até os mínimos regionais na primeira vez que o *pixel* é visitado. Para isso são utilizadas filas, que armazem os *pixels* que devem ser visitados em cada momento, e o que deve ser feitos com estes. O Alg. 15 apresenta o pseudocódigo para a proposta de Osma-Ruiz *et al.*

#### ALGORITMO 15: Caminhos Mínimos

**Entrada:** *im*: Imagem de níveis de cinza com domínio *D*

**Saida:** *lab*: Imagem rotulada

```

1: Initialise
2:   UNVISITED  $\leftarrow -8$ 
3:   PENDING  $\leftarrow -9$ 
4:    $\forall p \in D, \text{lab}(p) \leftarrow \text{UNVISITED}$ 
5:   basins  $\leftarrow 1$ 
6:   qPending, qEdge, qInner, qDescending: filas
7: End

8: for  $p \in D \mid \text{lab}(p) = \text{UNVISITED}$  do
9:   // Passo 1: Verificação de mínimos na vizinhança
10:  for  $q \in N(p)$  do
11:    if  $\text{im}(q) = \text{im}(p)$  then
12:      if QUEUEEMPTY(qPending) = true then
13:        lab(p)  $\leftarrow \text{PENDING}$ 
14:        QUEUERUSH(p, qPending)
15:      end if
16:      lab(q)  $\leftarrow \text{PENDING}$ 
17:      QUEUERUSH(q, qPending)
18:    else if  $\text{im}(q) = \min_{u \in N(p)} \text{im}(u)$  and  $\text{im}(q) < \text{im}(p)$  then
19:      min  $\leftarrow q$ 
20:    end if
21:  end for

22:  if QUEUEEMPTY(qPending) = false then
23:    // Passo 2.1: Detecção de zona plana
24:    while QUEUEEMPTY(qPending) = false do
25:      q  $\leftarrow \text{QUEUEPOP}(qPending)$ 
26:      if  $q \neq p$  then
27:        min  $\leftarrow$ 
28:        for all  $u \in N(q)$  do
29:          if  $\text{im}(u) = \text{im}(p)$  then
30:            if lab(u) = UNVISITED then
31:              lab(u)  $\leftarrow \text{PENDING}$ 
32:              QUEUERUSH(u, qPending)
33:            end if
34:            else if  $\text{im}(u) = \min_{v \in N(q)} \text{im}(v)$  and  $\text{im}(u) < \text{im}(q)$  then
35:              min  $\leftarrow u$ 
36:            end if
37:          end for
38:        end if

```

```

39:      if  $\exists min$  then
40:          lab(q)  $\leftarrow$  ARROW(q, min)
41:          QUEUEPUSH(q, qEdge)
42:      else
43:          QUEUEPUSH(q, qInner)
44:      end if
45:  end while

46:  if QUEUEEMPTY(qEdge) = false then
47:      // Passo 3.1: Propagação das bordas para dentro da zona
48:      // plana
49:      if QUEUEEMPTY(qInner) = false then
50:          // Propaga o caminho descendente da borda para dentro da
51:          // zona plana
52:          while QUEUEEMPTY(qEdge) = false do
53:              q  $\leftarrow$  QUEUEPOP(qEdge)
54:              for all  $u \in N(q)$  do
55:                  if im(u) = im(q) and lab(u) = PENDING then
56:                      lab(u)  $\leftarrow$  ARROW(u, q)
57:                      QUEUEPUSH(u, qEdge)
58:                  end if
59:              end for
60:          end while
61:      else
62:          QUEUECLEAR(qEdge)
63:      end if
64:      QUEUECLEAR(qInner)
65:  else
66:      // Passo 3.2: Rotulação de zona plana mínima
67:      while QUEUEEMPTY(qInner) = false do
68:          q  $\leftarrow$  QUEUEPOP(qInner)
69:          lab(q)  $\leftarrow$  basins
70:      end while
71:      basins  $\leftarrow$  basins + 1
72:  end if
73: else
74:     // Passo 2.1: Rotulação de pixel que não forma zona plana
75:     if  $\nexists min$  then
76:         lab(p)  $\leftarrow$  basins
77:         basins  $\leftarrow$  basins + 1
78:     else
79:         lab(p)  $\leftarrow$  ARROW(p, min)

```

```

78:      end if
79:  end if
80: end for

81: // Passo 4: Rotulação dos caminhos em profundidade
82: for all  $p \in D$  do
83:    $q \leftarrow p$ 
84:   while  $\text{lab}(q) \leq 0$  do
85:     QUEUEPUSH( $q$ , qDescending)
86:      $q \leftarrow \text{POINTED}(q, \text{lab}(q))$ 
87:   end while
88:   while QUEUEEMPTY(qDescending) = false do
89:      $u \leftarrow \text{QUEUEPOP}(qDescending)$ 
90:      $\text{lab}(u) \leftarrow \text{lab}(q)$ 
91:   end while
92: end for

```

Analizando-se o Alg. 15, este está dividido em 4 passos principais, interdependentes nos dados gerados, estando os 3 primeiros no contexto de uma varredura da imagem. Nesta varredura são tratados apenas os *pixels* não visitados anteriormente. O primeiro passo trata da verificação da vizinhança do *pixel* em análise, onde detecta-se se este faz parte de uma zona plana e se algum de seus vizinhos forma um caminho de máxima inclinação. A partir desta verificação, o passo 2 é dividido dependendo se o *pixel* forma ou não uma zona plana. No caso mais simples, onde o *pixel* não forma uma zona plana - não tem nenhum vizinho com mesmo nível de cinza - determina-se se este é um mínimo regional e rotula-se este ou então utiliza-se a operação **Arrow** para determinar qual o caminho que este deve seguir. Neste algoritmo a operação **Arrow** é convencionada no intervalo  $(-N, 0]$ . Caso o *pixel* pertença a uma zona plana - verificado através da quantidade de elementos em uma fila - processa-se a zona plana, identificando *pixels* internos e de borda, armazenando-os em filas distintas e usando a operação **Arrow** nas bordas. A partir desta distinção, o terceiro passo depende da existência ou não de *pixels* de borda na zona plana, que determina se a zona plana é um mínimo regional ou não. Se *pixels* de borda tiverem sido detectados, o algoritmo propaga esta borda para os *pixels* internos até a exaustão destes. O quarto passo, que caracteriza o algoritmo como uma busca em profundidade, trata da resolução dos rótulos dos *pixels* que não pertencem a um mínimo regional. Isto é feito percorrendo o caminho gerado através da operação **Pointed**, e atualizando os rótulos do caminho conforme o mínimo regional onde este termina.

A proposta deste algoritmo mostra uma forma de integrar procedimentos realizados separadamente em outros algoritmos, como o código de corrente de Sun, Yang e Ren [23], onde os passos de descoberta de vizinhos mínimos, resolução de zonas planas, rotulação de mínimos regionais e rotulação dos outros *pixels* são feitos em varreduras independentes. No entanto, a proposta de Osma-Ruiz *et al.* [25] necessita de filas otimizadas, pois são um ponto chave que afeta gravemente seu desempenho. Ainda em relação a estas, o algoritmo necessitou de duas correções, para incluir operações de limpeza das filas **qEdge** e **qInner** nas linhas 59 e 61. A Fig. 3.18 apresenta um exemplo do resul-

tado do algoritmo de Osma-Ruiz *et al.* de caminhos mínimos, de acordo com as possíveis soluções apresentadas pela definição LC-WT, sendo que outras soluções podem ser encontradas alterando-se a ordem de visitação de vizinhos e de varredura da imagem.

8	6	6	6
0	3	7	5
2	4	0	2
2	4	6	9

(a)

A	A	A	B
A	A	B	B
A	B	B	B
A	A	B	B

(b)

Fig. 3.18: Caminhos Mínimos. (a) Imagem, (b) Resultado (N4)

### 3.13.1 Implementação Python

```
from common import *

#constants
UNVISITED = -32767
PENDING = -32766

def shortestPaths(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)
    lab[:] = UNVISITED
    basins = 1

    qPending = wsQueue()
    qEdge = wsQueue()
    qInner = wsQueue()
    qDescending = wsQueue()

    def arrow(p, q):
        # finds the offset
        c = q - p
        # query the neighbourhood for the offset
        idx = ws.neighbour.query(c)
        if idx == -1:
            raise Exception("Neighbour not found")
        return -idx # inverted signal
```

```

def pointed(p, direction):
    # add the offset of the previously detected direction
    direction = int(direction)
    q = ws.neighbour.addOffset(p, -direction) # inverted signal
    if not q:
        raise Exception("Invalid direction")
    return q

# start looping
for p in D:

    if lab[p] != UNVISITED:
        continue

    minValue = min(im[N(p)])
    minpx = None

    for q in N(p):

        if im[q] == im[p]:
            if qPending.empty():
                lab[p] = PENDING
                qPending.push(p)
                lab[q] = PENDING
                qPending.push(q)
            elif im[q] < im[p] and im[q] == minValue:
                minpx = q

        if not qPending.empty():
            while not qPending.empty():
                q = qPending.pop()
                if q != p:

                    minValue = min(im[N(q)])
                    minpx = None

                    for u in N(q):

                        if im[u] == im[p]:
                            if lab[u] == UNVISITED:
                                lab[u] = PENDING
                                qPending.push(u)
                            elif im[u] < im[q] and im[u] == minValue:
                                minpx = u

                        if not minpx is None:
                            lab[q] = arrow(q, minpx)
                            qEdge.push(q)
                    else:
                        qInner.push(q)

```

```

if not qEdge.empty():
    if not qInner.empty():
        while not qEdge.empty():
            q = qEdge.pop()
            for u in N(q):

                if im[u] == im[q] and lab[u] == PENDING:
                    lab[u] = arrow(u, q)
                    qEdge.push(u)

            else:
                qEdge.clear()
                qInner.clear()
        else:
            while not qInner.empty():
                q = qInner.pop()
                lab[q] = basins
                basins += 1
            else:
                if minpx is None:
                    lab[p] = basins
                    basins += 1
                else:
                    lab[p] = arrow(p, minpx)

for p in D:

    q = p
    while lab[q] <= 0:
        qDescending.push(q)
        q = pointed(q, lab[q])

    while not qDescending.empty():
        u = qDescending.pop()
        lab[u] = lab[q]

# crop and return
return ws.end()

```

### 3.14 Algoritmo Watershed Cut de Cousty *et al.*

O algoritmo *watershed cut* foi introduzido por Cousty *et al.* [9], sendo definido como uma operação de corte de grafo sobre um grafo valorado nas arestas, obtido da imagem. De fato, o algoritmo produz um corte que gera uma MSF relativa aos mínimos regionais, isto implica que cada MST contém um e apenas um mínimo regional, também definidos sobre o grafo, e que cada mínimo também pertence a apenas uma MST. A interpretação de uma MSF relativa aos mínimos regionais pode ser vista como se cada conjunto de vértices que compõe um mínimo fosse visto como apenas um vértice,

e assim retornando à definição tradicional da MSF.

O algoritmo criado por Cousty et. al. calcula uma MSF relativa, conforme explicado anteriormente, e esta restrição permite-lhe aperfeiçoar outros algoritmos de MSF e atingir complexidade linear [9]. Iterativamente, são calculados caminhos de máxima inclinação - chamados de *stream* - onde busca-se, a partir do mínimo (*bottom*) do caminho, a aresta mínima deste, intuitivamente, para onde uma gota d'água escorreria. No entanto, permite-se que mais de um *pixel* no caminho tenha altitude mínima, caso que ocorre quando uma zona plana é explorada, ou seja, mais de um *pixel* pode ter o mesmo valor mínimo explorado até o momento, até que se encontre um valor de altitude menor que o atual. Desta forma, a exploração do *stream* é alternada entre busca em profundidade e busca em largura. Todavia, esta abordagem não trata o problema de divisão de zonas planas de modo a separar as regiões de forma mais igualitária possível. Este é um dos pontos, porém não o único, onde pode-se obter múltiplas soluções do *watershed cut*. Assim sendo, para se obter equivalência com outros métodos, é necessário realizar a remoção de zonas planas da imagem antes da construção do grafo. O Alg. 16 apresenta o *watershed cut* proposto por Cousty *et al.* [9].

---

**ALGORITMO 16:** Watershed Cut

---

**Entrada:**  $(D, E, F)$ : Grafo valorado

**Saida:** *lab*: Imagem rotulada

1: **Initialise**

2:      $\forall p \in D : \text{lab}(p) \leftarrow \text{MASK}$

3:      $\text{basins} \leftarrow 0$

4: **End**

5: **for**  $p \in D \mid \text{lab}(p) = \text{MASK}$  **do**

6:      $[L, \text{label}] \leftarrow \text{STREAM}(p)$  // Busca o stream do pixel

7:     **if**  $\text{label} = -1$  **then**

8:         // L é um stream mínimo (inf-stream)

9:          $\text{basins} \leftarrow \text{basins} + 1$

10:          $\text{label} \leftarrow \text{basins}$

11:     **end if**

12:     **for all**  $q \in L$  **do**

13:          $\text{lab}(q) \leftarrow \text{basins}$

14:     **end for**

15: **end for**

---

1: **Procedure** STREAM( $p$ )

2:      $L \leftarrow \{p\}$

3:      $L' \leftarrow \{p\}$  // Mínimos *bottoms* não explorados de  $L$

4:     **while**  $\exists u \in L'$  **do**

5:          $L' \leftarrow L' \setminus \{u\}$  // Mínimo explorado é removido

6:         BREADTH-FIRST  $\leftarrow \text{true}$

7:         **while** BREADTH-FIRST = **true** **and**  $(\exists \{u, v\} \in E \mid v \notin L \text{ and } F(\{u, v\}) = F^\ominus(u))$  **do**

```

8:           // Verifica na vizinhança do pixel aqueles cuja aresta
    forma o caminho de máxima inclinação (steepest descent)
9:   if lab(v) ≠ MASK then
10:    // O pixel do caminho já está rotulado, portanto retorna
     este rótulo
11:    return [L, lab(v)]
12:   else if F⊖(v) < F⊖(u) then
13:    // Encontrou um caminho de descida com um novo mínimo
14:    L ← L ∪ {v}
15:    L' ← {v}
16:    // Volta a buscar a partir dos mínimos
17:    BREADTH-FIRST ← false
18:   else
19:    // F⊖(v) = F⊖(u), portanto v também é um mínimo de L
20:    L ← L ∪ {v}
21:    L' ← L' ∪ {v}
22:    // Continua procurando por caminhos de descida na
     vizinhança
23:   end if
24:   end while
25: end while
26: // Nenhum rótulo encontrado
27: return [L, -1]
28: End

```

A implementação deste algoritmo não requer estruturas de dados complexas, como listas ou filas. Entretanto, é necessária alguma metodologia para armazenamento do grafo que é processado, bem como formas rápidas de acesso às arestas e valores de altura mínima. Seu funcionamento é bastante intuitivo: para cada *pixel*, busca-se um caminho de máxima inclinação, procurando em profundidade, e alternando para busca em largura em zonas planas. Em relação à rotulação, ao encontrar um vértice já rotulado no caminho de máxima inclinação, este rótulo será utilizado; caso contrário, a busca em largura permite que o caminho detecte as zonas planas que de fato são mínimos regionais, de modo a rotulá-las unicamente. No Alg. 16, o conjunto  $L$  representa o caminho de máxima inclinação, enquanto  $L'$  representa o conjunto de vértices que estão na fronteira de expansão. A Fig. 3.19 apresenta em (b) um dos resultados possíveis do *watershed cut* para a imagem (a) com seu grafo correspondente em (c), ressaltando os cortes feitos com arestas tracejadas. Nesta figura também é mostrada a árvore de predecessores gerada pela propagação dos rótulos.

É importante ressaltar que, devido à diferença causada pela definição de mínimos regionais nas arestas de um grafo valorado, pode-se ter diferenças no número de regiões identificadas em comparação com os outros algoritmos apresentados neste capítulo. No entanto, analisando-se as condições em que o problema de fusão de mínimos ocorre, este está restrito a mínimos regionais separados por uma zona plana, onde o valor dos *pixels* nesta zona plana é o valor mínimo entre os vizinhos do mínimo

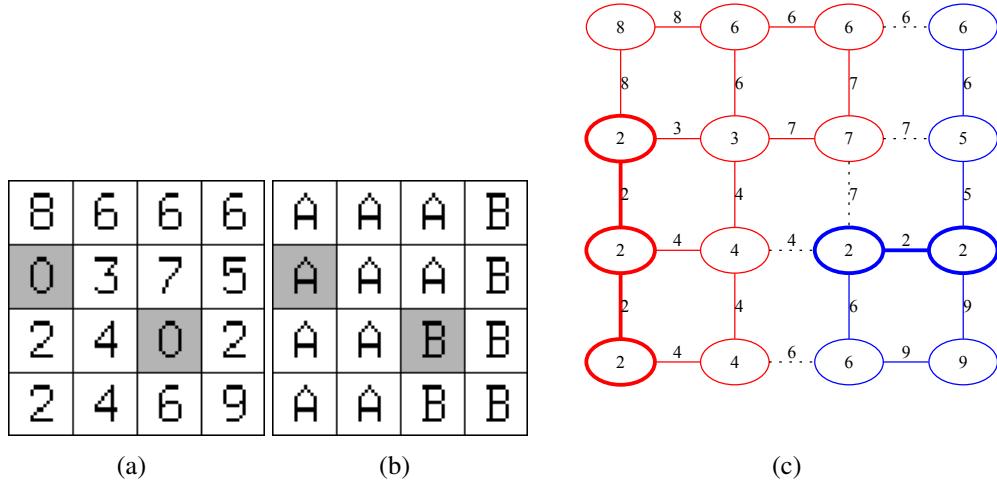


Fig. 3.19: Watershed Cut. (a) Imagem, (b) Resultado, (c) Grafo correspondente (N4)

regional. Esta análise é válida quando usada a função de máximo para valoração das arestas. No caso da função de mínimo, o problema é ainda mais restrito, pois a zona plana deve ter apenas um *pixel* para que ocorra a fusão dos mínimos regionais.

A programação do algoritmo permite tomar medidas de modo a se evitar repetidas varreduras do conjunto  $L$  para verificar se um vértice está contido neste. Isto é efetuado utilizando um rótulo distinto na imagem de saída, representando que o *pixel* pertence ao conjunto  $L$ . Após concluir o cálculo do *stream*, este é sobreescrito pelo rótulo definitivo do *pixel*. Em relação à estrutura de dados para armazenamento do grafo, duas alternativas podem ser consideradas clássicas: uso de matriz de adjacências ou listas de vértices. A primeira opção, apesar de rápida, é inviável por sua necessidade de memória (e.g. para uma imagem de 256x256 *pixels*, seria necessária uma matriz quadrada de lado 65.536, correspondendo a aproximadamente 4 bilhões de elementos). Desta forma, devido ao número limitado de arestas adjacentes a cada vértice, opta-se pelo uso de listas, ou dicionários, para indexação dos vértices, assim como nos algoritmos *Union-Find* e Tobogã Invariante a Ordem.

### 3.14.1 Implementação Python

```
from common import *

# constants
MASK = -2
IN\_SET = -1

def watershedCut(im, offsets):

    # initialise variables
```

```

ws = wsImage(im)
N, im, lab, D = ws.begin(offsets)

basins = 0

max\_value = max(im) + 1

# build the graph
graph = dict() # use a dict
for p in D:

    lab[p] = MASK # "mask" the image
    fminus = max\_value
    edges = [] # create the edges
    for q in N(p):
        m = max(im[p], im[q])
        edges.append({ 'w': m, 'index': q })
        if m < fminus: # minimize the fminus value of the vertex
            fminus = m

    graph[p] = { 'fminus': fminus, 'edges': edges } # append the edges and
fminus dict to the graph

# iterate on the graph/label image
for p in D:
    if lab[p] == MASK:
        # make a stream on the pixel
        L, label = stream(p, graph, lab)
        # generate a new label if necessary
        if label == -1:
            basins += 1
            label = basins
        # set the found/generated label on the set
        for q in L:
            lab[q] = label

# crop and return the labeled image
return ws.end()

# define the stream function
def stream(p, graph, lab):
    L = [p]
    Li = [p]
    lab[p] = IN\_SET # mark for signalizing the pixel is in the set

    while len(Li) > 0:
        u = Li.pop()
        breadth\_first = True
        fminus = graph[u][ 'fminus' ]

```

```
edges = graph[u][ 'edges' ]  
  
while breadth\_first:  
    edge\_found = False  
    for e in edges:  
        v = e[ 'index' ]  
        w = e[ 'w' ]  
        if lab[v] != IN\_SET and w == fminus:  
            edge\_found = True  
            break  
  
    if not edge\_found:  
        break  
  
    if lab[v] != MASK:  
        return (L, lab[v])  
    elif graph[v][ 'fminus' ] < fminus:  
        L.append(v)  
        lab[v] = IN\_SET  
        Li = [v]  
        breadth\_first = False  
    else:  
        L.append(v)  
        lab[v] = IN\_SET  
        Li.append(v)  
  
return (L, -1)
```



# Capítulo 4

## Análise Crítica dos Algoritmos

Neste capítulo são realizadas diversas análises dos algoritmos de transformada *watershed* explorados anteriormente. Inicia-se por algumas considerações a respeito dos resultados dos algoritmos em zonas planas, formas de resolução destas quando isto é executado, e suas relações com as definições. Também é realizado um estudo sobre o impacto das diferenças entre as definições em aplicações práticas. Em seguida, faz-se uma análise comparativa, verificando características comuns, como a forma de visitação dos *pixels*, técnicas utilizadas para endereçamento na matriz, para construir e percorrer caminhos. Por último, uma análise de desempenho é feita, comparando-se relativamente os tempos medidos para execução em um grupo de imagens. Nesta análise busca-se apontar possíveis gargalos nos algoritmos e técnicas que poderiam ser combinadas para atingir mais velocidade. Entre estas técnicas, reservou-se uma seção especial para discussão do paralelismo.

### 4.1 Análise Comparativa de Resultados

#### 4.1.1 Resolução de Zonas Planas

O tratamento das zonas planas compõe uma parte importante do cálculo da transformada *watershed*, por serem regiões das imagens onde os custos dos caminhos tornam-se iguais e requerem alguma ação dos algoritmos. Pode-se classificar estas ações em: *lower completion*; propagação FIFO de rótulos a partir das bordas; cálculo de custo lexicográfico; propagação aleatória. Nesta seção, analisa-se a influência destas alternativas no resultado da transformada *watershed*. A operação de *lower completion* é utilizada no pré-processamento da imagem com o intuito de poder-se ignorar a existência de zonas planas, garantindo que para cada *pixel* exista um vizinho com nível de cinza menor, exceto nos mínimos regionais, e assim simplificar o algoritmo final. Quando este processo não é realizado, a técnica mais utilizada pelos algoritmos é o uso de filas, propagando-se rótulos a partir das bordas, realizando uma divisão uniforme destas, abdicando do custo da definição que se implementa. No entanto, implicitamente calcula-se o custo lexicográfico, que é tomado como segunda componente de custo, de desempate à primeira, simulando a velocidade constante de propagação de águas em superfícies planas [19]. Na literatura, ocorre uma confusão em relação ao uso destes conceitos para divisão correta de zonas planas, sendo tratados como equivalentes, quando não o são. Em paralelo a isto, a fila hierárquica é vista de forma geral como uma estrutura que implicitamente embute apenas o custo máximo, sendo a política FIFO responsável pelo custo lexicográfico. No entanto,

esta estrutura, utilizada como fundamento de diversos algoritmos, mesmo descartando-se a política FIFO, implementa, na teoria, um custo com mais componentes do que apenas o máximo, de difícil formulação.

Ao utilizar a definição de distância topográfica (TD-WT), onde o custo do caminho de um *pixel* até um mínimo regional é dado por uma soma, equivalente a se encontrar os menores vizinhos para cada *pixel* no cálculo do *downstream*, o uso ou de *lower completion* ou de custo lexicográfico equipara-se em resultado, como visto em diversos algoritmos que implementam as definições LC-WT e TD-WT. Especialmente comparando-se as abordagens de Meijster e Roerdink com o algoritmo *Union-Find*, onde a remoção de zonas planas via *lower completion* constitui a primeira etapa, e de Lin *et al.* com o algoritmo Tobogã Ordenado, onde as zonas planas são resolvidas utilizando uma fila para processamento a partir das bordas destas - implicitamente um custo lexicográfico. Dado que estes algoritmos implementam a mesma definição, produzindo os mesmos resultados, ambas formas podem ser ditas como equivalentes.

Entretanto, estas não são equivalentes quando utiliza-se como função de custo o máximo no caminho, base da definição IFT-WT, sendo que apenas o uso do custo lexicográfico conjunto ao máximo produz os resultados esperados. Para demonstrar esta hipótese é necessária a modificação dos algoritmos de Berge - para cálculo explícito dos custos -, IFT - para demonstração do papel da fila hierárquica - e Zona de Empate - para verificação de coerência das soluções.

### Alg. Berge-Max

O algoritmo Berge-Max baseia-se no algoritmo de florestas de caminhos mínimos proposto por Berge [43] com implementações por Meyer [7] e Roerdink e Meijster [12] (Alg. 6). No entanto, estas propostas utilizam a função de custo do caminho como a soma dos custos dos arcos neste, independente da forma de cálculo do custo dos arcos. No entanto, esta não é uma restrição do algoritmo, que depende apenas da convergência das funções para produzir o resultado correto. Assim sendo, propõe-se no Alg. 17 o uso da função de custo máximo, para cálculo da SPF.

#### ALGORITMO 17: Berge-Max

**Entrada:**  $im$ : Imagem de níveis de cinza, com domínio  $D$  e mínimos regionais  $m_i \in M$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D, lab(p) \leftarrow MASK, C_1(p) \leftarrow \infty$ 
3:    $\forall p \in m_i \in M, lab(p) \leftarrow i, C_1(p) \leftarrow im(p)$ 
4:   stable  $\leftarrow$  true
5: End
6: repeat
7:   stable  $\leftarrow$  true
8:   for  $p \in D^+$  do
9:     PROPAGATE( $p, N^+(p)$ )
10:    end for

```

```

11:   for  $p \in D^-$  do
12:     PROPAGATE( $p, N^+(p)$ )
13:   end for
14:   for  $p \in D$  do
15:     if lab(parent( $p)) \neq lab(p)$  then
16:       lab( $p) \leftarrow lab(parent(p))$ 
17:       stable  $\leftarrow$  false
18:     end if
19:   end for
20: until stable = true

21: Procedure PROPAGATE( $p, Q$ )
22:   for all  $q \in Q$  do
23:      $c \leftarrow max(C_1(p), im(q))$ 
24:     if  $c < C_1(q)$  then
25:        $C_1(q) \leftarrow c$ 
26:       parent( $q) \leftarrow p$ 
27:       stable  $\leftarrow$  false
28:     end if
29:   end for
30: End

```

Podem-se identificar duas diferenças do Alg. 17 em relação ao Alg. 6: (1) calcula-se o custo diretamente pelo máximo entre o custo atual e o valor do próximo *pixel* no caminho, formando assim o caminho de custo máximo; (2) faz-se necessária a utilização explícita da relação de antecessor, para uso na rotulação, através da matriz de endereços **par**. Esta matriz é necessária, pois a atualização dos antecessores ocorre a cada varredura realizada, minimizando os custos, sendo que em determinados pontos o antecessor de um *pixel* sofre alteração de rótulo sem que o *pixel* em questão seja alterado, pois seu custo já é mínimo. Desta forma, os rótulos necessitam ser determinados pela matriz **par**, para que sejam corretamente atualizados, de acordo com a SPF calculada.

O fato de não utilizar nenhum tipo de estrutura de dados exceto a matriz  $C_1$  para armazenamento do custo mínimo obtido em cada *pixel* garante que este algoritmo implementa apenas a função de máximo como custo do caminho, não tendo sua visitação guiada por nenhum tipo de prioridade que possa guiar os resultados. Esta característica também permite que o algoritmo produza soluções diferenciadas, todavia corretas. Ainda em relação às soluções possíveis, o Alg. 17 não adere diretamente a nenhuma definição, não produzindo *pixels* de *watershed*, apenas bacias de captação rotuladas condizendo com uma SPF onde o custo do caminho é o máximo dos arcos neste e as raízes são os mínimos regionais.

```

from common import *

# constants
MASK = -2

def berge\_max(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    def propagate(p, Q, cmp):
        s = True
        for q in Q:
            if not cmp(q, p):
                continue

            c = max(c1[p], im[q])
            if c < c1[q]:
                c1[q] = c
                par[q] = p
                s = False

        return s

    lab[:] = MASK
    c1 = ws.makeWorkCopy(inf)
    par = ws.makeWorkCopy(-1)
    stable = False

    D = list(D)

    for m in xrange(len(M)):
        for p in M[m]:
            lab[p] = m+1
            c1[p] = im[p]

    while not stable:
        stable = True
        for p in D:
            stable = propagate(p, N(p), lambda u,v: u > v) and stable

            if par[p] != -1 and lab[p] != lab[par[p]]:
                lab[p] = lab[par[p]]
                stable = False

        for p in reversed(D):
            stable = propagate(p, N(p), lambda u,v: u < v) and stable

```

```

if par[p] != -1 and lab[p] != lab[par[p]]:
    lab[p] = lab[par[p]]
    stable = False

return ws.end()

```

## Implementação Python

### Alg. Berge-MaxLex

Assim como o algoritmo Berge-Max (Alg. 17), o algoritmo Berge-MaxLex é baseado no método de construção de SPF de Berge [43]. De fato, o segundo é construído com base no primeiro, incluindo-se a componente de custo lexicográfico quando há empate da componente de máximo. Como a varredura do algoritmo Berge pode ser realizada aleatoriamente, o custo lexicográfico é calculado até sua convergência, sendo armazenado na matriz  $C_2$ . Tomam-se como regras para cálculo deste: (1) qualquer *pixel* para o qual existir um *pixel* de valor menor na vizinhança terá custo lexicográfico zero, (2) em caso de empate no custo máximo, o custo lexicográfico do *pixel* vizinho será o custo lexicográfico do *pixel* atual mais um, caso este seja menor que o custo lexicográfico atual do *pixel* vizinho. A partir destas, escreve-se o Alg. 18.

#### ALGORITMO 18: Berge-MaxLex

**Entrada:**  $im$ : Imagem de níveis de cinza, com domínio  $D$  e mínimos regionais  $m_i \in M$

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \in D, lab(p) \leftarrow MASK, C_1(p) \leftarrow \infty, C_2(p) \leftarrow 0$ 
3:    $\forall p \in m_i \in M, lab(p) \leftarrow i, C_1(p) \leftarrow im(p)$ 
4:   stable  $\leftarrow$  true
5: End
6: repeat
7:   stable  $\leftarrow$  true
8:   for  $p \in D^+$  do
9:     PROPAGATE( $p, N^+(p)$ )
10:   end for
11:   for  $p \in D^-$  do
12:     PROPAGATE( $p, N^+(p)$ )
13:   end for
14:   for  $p \in D$  do
15:     if  $lab(parent(p)) \neq lab(p)$  then
16:        $lab(p) \leftarrow lab(parent(p))$ 

```

```

17:         stable ← false
18:     end if
19: end for
20: until stable = true

21: Procedure PROPAGATE( $p, Q$ )
22:   for all  $q \in Q$  do
23:      $c \leftarrow \max(C_1(p), \text{im}(q))$ 
24:     if  $c < C_1(q)$  then
25:        $C_1(q) \leftarrow c$ 
26:       parent(q) ← p
27:       stable ← false
28:       if  $c = C_1(p)$  then
29:          $C_2(q) \leftarrow C_2(p) + 1$ 
30:       else
31:          $C_2(q) \leftarrow 0$ 
32:       end if
33:       else if  $c = C_1(q)$  then
34:         if  $c = C_1(p)$  then
35:           if  $C_2(p) + 1 < C_2(q)$  then
36:              $C_2(q) \leftarrow C_2(p) + 1$ 
37:             parent(q) ← p
38:             stable ← false
39:           end if
40:           else if parent(q) ≠ p and  $C_2(q) \neq 0$  then
41:              $C_2(q) \leftarrow 0$ 
42:             parent(q) ← p
43:             stable ← false
44:           end if
45:         end if
46:       end for
47: End

```

Nota-se que o Algoritmo 18 expande o Alg. 17, incluindo o cálculo explícito do custo lexicográfico após a avaliação do custo máximo. Desta forma, implementa-se a definição IFT-WT utilizando varredura aleatória na imagem. Da mesma maneira que o Alg. 17, a rotulação dos *pixels* é feita através de seus antecessores, pelos mesmos motivos apresentados anteriormente, que não são removidos com a introdução da segunda componente de custo. É importante ressaltar a necessidade da regra (1) exposta anteriormente, implementada no algoritmo na sua inicialização e nas linhas 31 e 41. A reinitialização do valor garante que este custo será calculado corretamente, pois não se pode antecipar o comportamento da varredura, e, por consequência, a direção de crescimento do custo lexicográfico.

```
from common import *

# constants
MASK = -2

def berge\max\lex(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    def propagate(p, Q, cmp):
        s = True
        for q in Q:

            if not cmp(q, p):
                continue

            c = max(c1[p], im[q])

            if c < c1[q]:
                c1[q] = c
                par[q] = p
                s = False
            if c == c1[p]:
                c2[q] = c2[p] + 1
            else:
                c2[q] = 0
            elif c == c1[q]:
                if c == c1[p]:
                    if c2[p] + 1 < c2[q]:
                        c2[q] = c2[p] + 1
                        par[q] = p
                        s = False
                elif par[q] != p and c2[q] != 0:
                    c2[q] = 0
                    par[q] = p
                    s = False

        return s

    lab[:] = MASK
    c1 = ws.makeWorkCopy(inf)
    c2 = ws.makeWorkCopy(0)
    par = ws.makeWorkCopy(-1)
    stable = False

    D = list(D)
```

```

for m in xrange(len(M)):
    for p in M[m]:
        lab[p] = m+1
        c1[p] = im[p]

while not stable:
    stable = True
    for p in D:
        stable = propagate(p, N(p), lambda u,v: u > v) and stable

        if par[p] != -1 and lab[p] != lab[par[p]]:
            lab[p] = lab[par[p]]
            stable = False

    for p in reversed(D):
        stable = propagate(p, N(p), lambda u,v: u < v) and stable

        if par[p] != -1 and lab[p] != lab[par[p]]:
            lab[p] = lab[par[p]]
            stable = False

return ws.end()

```

## Implementação Python

### Algoritmo IFT-Rand

Assim como o algoritmo Berge-Max, o algoritmo IFT-Rand busca - porém não atinge, por motivos explicados a seguir - uma implementação de transformada *watershed* aplicando unicamente o custo máximo no caminho, sendo baseado no algoritmo IFT [8]. No algoritmo IFT com custo lexicográfico, este não é calculado explicitamente, sendo inherente ao uso da fila de prioridade com política FIFO [19]. Desta forma, a estratégia adotada neste algoritmo foi remover a ordenação FIFO, permitindo que a remoção de elementos em um mesmo nível de prioridade torne-se independente da ordem de inserção. Outra estratégia possível é tornar a fila de prioridade com política LIFO. No entanto, implementações de filas de prioridade com quebra de desempate arbitrário são mais comumente incluídas em bibliotecas de linguagens de programação (e.g. *heapq* em Python, *priority\_queue* na STL em C++, *PriorityQueue* em Java).

Desta forma, alterando apenas o tipo de estrutura de dados do Alg. IFT (Alg. 10) pode-se degenerá-lo para o algoritmo IFT-Rand. Entretanto, o uso da prioridade como ordenação dos *pixels* a serem processados - que garante a otimalidade dos caminhos - implica em uma equação mais complexa que apenas o custo máximo do caminho. Este efeito ocorre pois os caminhos de custo  $k$  serão todos avaliados antes dos caminhos de custo  $k + 1$ . Isto implica que *pixels* de custo  $k + 1$  com vizinhos de custo  $k$ , nunca serão atingidos por outros *pixels* com custo  $k + 1$ , mesmo estes sendo também caminhos ótimos, do ponto de vista do custo máximo.

```
from common import *

# constants
MASK = -2

def ift\_\_rand(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    # create the working images
    done = ws.makeWorkCopy(False)
    c1 = ws.makeWorkCopy(inf)
    par = ws.makeWorkCopy(MASK)

    lab[:] = MASK

    queue = wsRandHeapQueue()

    for m in xrange(len(M)):
        for p in M[m]:
            c1[p] = im[p]
            lab[p] = m+1
            par[p] = p
            queue.push(p, im[p])

    while not queue.empty():
        p = queue.pop()
        done[p] = True
        for q in N(p):
            if done[q]:
                continue

            c = max(c1[p], im[q])
            if c < c1[q]:
                if c1[q] < inf:
                    if queue.contains(q, c1[q]):
                        queue.remove(q, c1[q])
                c1[q] = c
                lab[q] = lab[p]
                par[q] = p
                queue.push(q, c1[q])

    return ws.end()
```

### Algoritmo Zona de Empate Max

Tendo por base o algoritmo Zona de Empate sobre a definição IFT-WT [13], pode-se remover o custo lexicográfico e calcular também a zona de empate considerando-se apenas o custo máximo dos caminhos. No entanto, esta abordagem pode não ser considerada a mais apropriada, pois o fundamento do algoritmo Zona de Empate ou IFT é o algoritmo de Dijkstra [39] com a implementação de Dial [42], cujas propriedades garantem que os *pixels* removidos da fila de prioridade já estão com sua solução definitiva. Porém, para o cálculo da zona de empate com custo máximo, é necessário reinserir *pixels* com solução definitiva novamente na fila, de modo a propagar a zona de empate onde o custo for ótimo para os caminhos alternativos encontrados. Desta forma, a propriedade supracitada não é mais válida. Esta solução é apresentada no Alg. 19.

#### ALGORITMO 19: Zona de Empate Max

**Entrada:**  $im$ : Imagem de níveis de cinza com domínio  $D$ ,  $S$ : marcadores

**Saida:**  $lab$ : Imagem rotulada

```

1: Initialise
2:    $\forall p \notin S$ ,  $C_1(p) \leftarrow \infty$ ,  $lab(p) \leftarrow MASK$ ,  $parent(p) \leftarrow MASK$ 
3:    $\forall p \in S$ ,  $C_1(p) \leftarrow im(p)$ ,  $lab(p) \leftarrow \lambda(p)$ ,  $parent(p) \leftarrow p$ , HEAPQUEUEPUSH( $p$ ,  $im(p)$ )
4: End

5: while HEAPQUEUEEMPTY() = false do
6:    $p \leftarrow$  HEAPQUEUEPOP()
7:   for all  $q \in N(p)$  do
8:      $c \leftarrow max(C_1(p), im(q))$ 
9:     if  $c < C_1(q)$  then
10:      if HEAPQUEUECONTAINS( $q$ ) then
11:        HEAPQUEUEREMOVE( $q$ )
12:      end if
13:       $C_1(q) \leftarrow c$ 
14:       $lab(q) \leftarrow lab(p)$ 
15:       $parent(q) \leftarrow p$ 
16:      HEAPQUEUEPUSH( $q$ ,  $C_1(q)$ )
17:    else if  $c = C_1(q)$  and  $lab(q) \neq lab(p)$  then
18:       $lab(q) \leftarrow TIE-ZONE$ 
19:      HEAPQUEUEPUSH( $q$ ,  $C_1(q)$ )
20:    end if
21:  end for
22: end while
```

Pode-se ver no Alg. 19 que foram removidos o cálculo explícito da segunda componente do custo

do caminho, correspondente ao custo lexicográfico, e o uso da variável **done**, que indicava *pixels* cuja solução já era definitiva e não deveriam ser reprocessados. A reinserção dos *pixels* que recebem rótulo **TZ** na fila é executada na linha 19, permitindo assim que se descubra todos os caminhos de custo máximo na imagem.

```

from common import *

# constants
MASK = -2
TIE\_ZONE = 0

def tieZone\_max(im, offsets):

    # initialise variables
    ws = wsImage(im)
    N, im, lab, D = ws.begin(offsets)

    # find minima
    M = findMinima(im, N, D)

    # create the working images
    done = ws.makeWorkCopy(False)
    c1 = ws.makeWorkCopy(inf)
    par = ws.makeWorkCopy(MASK)
    lab[:] = MASK

    queue = wsHeapQueue()

    for m in xrange(len(M)):
        for p in M[m]:
            c1[p] = im[p]
            lab[p] = m+1
            par[p] = p
            queue.push(p, im[p])

    while not queue.empty():
        p = queue.pop()
        for q in N(p):

            c = max(c1[p], im[q])
            if c < c1[q]:
                if c1[q] < inf:
                    if queue.contains(q, c1[q]):
                        queue.remove(q, c1[q])
                    c1[q] = c
                    lab[q] = lab[p]
                    par[q] = p
                    queue.push(q, c1[q])
            elif c == c1[q] and lab[q] != lab[p]:

```

```

lab[q] = TIE\_ZONE
queue.push(q, c1[q])

return ws.end()

```

## Implementação Python

### Experimento

Neste experimento é utilizada uma imagem de tamanho  $10 \times 10$  pixels, constante em valor 1, com quatro mínimos regionais de valor 0, posicionados nas coordenadas (2,2), (2,9), (9,2), (9,9). O cálculo da operação de remoção de zonas planas é feito utilizando-se o Alg. 1. A Fig. 4.1 apresenta em (a) a imagem original -  $f$  - e em (b) a imagem sem zonas planas -  $lc$ , utilizando vizinhança-4.

1	1	1	1	1	1	1	1	1	1	1	7	6	7	8	9	9	8	7	6	7
1	0	1	1	1	1	1	1	1	0	1	6	0	6	7	8	8	7	6	0	6
1	1	1	1	1	1	1	1	1	1	1	7	6	7	8	9	9	8	7	6	7
1	1	1	1	1	1	1	1	1	1	1	8	7	8	9	10	10	9	8	7	8
1	1	1	1	1	1	1	1	1	1	1	9	8	9	10	11	11	10	9	8	9
1	1	1	1	1	1	1	1	1	1	1	9	8	9	10	11	11	10	9	8	9
1	1	1	1	1	1	1	1	1	1	1	8	7	8	9	10	10	9	8	7	8
1	1	1	1	1	1	1	1	1	1	1	7	6	7	8	9	9	8	7	6	7
1	0	1	1	1	1	1	1	1	0	1	6	0	6	7	8	8	7	6	0	6
1	1	1	1	1	1	1	1	1	1	1	7	6	7	8	9	9	8	7	6	7

(a)

(b)

Fig. 4.1: Imagens utilizadas para experimento de resolução de zonas planas. (a) Original -  $f$ , (b) após remoção de zonas planas -  $lc$ .

Inicia-se apresentando as soluções que consideram a função de custo do caminho com dois componentes. O algoritmo IFT não realiza o cálculo explícito do custo lexicográfico, pois a política FIFO garante o comportamento adequado. Todavia, o algoritmo Berge-MaxLex necessita realizar o cálculo explícito, e, devido a sua varredura aleatória, processar estas distâncias até a estabilização com custo mínimo. Desta forma, com as duas componentes explícitas, o algoritmo Berge-MaxLex produz os mesmos resultados do algoritmo IFT. A Fig. 4.2 apresenta os resultados considerando custo máximo e lexicográfico para os algoritmos (a) IFT e (b) Berge-MaxLex, aplicados sobre a imagem  $f$ .

Aplicando estes mesmos algoritmos sobre a imagem  $lc$ , obtém-se os mesmos resultados, indicando que este processo não influencia no resultado quando as duas componentes são utilizadas. No entanto, removendo a componente lexicográfica e aplicando os algoritmos apenas com custo máximo

A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D

(a)

(b)

Fig. 4.2: Resultado dos algoritmos (a) IFT e (b) Berge-MaxLex aplicando custo combinado sobre imagem com zonas planas f

sobre a imagem  $f$ , os resultados destes serão fortemente dependentes das estratégias de visitação utilizadas. Assim, ao remover a política FIFO e o cálculo explícito do custo lexicográfico do algoritmo Berge-MaxLex, degenerando-os nos algoritmos IFT-Rand e Berge-Max respectivamente, obtemos os resultados da Fig. 4.3.

A	A	B	B	B	B	B	B	B	B
A	A	A	B	B	B	B	B	B	B
A	A	A	B	B	B	B	B	B	B
A	A	A	B	B	B	B	B	B	B
A	A	A	A	B	B	B	B	B	B
C	C	C	C	B	B	B	B	B	B
C	C	C	C	C	C	B	B	B	B
C	C	C	D	D	D	D	D	B	B
C	C	C	D	D	D	D	D	D	D
C	C	D	D	D	D	D	D	D	D

(a)

(b)

Fig. 4.3: Resultados dos algoritmos (a) IFT-Rand e (b) Berge-Max aplicando custo máximo sobre imagem com zonas planas f

Na Fig. 4.3 são geradas duas soluções distintas, porém válidas, considerando-se apenas o custo

do caminho como o máximo deste. A solução em (a) depende da forma de organização utilizada na implementação de uma fila de prioridade, não sendo possível prever qual o próximo *pixel* a ser analisado dentro de um mesmo nível de prioridade, dado que esta ordem é dependente da inserção e organização na memória. Já a solução em (b) é claramente direcionada da origem em (0,0) para o canto inferior direito. Isto ocorre pois o algoritmo Berge-Max utiliza a mesma matriz para processamento e armazenamento, e assim o rótulo do primeiro mínimo regional encontrado em (2,2) é propagado a quase todos os *pixels* da imagem, pois o custo máximo é constante nestes, formando uma solução válida. Alternando-se a ordem de varredura da imagem e da vizinhança, diferentes resultados podem ser encontrados, todos válidos.

Entretanto, ao aplicar estes algoritmos na imagem *lc*, obtém-se o efeito desejado para comprovação da hipótese inicial. Para obter a equivalência entre *lower completion* e custo lexicográfico usando custo máximo, seria necessário que o resultado do algoritmo Berge-Max na imagem *lc* fosse igual ao resultado do algoritmo Berge-MaxLex sobre a imagem *f*. No entanto, esta igualdade não é encontrada. A Fig. 4.4 apresenta os resultados dos algoritmos (a) IFT-Rand e (b) Berge-Max aplicados sobre a imagem *lc*. É interessante notar que o algoritmo IFT-Rand produz resultado igual ao algoritmo IFT, todavia isto se deve ao fato da dependência da fila de prioridade, que, na verdade, implica em uma equação mais complexa do que apenas o custo máximo.

A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
A	A	A	A	A	B	B	B	B	B
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D
C	C	C	C	C	D	D	D	D	D

(a)	(b)
-----	-----

Fig. 4.4: Resultado dos algoritmos (a) IFT-Rand e (b) Berge-Max aplicando custo máximo sobre imagem sem zonas planas *lc*

Com estes exemplos demonstra-se que, para o uso do custo máximo, não há equivalência de resultados entre *lower completion* e custo lexicográfico. Ocorre de fato uma redução na zona de empate, mas não a eliminação desta, como seria o desejável para a imagem *f*. A Fig. 4.5 apresenta as zonas de empate calculadas (a) na imagem *f* considerando custo combinado, e apenas custo máximo (b) na imagem *f* e (c) na imagem *lc*. Ao aplicar o custo combinado na imagem *lc* o resultado é o mesmo de (a).

De fato, o cálculo das zonas de empates nestes três casos reforça os exemplos anteriores, e de-

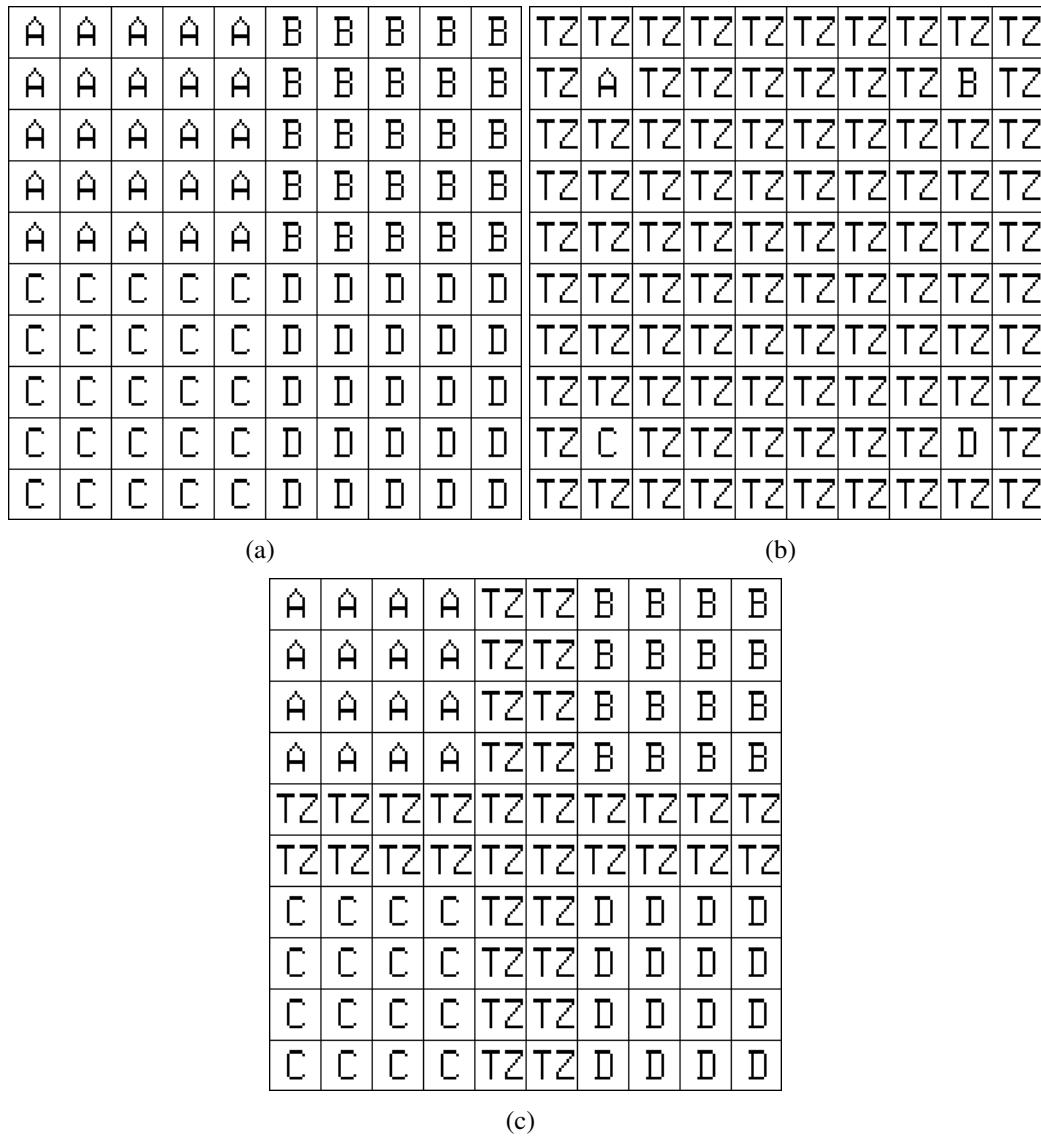


Fig. 4.5: Zonas de Empate com (a) custo máximo e lexicográfico sobre imagem com zonas planas f, e com custo máximo sobre (b) a imagem com zonas planas f e (c) sem zonas planas lc

monstra de forma independente o mesmo fato. A simples redução da zona de empate entre as imagens (b) e (c) e o fato de (c) não ser igual a (a), prova que a remoção de zonas planas através do processo de *lower completion* não equivale ao uso do custo lexicográfico como segunda componente de custo do caminho, onde a primeira é o máximo.

A partir destas conclusões pode-se verificar a ocorrência destas regiões problemáticas em imagens de aplicações reais. Para este efeito, utilizam-se as mesmas aplicações das Figs. 1.1 e 1.2, apresentadas na introdução deste trabalho. Deseja-se com estes exemplos ressaltar que o problema da resolução de zonas planas não é meramente teórico, e tem influência nos resultados obtidos da transformada *watershed*. A Fig. 4.6 apresenta o cálculo da zona de empate considerando apenas o

custo máximo sobre a imagem *beef* em (a), sobre esta após *lower completion* em (b) e a zona de empate considerando custo máximo e lexicográfico em (c). Calcula-se então a subtração entre (a) e (b) e temos as regiões da imagem onde o *lower completion* resolve a ambiguidade de soluções em (d). No entanto, para identificar a influência das zonas onde o *lower completion* não resolve a ambiguidade e permite soluções múltiplas enquanto o custo lexicográfico é único, subtrai-se da imagem (b) as zonas de empate calculadas com custo máximo e lexicográfico, conforme Alg. 12, apresentada em (c). Desta forma, os pontos de empate insolúveis por natureza são eliminados (e.g. picos de máximo separando regiões), restando as regiões em que apenas o custo lexicográfico é capaz de eliminar o empate, sendo apresentada em (e).

Nota-se na Fig. 4.6 (e) diversos pontos onde há diferença entre o resultado da aplicação de *lower completion* e custo lexicográfico. A mesma metodologia pode ser aplicada na imagem *csample*, de modo a identificar estas regiões problemáticas. Assim, na Fig. 4.7 são apresentadas as zonas de empate de custo máximo para a imagem *csample* em (a), e para esta sem zonas planas em (b). Em (c) é apresentada a zona de empate considerando custos máximo e lexicográfico. Em (d) é calculada a subtração entre (a) e (b), apresentando as regiões onde o *lower completion* foi eficiente, e em (e) as regiões onde este não é suficiente.

Para avaliar o impacto das regiões onde o *lower completion* é ineficiente na resolução de zonas planas, pode-se comparar a área destas com a área total da zona de empate, onde qualquer solução é válida. Na imagem *beef*, 25,7% dos *pixels* em empate estariam incorretos, não constituindo um empate real, assim como 13,1% dos *pixels* na imagem *csample*. Esta porcentagem reitera as conclusões anteriores, de que é necessário o uso de custo lexicográfico para divisão de zonas planas quando é utilizado o custo máximo, sendo um problema não apenas de definição, mas impactante nos resultados em imagens reais.

## Considerações

É importante ressaltar como principal resultado deste experimento que a equivalência entre o processo de remoção de zonas planas e a aplicação do custo lexicográfico depende da função de custo aplicada ao caminho, e que se deseja minimizar. O projeto de um algoritmo deve considerar este resultado na técnica utilizada para resolução de zonas planas, de modo a produzir soluções consistentes. A escolha do método depende, de forma geral, da definição a ser implementada e da forma do algoritmo, recaindo em 4 possibilidades: (1) propagação da borda através de fila, (2) *lower completion* e uso de fila de prioridade ou seleção do menor vizinho, (3) uso de fila de prioridade com política FIFO, e (4) cálculo explícito do custo lexicográfico.

Cada método tem suas implicações. A implementação de uma fila não é uma tarefa trivial e sua eficiência depende da forma como esta é realizada e do objetivo que se deseja cumprir. No entanto, o processo de *lower completion* também depende de filas para calcular explicitamente o custo lexicográfico e recalcular os valores dos *pixels*. O cálculo explícito, realizado como no algoritmo Berge-MaxLex, não depende de estruturas de dados, porém, são feitas diversas varreduras até estabilização destes valores. Desta forma, um balanço entre as alternativas deve ser feito, de modo que o método escolhido seja o mais apropriado para as características do algoritmo, entretanto considera-se que o tratamento de zonas planas é essencial para garantir a consistência entre a definição e a implementação dos algoritmos de transformada *watershed*.

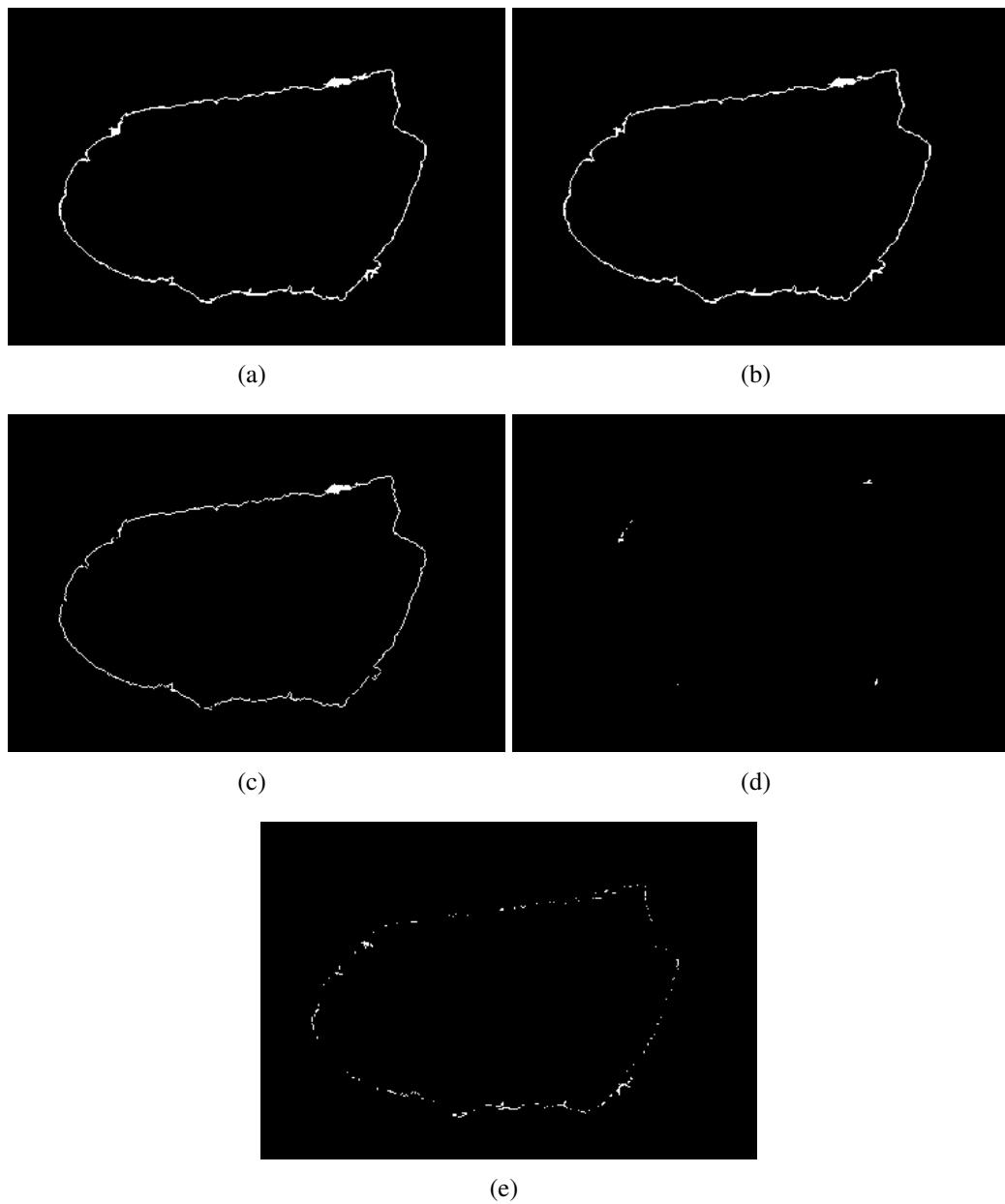


Fig. 4.6: Zonas de empate de custo máximo sobre (a) imagem beef e (b) beef sem zonas planas. (c) Zona de empate de custo máximo e lexicográfico sobre imagem beef. (d) Subtração entre (a) e (b), indicando pontos resolvidos por lower completion. (e) Subtração entre (b) e (c), indicando pontos onde apenas o custo lexicográfico resolve o empate

#### 4.1.2 Aplicações Práticas

Nesta seção comparam-se os algoritmos de *watershed* por seus resultados em aplicações práticas. Para tal efeito, utilizam-se os mesmos exemplos apresentados na introdução deste trabalho, nas Figs. 1.1 e 1.2, apresentando os resultados para os diferentes algoritmos. Desta forma, busca-se ressaltar

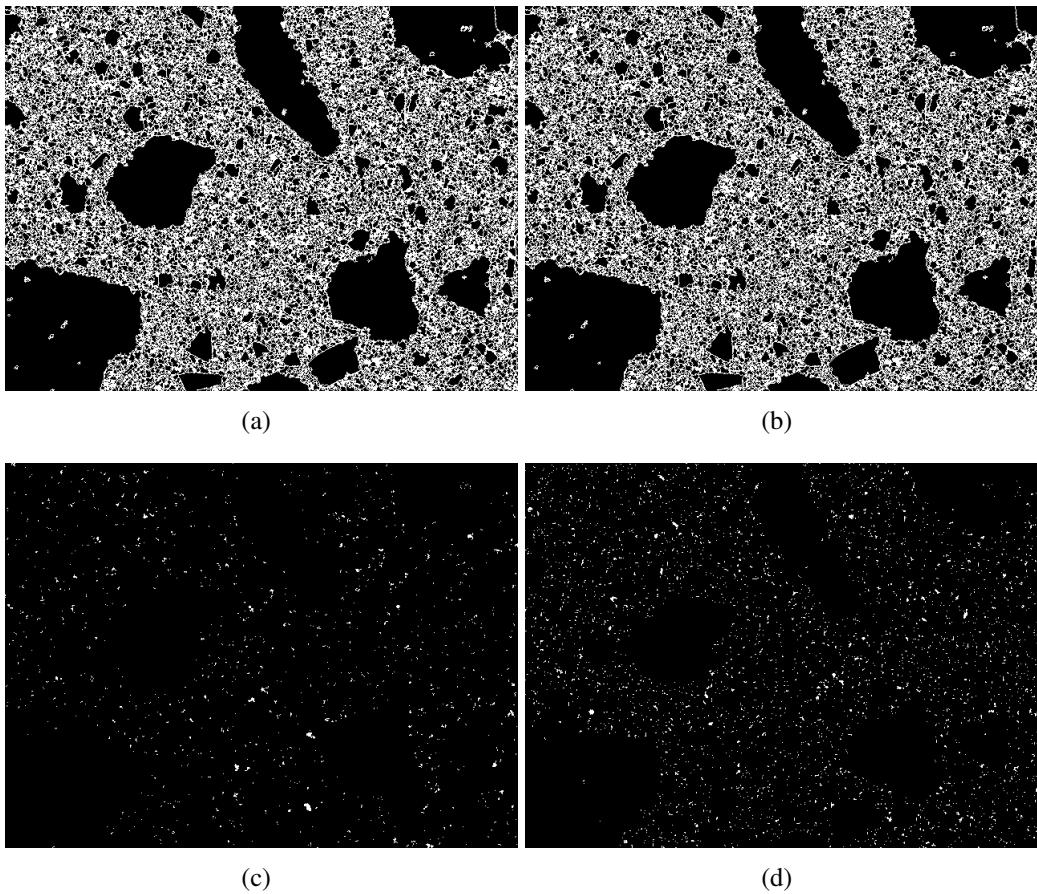


Fig. 4.7: Zonas de empate de custo máximo sobre (a) imagem csample e (b) csample sem zonas planas. (c) Zonas de empate de custo máximo e lexicográfico sobre imagem csample. (d) Subtração entre (a) e (b). (e) Subtração entre (b) e (c).

a sutileza nas diferenças entre os resultados, que em aplicações práticas podem ou não causar perdas de qualidade. Considerando-se que cada um dos algoritmos está associado a uma definição da transformada, serão apresentados apenas os resultados para cada uma das diferentes possibilidades. Um cuidado deve ser tomado em relação à definição Flooding-WT, que por não ter nenhum algoritmo que a implemente corretamente, é representada neste experimento pelo algoritmo de Imersão.

Assim, inicia-se com a primeira aplicação, da imagem *beef*, onde deseja-se segmentar uma região interior utilizando uma filtragem por marcadores e processamento do *watershed*. A Fig. 4.8 apresenta as etapas aplicadas à imagem original. Em (a) é mostrada a imagem *beef*; (b) calcula-se um filtro de fechamento com disco de raio 2 para eliminação de ruídos; (c) limiariza-se a imagem em nível 10 para separar o bife do fundo; (d) aplica-se filtro de fechamento por área para eliminar ruídos internos; (e) calcula-se o gradiente para gerar o marcador externo; (f) faz-se a erosão para gerar o marcador interno; (g) executa a união dos marcadores; (h) calcula-se gradiente da imagem (b) com marcadores como mínimos regionais; (i) filtra-se os mínimos regionais da imagem através da sua reconstrução morfológica pelos marcadores em (g).

Com base na imagem (i) da Fig. 4.8 calcula-se o *watershed*, normaliza-se seus rótulos e se obtém

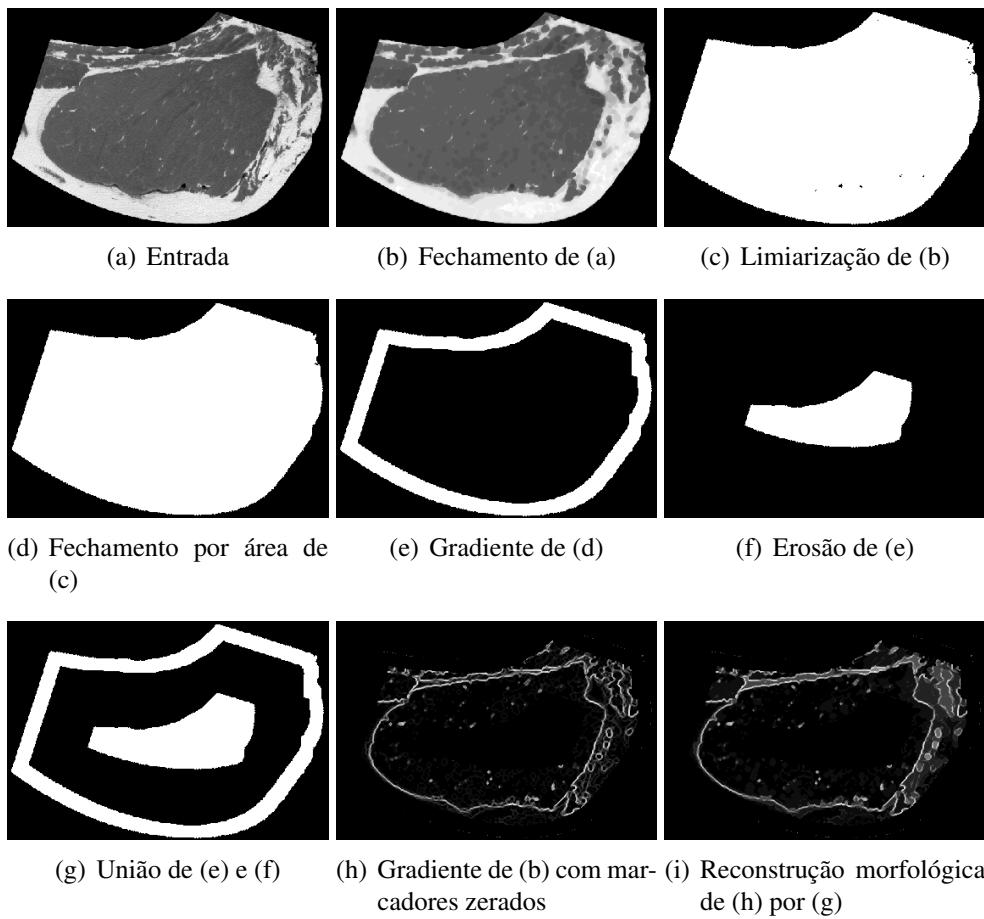


Fig. 4.8: Etapas da aplicação beef para geração da imagem para segmentação por transformada watershed utilizando marcadores indiretamente

o contorno interno da região de interesse. A Fig. 4.9 apresenta em (a) a imagem e marcadores e as diferentes definições em (b) - (g), uma imagem de diferença simétrica entre todas elas em (h), e uma imagem da união de todos os contornos rotulados em (i). É importante ressaltar que os contornos apresentados em (b) - (g) foram dilatados para melhor visualização.

Avaliando-se os resultados das imagens (b) - (g) da Fig. 4.9 pode-se dizer que são todos aceitáveis e equivalentes para esta aplicação, onde deseja-se calcular a área interna do bife, obtendo média de 35313,5 pixels com variação máxima de 0,3% para mais e 0,9% para menos. A análise da imagem (h) revela que as diferenças entre estes contornos são mínimas reforçando a variação obtida, exceto em regiões críticas, sendo este resultado embasado por uma análise da imagem (i), onde a sobreposição dos contornos é evidente, restando apenas alguns elementos em desacordo. Entretanto, dependendo da aplicação e da margem de erro utilizada, tais diferenças devem ser consideradas, optando-se então por abordagens mais ou menos restritivas a respeito do tamanho de regiões e formas de obtenção de contorno de uma região definida.

A segunda aplicação mostrada na introdução utiliza o *watershed* como um detector de texturas, baseando-se no tamanho das regiões identificadas. Desta forma, o experimento é repetido, aplicando-

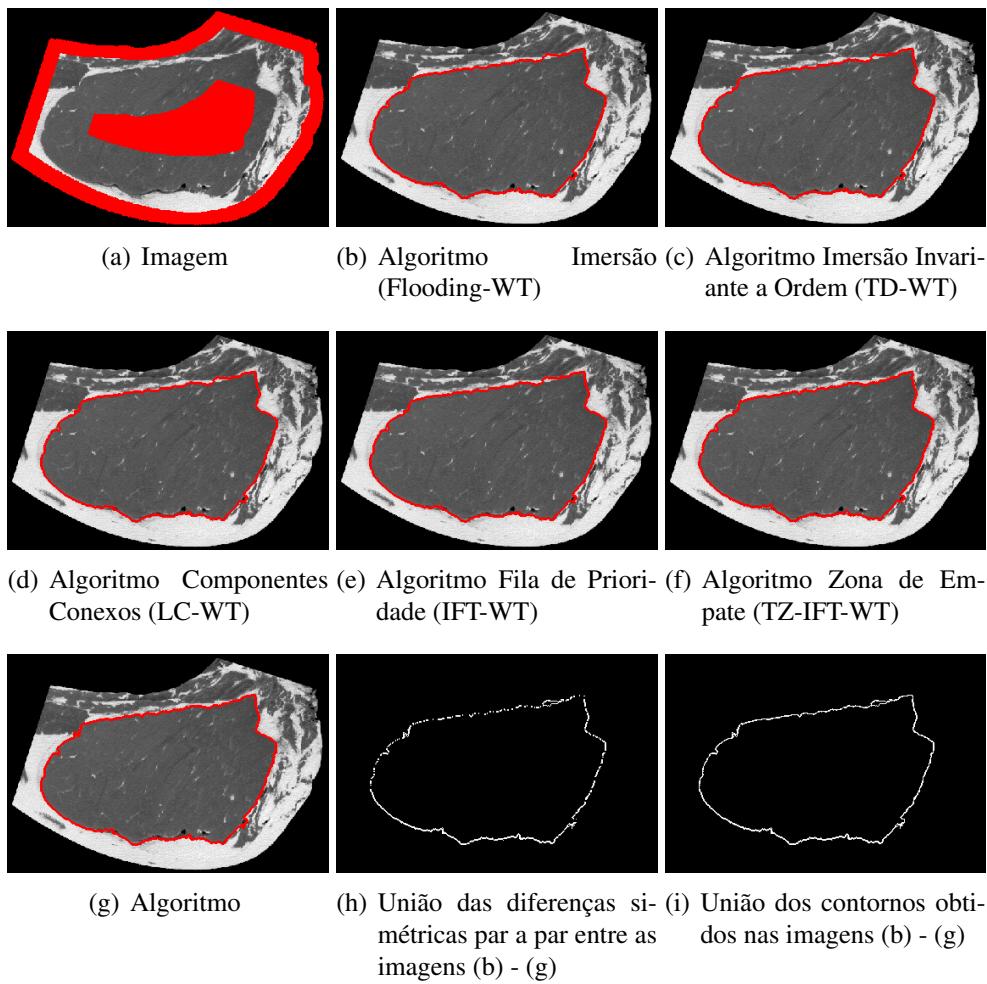


Fig. 4.9: Comparação de resultados das definições através de algoritmos aplicados na imagem beef preparada por marcadores

se os mesmos algoritmos, e comparando os resultados através do resultado esperado da aplicação, a medida de área total, e das diferenças e uniões entre os contornos das regiões identificadas. A Fig. 4.10 apresenta as etapas para extração das regiões de interesse: (a) imagem *csample*; (b) gradiente morfológico filtrado por dinâmica em nível 10; (c) transformada *watershed* rotulada; (d) filtragem das regiões por área maior ou igual a 300; (e) fechamento por área para eliminação de ruídos internos; (f) contornos internos das regiões obtidas.

A Fig. 4.11 apresenta as imagens produzidas pelos algoritmos, variando-se o algoritmo de transformada *watershed* utilizado na etapa (b) da Fig. 4.10, e mantendo as outras operações intactas. Em (a) é apresentada a imagem e as regiões de interesse numeradas de 1 a 5; (b) - (g) apresenta as diferentes definições; (h) diferença simétrica entre todas as definições; (i) união dos contornos.

Nesta aplicação deseja-se medir um tipo específico de região homogênea, de característica mais escura, e que forma blocos maiores. Como medida de comparação entre os resultados, pode-se usar o número total de regiões encontradas, visto que em todos os resultados as 5 regiões de interesse

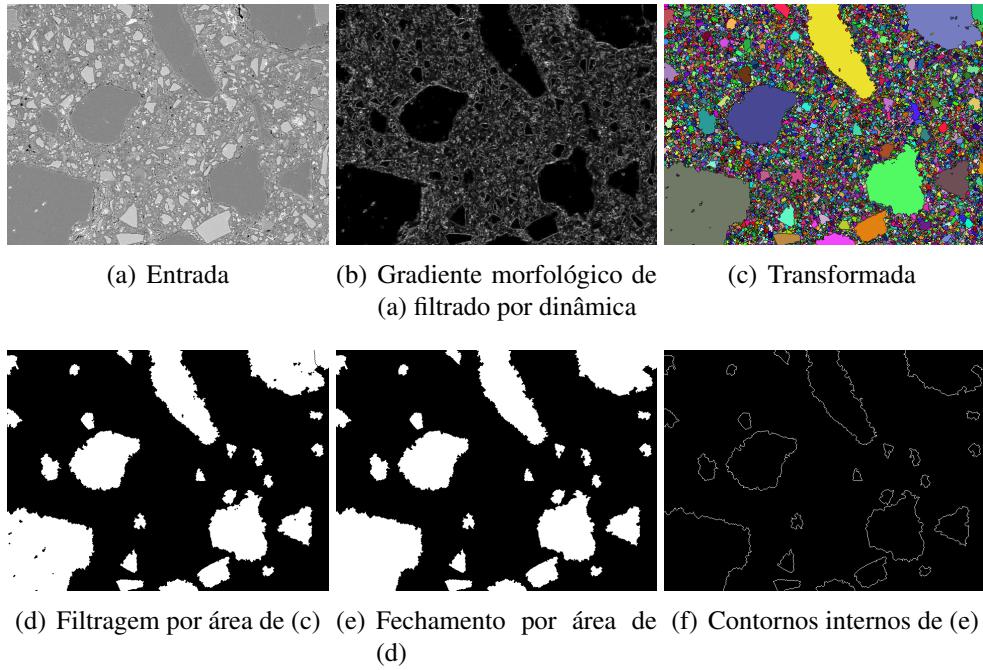


Fig. 4.10: Etapas da aplicação concreta para segmentação de regiões homogêneas

foram identificadas, e a área destas 5 regiões. Desta forma, respectivamente os algoritmos (b) - (g) encontraram 26, 27, 28, 28, 23 e 30 regiões. Em relação às regiões foram medidas áreas médias e a variação percentual máxima para mais e para menos, obtendo os valores apresentados na Tab. 4.1. Da mesma forma que o exemplo da aplicação anterior, as imagens de diferenças simétricas em (h) e união dos contornos em (i) apresentam resultados muito próximos entre as definições. De

Região	Área média	Var. Positiva	Var. Negativa
1	12749,66	2,3%	3,5%
2	11343,5	1,9%	2,8%
3	11030,16	1,8%	2,9%
4	11041,83	6,2%	4,0%
5	22776	0,3%	1,0%

Tab. 4.1: Medidas sobre a aplicação de deteção de regiões homogêneas

forma geral, a análise dos resultados aponta para consistência entre as soluções, sendo algumas mais restritivas em relação à dimensão das regiões produzidas, como a definição TZ-IFT-WT que produz regiões menores devido à zona de empate, e outras mais amplas, produzindo regiões maiores, como a definição WC-WT. Tais considerações devem ser levadas em conta quando a transformada *watershed* for utilizada em situações com margens de erro baixas. Em outros problemas, pode ser necessária uma avaliação mais criteriosa, considerando qual abordagem é mais apropriada, especialmente em relação ao comportamento das definições, visto que os algoritmos são fortemente relacionados a estas

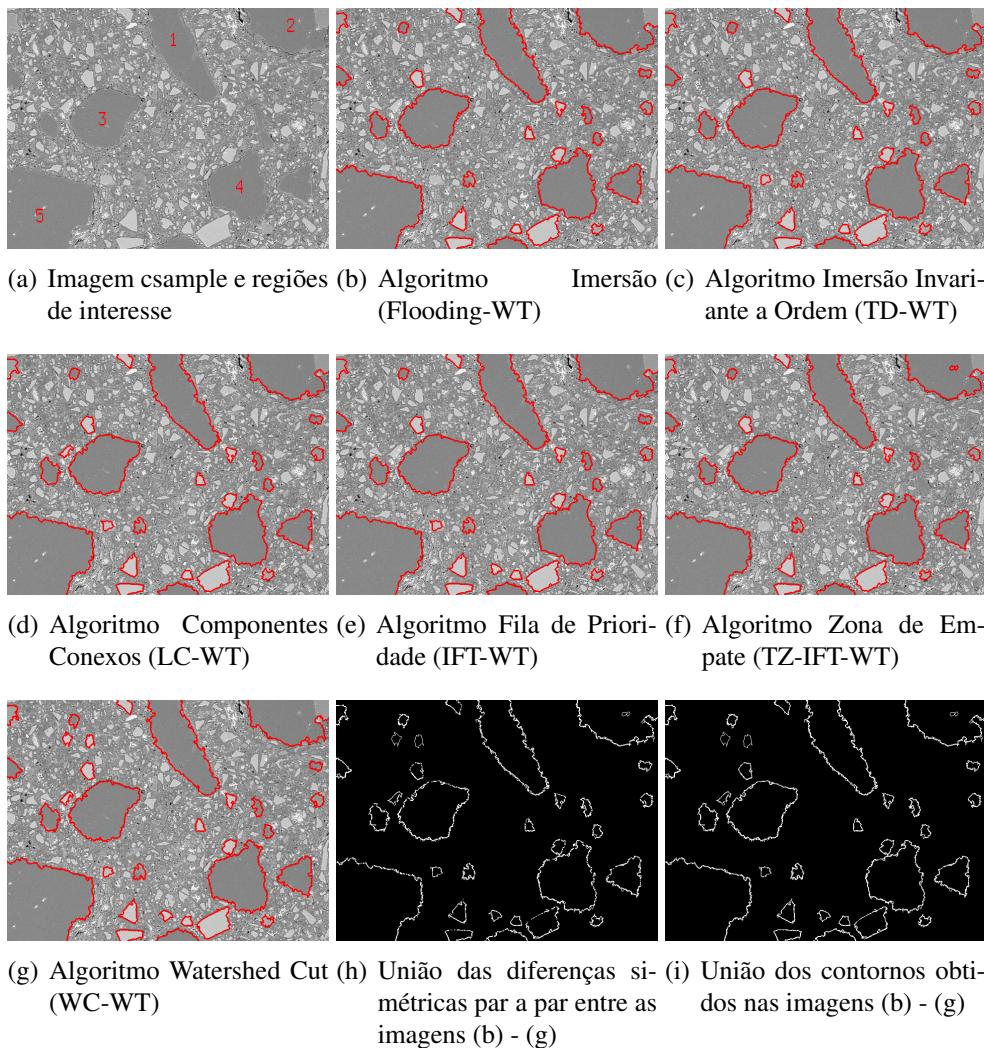


Fig. 4.11: Comparação de resultados das definições de watershed através de algoritmos, na aplicação de identificação de regiões homogêneas

nas soluções que oferecem. Entretanto, em casos onde ocorre um pré-processamento mais intenso, especialmente com a filtragem de mínimos regionais, estas diferenças tendem a ser grandemente diminuídas, como no primeiro exemplo apresentado, onde a variação de área medida foi inferior a 1%, representando um caso de uso mais comum da transformada.

## 4.2 Análise Comparativa das Implementações

Nesta seção os algoritmos apresentados no Cap. 3 são comparados nas técnicas empregadas para realizar algumas tarefas comuns. São discutidos os métodos de exploração da imagem, endereçamento dos caminhos de gota d'água, rotulação dos caminhos, descoberta e rotulação de mínimos regionais, e a influência do uso de caminhos múltiplos para cada *pixel* em contraste com caminhos

únicos.

### 4.2.1 Exploração da Imagem

A análise de exploração da imagem realizada neste trabalho busca caracterizar os algoritmos de acordo com a estratégia utilizada para a rotulação dos *pixels*. Ao expor esta análise pretende-se criar guias para a compreensão dos algoritmos. Entre as formas de exploração da imagem identificadas, há duas vertentes representativas e ainda outra alternativa pouco utilizada. Adotamos aqui a nomenclatura utilizada por Cormen *et al.* [26] e clássica na avaliação de algoritmos de busca em grafos: busca em largura e busca em profundidade. A terceira linha constitui-se da varredura aleatória, onde não é imposta nenhuma ordem no acesso aos *pixels*, representada unicamente no âmbito da transformada *watershed* pelo algoritmo de Berge de construção de florestas de caminhos mínimos. Esta representação única não implica baixo nível de importância, pois devido a este fato o algoritmo é livre de implicações inerentes ao uso de estruturas de dados para controlar a forma de varredura. Entretanto, por ser único e já explorado em vários aspectos neste trabalho, serão consideradas apenas as abordagens em largura e profundidade nesta comparação. É importante ressaltar que esta caracterização não implica em qual definição o algoritmo implementa.

#### Busca em Largura

Métodos de busca em largura são bem conhecidos na literatura de computação, sendo os fundamentos de diversos procedimentos, como os algoritmos de busca em grafos de Dijkstra para construção de SPF's e de Prim para construção de MSF's [26]. A principal característica deste tipo de algoritmo é dada por sua natureza de expansão, sempre da última borda e uniformemente em sua largura. A respeito do custo do caminho a partir da semente original, todos os vértices de custo  $k$  são visitados antes de visitar qualquer vértice de custo  $k + 1$  [26].

No campo dos algoritmos de transformada *watershed*, pode ser vista uma similaridade entre a busca em largura e a simulação do *watershed* por inundação, onde um conjunto de sementes é expandido para se encontrar a partição ótima formada por estas. Neste sentido, diversos algoritmos também podem ser reconhecidos como buscas em largura, conforme foi caracterizado em suas respectivas seções, considerando-se como sementes os mínimos regionais. Estas propostas se diferenciam em uma série de características e implementam definições distintas, mas preservam a varredura em largura, onde a distância  $k$  passa a ser o custo do caminho, dependente da definição adotada.

Desta forma, pode-se generalizar os algoritmos que aplicam esta estratégia em 3 passos, onde permite-se variar a forma de expansão e rotulação de acordo com a definição adotada, com a variação fundamental sendo as conexões iterativas no passo 2:

1. Defina as sementes/marcadores
  - a. Por entrada; ou
  - b. Por cálculo dos mínimos regionais
2. Calcule as conexões dos *pixels* da iteração atual com a anterior
3. Rotule os *pixels* de acordo com as conexões calculadas. Vá para o passo 2 e expanda as regiões da iteração atual, até visitar todos os *pixels*

De maneira geral, algoritmos de transformada *watershed* com varredura em largura tem como desvantagem a detecção inicial dos mínimos regionais, seja por um procedimento independente, ou por ordenação dos *pixels* por seus valores, para detecção dos componentes conexos mínimos. Este pré-processamento pode se tornar uma operação custosa, e em alguns casos ter seu desempenho reduzido comparado ao equivalente baseado em busca em profundidade [24].

### Busca em Profundidade

Assim como os algoritmos de busca em largura, a busca em profundidade também é muito comum na literatura, especialmente em grafos, representada comumente por algoritmos gulosos. A estratégia adotada aqui é continuar a busca sempre a partir do vértice visitado mais recentemente, enquanto possível, e então retornar para analisar vértices pendentes. De fato, em contraste com a busca em largura, a busca em profundidade prioriza vértices de custo  $k + 1$  assim que são descobertos, antes de visitar todos de custo  $k$  [26].

Pode-se facilmente ver a semelhança entre este procedimento descrito e uma gota d'água descendo sobre uma superfície, onde esta segue um caminho único. Em relação a transformada *watershed*, segue-se o mesmo raciocínio, representado pela técnica de *arrowing*, onde identifica-se para cada *pixel* o vizinho com menor valor. Diversos algoritmos recentes utilizam esta abordagem, seguindo um caminho até que este seja esgotado em um mínimo regional, e assim rotulando-o. Outras propostas são baseadas em construir os caminhos em etapas anteriores, identificar os mínimos, e por último executar a rotulação percorrendo o caminho, caracterizando-os como buscas em profundidade. Assim como gotas d'água que terminam em uma mesma bacia de captação, os caminhos identificados que levam a um mesmo mínimo regional são rotulados igualmente.

Assim como o algoritmo de transformada *watershed* por busca em largura, pode-se generalizar a busca em profundidade em 3 passos, conforme a caracterização mencionada anteriormente:

1. Conecte cada *pixel* ao(s) seu(s) vizinho(s) de acordo com o custo desejado
2. Rotule os mínimos regionais
3. Percorra para cada *pixel* o caminho até o mínimo regional e utilize o rótulo deste

Diversas variações existem para cada passo, vistas nos algoritmos apresentados no Cap. 3. Estes passos são geralmente distinguíveis nos algoritmos, no entanto podem ser unidos, na tentativa de atingir melhor desempenho. Em especial deve-se comentar sobre o passo 1, que depende da composição da imagem na resolução das zonas planas. Estas zonas não podem, de maneira geral, ser conectadas utilizando um algoritmo de busca em profundidade, optando-se por utilizar variações de algoritmos em largura para propagação e divisão correta, aplicando-se uma segunda componente de custo para resolução destes empates. Entretanto, mesmo nestes casos, o passo 3 é característico de algoritmos de busca em profundidade, percorrendo o caminho até encontrar um *pixel* já rotulado ou um mínimo regional.

### Considerações

A classificação dos algoritmos de transformada *watershed* em largura ou profundidade busca auxiliar na compreensão destes, de modo a identificar dificuldades inerentes ao método adotado, ou

a determinadas técnicas. Deve-se ressaltar que esta classificação não tem relação com a definição do algoritmo, sendo possível implementar qualquer definição existente em ambos os paradigmas, com diversos exemplos já existentes e vistos no Cap. 3. No entanto, nota-se uma preferência na literatura por projetar algoritmos onde não é necessário identificar marcadores iniciais, como no caso da busca em largura. Pode-se dizer que os algoritmos com marcadores são preferidos em aplicações onde o usuário os seleciona ou são pré-calculados de alguma forma, sendo que nestes casos, os algoritmos existentes são baseados em propostas clássicas, como os algoritmos de Dijkstra ou Prim. Entretanto, nota-se que para o caso da transformada *watershed* por marcadores, não há técnicas existentes que utilizem varreduras em profundidade, apenas algoritmos em largura, que expandem os marcadores iterativamente, por vezes baseados em algoritmos de grafos, como árvores geradoras mínimas [45].

### 4.2.2 Endereçamento de Caminhos

A maioria dos algoritmos apresentados no Cap. 3, constrói, de alguma forma, caminhos entre os mínimos regionais e todos os *pixels* da imagem. Em alguns destes, os caminhos não são armazenados, por não serem necessários na etapa de rotulação, como no caso do algoritmo de imersão e filas de prioridade ou servem apenas como referência, como no algoritmo IFT. Um caso especial é o algoritmo *watershed cut*, que, apesar de construir um caminho a partir de cada *pixel*, o armazena em um conjunto, não impondo nenhuma relação de conexidade entre os elementos. Todavia, a maioria dos algoritmos faz uso intenso de duas técnicas de endereçamento para construção e armazenamento dos caminhos: via endereço relativo ou endereço absoluto. Ambas as técnicas são consideradas *arrowing*, pois indicam através de um número o próximo *pixel* no caminho. Apesar da equivalência conceitual, a escolha de uma ou outra abordagem afeta outras decisões no projeto de um algoritmo de *watershed*.

O endereço relativo é uma abordagem onde a gama de valores possíveis para cada *pixel* é limitada ao tamanho da vizinhança definida. Assim, para cada *pixel*, utiliza-se um número para indicar qual a direção para onde o caminho segue, sendo que cada vizinho corresponde a uma direção. Esta técnica não permite indicar caminhos para *pixels* fora da vizinhança, por exemplo quando é necessário fazer a compressão de caminhos. Porém a limitação da amplitude de valores possíveis habilita o uso de técnicas como no algoritmo código de corrente de Sun, Yang e Ren [23], onde os códigos de entrada são armazenados em apenas um *byte*, onde cada *bit* corresponde a uma direção possível de entrada.

O endereço absoluto não possui a limitação de gama de valores possíveis ao tamanho da vizinhança, pois neste caso utiliza-se diretamente a posição do *pixel* na matriz, ou seja, seu índice no vetor. De modo a facilitar o armazenamento destes valores, implementações em software geralmente utilizam um endereço unidimensional para as matrizes de imagens. Com esta técnica não há necessidade de avaliar o referencial e qual o deslocamento que deve ser feito, utilizando diretamente o endereço armazenado para seguir o caminho. Desta forma, os *pixels* não precisam ser diretamente conectados, permitindo o uso de técnicas como a compressão de caminhos, além do atravessamento mais eficiente dos caminhos por não haver necessidade de avaliação de vizinhança para identificação do próximo ponto.

### 4.2.3 Rotulação de Caminhos

Como mencionando na seção anterior, diversos algoritmos constróem caminhos que posteriormente são percorridos para determinação dos rótulos dos *pixels* que pertencem a estes. No entanto,

esta operação pode ser realizada de duas formas diferentes, que são atreladas à forma de endereçamento escolhida.

A primeira técnica, mais simples, consiste em percorrer o caminho a partir do *pixel* em avaliação até o ponto em que um rótulo é encontrado, seja em um mínimo regional ou na fusão com outro caminho, e então, com este rótulo em memória, percorrer novamente o caminho aplicando o rótulo encontrado. Este processo também pode ser aplicado no sentido inverso, a partir dos mínimos regionais propagar o rótulo para os *pixels* cujos caminhos terminam nestes. Em ambos os sentidos pode-se utilizar uma estrutura de dados como uma pilha ou fila para armazenar os *pixels* pertencentes ao caminho. Esta técnica não modifica os endereços no caminho, e portanto pode ser utilizada tanto com posições absolutas quanto relativas, sendo mais comum em conjunto com a última.

A compressão de caminhos, opção à rotulação direta, é advinda do *Union-Find*, estrutura de dados introduzida por Tarjan para gerenciamento de conjuntos [27]. Todavia esta operação requer que cada elemento possa ser endereçado para qualquer outro elemento em posição arbitrária, ou seja, o endereçamento absoluto é um requisito. A técnica consiste em percorrer o caminho até encontrar o seu final, considerado como o elemento representativo daquele conjunto, e então modificar os elementos percorridos para apontar diretamente para o representante. Desta forma, não é necessário percorrer novamente o caminho inteiro até o mínimo regional quando um *pixel* que já foi atravessado for avaliado novamente para determinação de seu rótulo. De fato, a eficiência desta operação é mais visível quando são utilizados caminhos múltiplos, já que para caminhos únicos o número de visitas realizada para cada *pixel* pode ser até maior do que utilizando a primeira técnica apresentada. No entanto, o requisito de se utilizar o endereço absoluto resulta em mais agilidade no atravessamento do caminho.

Em paralelo à estas duas técnicas, pode-se aplicar na implementação destes algoritmos uma tabela de equivalência de rótulos. Ao utilizar esta tabela, os caminhos são percorridos apenas uma vez, onde para cada caminho novo - encontrado a partir de um *pixel* que não foi rotulado - atribui-se um novo rótulo. Ao atingir um outro rótulo - seja de outro caminho ou de um mínimo regional - adiciona-se uma entrada na tabela de equivalência. Após percorrer todos os caminhos, a imagem é novamente varrida, desta vez corrigindo os rótulos pré-determinados pelos seus equivalentes definitivos. A eficiência desta alternativa está estritamente ligada ao desempenho da tabela de equivalência e da varredura da imagem na memória. Apesar de ser uma possibilidade, nenhum dos algoritmos analisados utiliza esta técnica.

#### 4.2.4 Descoberta e Rotulação de Mínimos Regionais

Em relação aos mínimos regionais, os algoritmos dividem-se em detectar por ordenamento dos *pixels*, na construção de caminhos de máxima inclinação, ou aplicar um algoritmo específico para isto. O último caso foge ao escopo deste trabalho, dada a variedade de algoritmos possíveis para rotulação de componentes conexos e detecção de mínimos regionais. No primeiro caso, a ordenação dos *pixels* é utilizada nas simulações de imersão, onde componentes conexos novos recebem seus rótulos definitivos assim que são descobertos, e estes são propagados a partir das bordas.

Todavia, ao construir caminhos entre os *pixels* e mínimos regionais, surgem, de forma geral, duas possibilidades, comumente associadas à forma de endereçamento e de rotulação dos caminhos utilizada. Em ambos os casos, parte-se do princípio de que sabe-se quais são os mínimos regionais, onde estes foram descobertos por serem componentes conexos sem *pixels* vizinhos de menor valor, ou seja, nenhum caminho é construído partindo destes. Assim, procede-se então atribuindo um rótulo

novo e propagando-o aos *pixels* vizinhos no componente conexo, ou selecionando-se um *pixel* como representante do mínimo regional, para onde os caminhos dos outros *pixels* passam a ser direcionados. Ao percorrer o caminho e encontrar este representante, seu rótulo será propagado ao caminho, que inclui os elementos restantes do mínimo.

De forma geral, pode-se dizer que a opção por rotular o mínimo regional em uma primeira etapa implica em mais eficiência computacional, pois o atravessamento dos caminhos até um rótulo não incluirá *pixels* já identificados como partes de mínimos regionais. Entretanto, dependendo do projeto do algoritmo, sendo baseado estritamente em grafos por exemplo, esta opção não é possível na etapa de construção destes, sendo necessário utilizar o representante como indicador do rótulo dos demais.

#### 4.2.5 Estruturas de Dados

Uma parte importante das implementações de algoritmos de transformada *watershed* reside nas estruturas de dados utilizadas para controle de visitação dos *pixels*, rotulação e ordenação. Pode-se identificar quatro tipos de estruturas utilizadas: pilha (LIFO), fila (FIFO), fila de prioridade com desempate FIFO e conjuntos/listas. Foge do escopo deste trabalho um aprofundamento nas características de cada estrutura, no entanto, deve-se considerar que o desempenho das suas operações de inserção e remoção é determinante para o desempenho geral do algoritmo que as utilizar.

A estrutura mais comum entre os algoritmos é a fila, utilizada majoritariamente para realizar a propagação das bordas para os *pixels* internos das zonas planas. Seu uso neste caso está associado à necessidade de propagação uniforme das bordas, garantido pela política FIFO, uma vez que inseridos todos os *pixels* de borda conhecidos, estes serão processados antes de qualquer *pixel* interno, inserido por algum *pixel* removido da fila. Intrinsecamente, o uso da fila é equivalente ao cálculo do custo lexicográfico.

O uso da fila de prioridade está ligado às técnicas de simulação de imersão, onde avalia-se os *pixels* por ordem de nível de cinza, propagando seu rótulo para a vizinhança. A política FIFO garante que no desempate ocorra o mesmo comportamento da fila comentado acima, onde ocorre o cálculo intrínseco do custo lexicográfico, e a divisão das zonas planas de acordo com este. Uma forma simples de se visualizar a fila de prioridades FIFO é dada por Beucher e Meyer [11], onde se tem uma estrutura com  $N$  filas independentes, onde  $N$  é o número de níveis de cinza da imagem a ser analisada, com a inserção realizada na fila correspondente ao nível do *pixel*, e a remoção sempre realizada a partir do menor nível disponível. Outras implementações baseiam-se no conceito de árvores para organizar os elementos e otimizar seu desempenho. Estas, no entanto, necessitam de elementos adicionais na comparação para calcular o desempate com política FIFO.

O uso da estrutura de pilha é restrito a apenas um algoritmo, no entanto poderia ser expandido, substituindo o uso de filas onde estas servem apenas como estruturas de armazenamento. No entanto, no algoritmo de código de corrente, o uso da pilha caracteriza sua rotulação como uma busca em profundidade. Ao substituir, neste caso, a pilha por uma fila, o efeito do algoritmo seria de uma rotulação em largura, descaracterizando-o.

Assim como a pilha, o uso de conjuntos simples é restrito apenas ao algoritmo *watershed cut*, todavia poderia substituir filas no mesmo caso das pilhas citado acima. Conjuntos não implicam em ordem na inserção ou remoção de elementos, mas devem permitir acesso aleatório a seus elementos. No entanto, no algoritmo *watershed cut* esta propriedade - que pode acarretar em perda de desempenho por requerer operações de busca aleatória na memória - não é utilizada, e assim, o conjunto

poderia ser substituído por outra estrutura mais simples, como uma pilha, onde o elemento a ser removido sempre tem sua posição em memória conhecida.

Além das estruturas citadas aqui, o armazenamento de grafos é recorrente nos algoritmos, optando-se geralmente por estruturas dinâmicas para estes, visto que matrizes de adjacência tornam-se inviáveis devido às dimensões. Ocasionalmente também são utilizadas estruturas especiais, como na operação de ordenação por frequência sugerida para o algoritmo de imersão de Vincent e Soille [6].

Uma consideração válida a todas estruturas é referente ao gerenciamento de memória, dado o volume de dados trabalhados. Considerando-se que são geralmente armazenados nas estruturas os endereços dos *pixels*, e que uma imagem relativamente pequena pode fazer dezenas de milhares de operações de inserção e remoção, alocações e liberações de memória em estruturas totalmente dinâmicas tornam-se muito custosas, dada a natureza destas operações. Entretanto, por não se conhecer previamente o tamanho máximo que as estruturas atingirão, não se pode utilizar abordagens completamente estáticas. Desta forma, abordagens híbridas são mais adequadas, por permitirem o crescimento das estruturas e por realizarem operações de alocação e liberação de memória para blocos maiores.

#### 4.2.6 Considerações

Nesta seção, buscou-se realizar uma análise comparativa dos algoritmos apresentados no Cap. 3 tendo em vista seus aspectos técnicos. Assim, vislumbra-se auxiliar no projeto de algoritmos mais eficientes, combinando soluções já consolidadas para problemas comuns entre todos os algoritmos. Pode-se resumir estes problemas e as soluções apontadas na seguinte lista:

1. Exploração da imagem
  1. Largura
  2. Profundidade
  3. Aleatório
2. Endereçamento de caminhos ótimos
  1. Endereço absoluto
  2. Endereço relativo
  3. Conjuntos (implícito)
3. Rotulação de caminhos
  1. Compressão
  2. Atravessamento e rotulação
  3. Tabela de equivalência
4. Localização/Rotulação de mínimos regionais
  1. Uso de algoritmo específico (Parâmetro de entrada)
  2. Ordenação dos *pixels* e detecção de componentes conexos
  3. Detecção de componentes mínimos por *arrowing* da imagem

## 5. Estruturas de dados

1. Fila (FIFO)
2. Pilha (LIFO)
3. Fila de Prioridade com política FIFO de desempate
4. Grafo
5. Ordenação
6. Conjunto

Além destas soluções apontadas, diversas delas implicam no uso de estruturas de dados, desde algumas reconhecidamente simples como pilhas (LIFO), até filas de prioridade com política FIFO. A eficiência de muitos algoritmos está severamente ligada à eficiência das operações nestas estruturas, bem como seu gerenciamento de memória. Uma discussão mais aprofundada destas foge do escopo deste trabalho, sendo encontrada na literatura, por exemplo, na Ref. [26].

Utilizando as informações obtidas a partir das análises realizadas nesta seção, pode-se resumir este conteúdo na Tab. 4.2. Nesta tabela são apresentados, por ordem cronológica, os algoritmos apresentados no Cap. 3, listando a definição implementada, a forma de expansão utilizada na visitação dos *pixels*, a forma como endereços são armazenados e processados na etapa de rotulação. Por último, a informação das estruturas de dados necessárias à implementação destes é listada, por ser extremamente relevante ao desempenho final obtido. As informações das características são referentes aos itens listados acima. Há três anotações nesta tabela que devem ser consideradas: (1) O algoritmo de Imersão de Vincent e Soille é baseado na definição Flooding-WT mas a implementa incorretamente; (2) Consideram-se as versões revistas por Roerdink e Meijster [12]; (3) Consideram-se as versões corrigidas e propostas nos Algs. 13 e 14.

## 4.3 Análise de Desempenho do Tempo de Execução

Nesta seção é apresentado o experimento realizado para avaliação do desempenho relativo ao tempo de execução dos algoritmos apresentados no Cap. 3. A implementação destes foi realizada, para efeito de comparação relativa, na linguagem Python [15], utilizando uma abordagem que reflete a especificação do algoritmo de forma mais próxima possível na implementação. Desta forma, procurou-se evitar a introdução precoce de elementos que pudesse otimizar apenas alguns algoritmos, mantendo assim uma base comum a todos, e avaliando apenas o algoritmo em si. Deve-se ressaltar que o uso de implementações C/C++, possuindo características diferentes de execução, geraria resultados diferentes, com possíveis alterações na classificação por velocidade.

Para esta avaliação, foram utilizadas quatro imagens conhecidas na literatura como: **baboon**, **camera**, **lena** e **peppers**. Estas imagens, obtidas no tamanho 512x512, foram redimensionadas proporcionalmente aos tamanhos com largura 64, 128 e 256. Foi obtido então o gradiente morfológico de cada uma delas e também calculado um filtro de dinâmica (remoção de mínimos por contraste), constituindo assim um conjunto de 32 (4x4x2) imagens. A Tab. 4.3 resume as imagens utilizadas e o número de mínimos regionais para cada tamanho, sendo para cada imagem mostrada na primeira linha os valores correspondentes à original redimensionada e na segunda à imagem redimensionada e filtrada. Em relação aos algoritmos, foram utilizados os seguintes: imersão (Alg. 2), fila de priorida-

des (Alg. 3), componentes conexos (Alg. 7), *union-find* (Alg. 9), IFT (Alg. 10), código de corrente (Alg. 11), zona de empate (Alg. 12), tobogã (Alg. 13) e imersão (Alg. 14) invariantes a ordem, caminhos mínimos (Alg. 15) e *watershed cut* (Alg. 16). De modo a obter resultados equivalentes, foram efetuadas duas comparações, uma utilizando apenas algoritmos que não necessitam de tratamento de zonas planas e outra onde para todos os algoritmos foi realizado este pré-processamento. Assim, no primeiro teste foram excluídos os algoritmos *watershed cut* e *union-find*. Desta forma, o custo associado ao algoritmo de remoção de zonas planas é isolado do teste de desempenho. Dos algoritmos apresentados no Cap. 3, foram excluídos os algoritmos Dijkstra-Moore de caminhos mínimos, *hill climbing* e Berge, pois seu desempenho - implementados conforme especificação - inviabilizaria o experimento. A técnica utilizada para otimizar os dois primeiros algoritmos seria o uso de uma fila de prioridade, tornando-os próximos aos algoritmos de fila de prioridade e IFT em comportamento e desempenho. Quanto ao algoritmo Berge, a melhoria de seu desempenho depende de *hardware* otimizado em relação à leitura de memória utilizada. Cada algoritmo foi executado 10 vezes em sequência - alternando-se entre estes - para cada imagem, em um computador com processador Athlon 64 3000+ (1.8Ghz), com 1Gb de memória RAM. Os tempos foram mensurados em milissegundos, descartando-se a menor e maior medida para cada tamanho e classe (filtrada ou não) de imagem.

Desta forma, obtém-se os tempos médios de execução para as imagens originais e filtradas, nos tamanhos citados acima, sem pré-processamento. A análise dos dados obtidos para cada tipo de imagem revela pouca variação entre o desempenho dos algoritmos nas diferentes imagens de um mesmo tamanho, sendo mais influenciado pelo número total de *pixels*. Ordenando-se os algoritmos por seu desempenho, temos que, na implementação Python, o algoritmo Componentes Conexos é o mais rápido, e o algoritmo Código de Corrente é o mais lento. Esta diferença é inesperada em relação ao projeto do algoritmo, ambos com abordagens muito próximas. No entanto, na avaliação de Osma-Ruiz *et al.* [25], foram também necessárias modificações na implementação, sem as quais o algoritmo tem perdas de até 200% de desempenho. Estas modificações não foram incluídas na implementação testada aqui, seguindo apenas o projeto e recomendações originais.

Calculando-se a média para cada grupo de imagens de mesmo tamanho, pode-se calcular o desempenho relativo entre os algoritmos, verificando o aumento de tempo em relação ao algoritmo mais rápido, para cada tamanho. Estes resultados são apresentados nas tabelas 4.4 e 4.5, onde as 4 primeiras colunas correspondem às médias, e as 4 últimas aos desempenhos relativos. A interpretação do desempenho relativo deve ser realizada conforme exemplo: tomando o algoritmo Imersão para o tamanho 64 na Tab. 4.4, este consome 1.43 vezes o tempo do algoritmo Componentes Conexos para completar seu processo. Pode-se então construir um gráfico sobre as médias dos tempos de execução dos algoritmos, apresentado na Fig. 4.12, utilizando os dados das imagens não filtradas, com a escala no eixo Y correspondendo ao tempo em segundos, e o eixo X o número total de *pixels* na imagem. Pode-se dizer também que os algoritmos são bem comportados a respeito de sua classificação relativa geral, sendo estáveis em relação ao crescimento de *pixels* na imagem, indicando características de complexidade parecidas.

Em relação ainda aos dados das Tabs. 4.4 e 4.5, calcula-se a influência do número de mínimos regionais no desempenho geral do algoritmo, com a razão percentual de tempo entre a execução nas imagens filtradas e as originais apresentada na Tab. 4.6. Analisando-se esta tabela, percebe-se que o algoritmo Componentes Conexos apresenta a maior perda de desempenho, de aproximadamente 20%, e que alguns algoritmos, como Fila de Prioridade, IFT e Código de Corrente praticamente não apresentam variabilidade em relação ao número de mínimos. Para os algoritmos Tobogã e Imersão

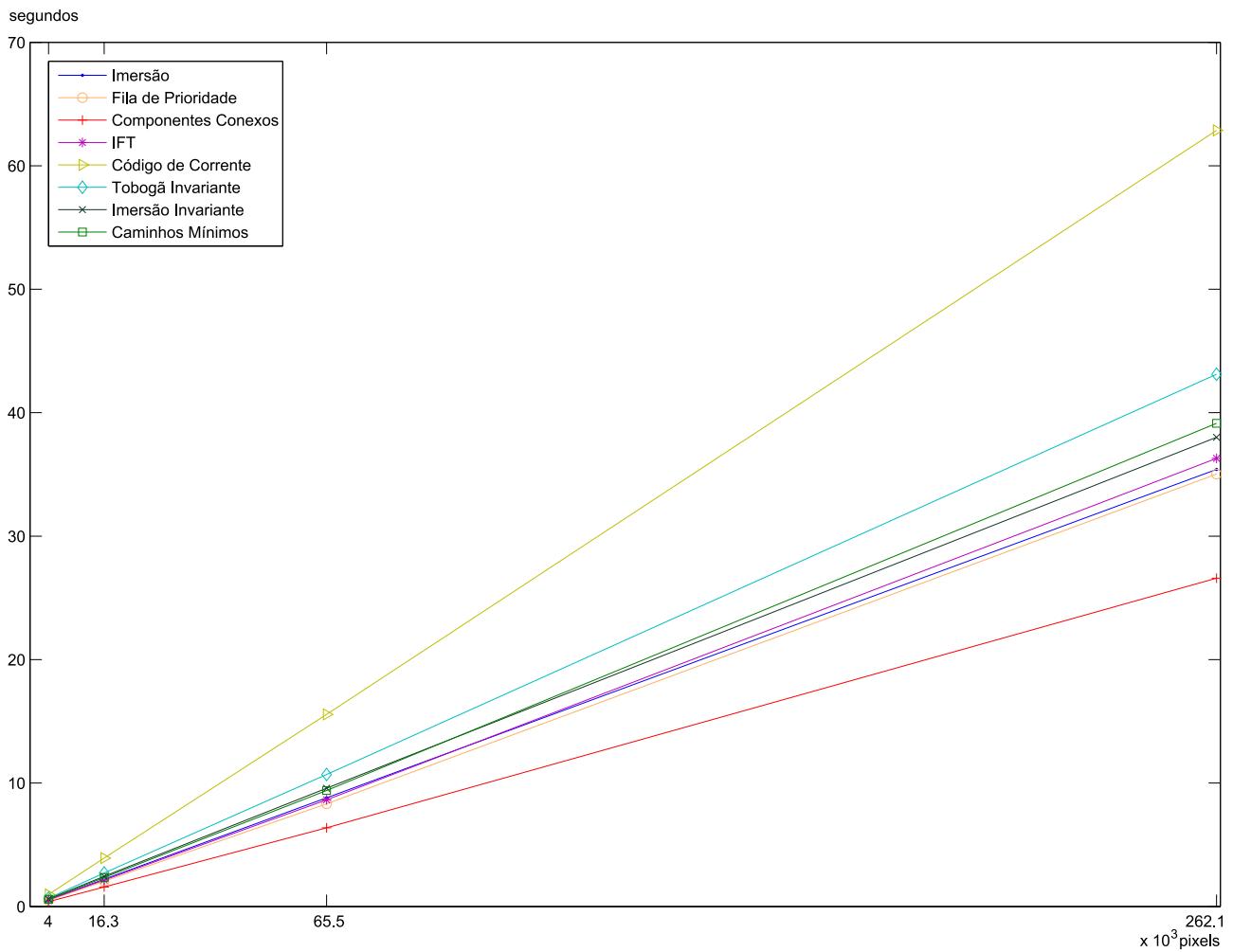


Fig. 4.12: Gráfico comparativo do tempo de execução médio dos algoritmos em segundos por número de *pixels*, para imagens com zonas planas e sem filtragem

Invariante, a redução de mínimos regionais causa a redução no tempo de execução. De forma geral, os resultados dos algoritmos Componentes Conexos, Tobogã e Imersão Invariantes se devem à forma como os *pixels* são rotulados, percorrendo caminhos maiores ou não. Em relação aos algoritmos Fila de Prioridade e IFT, a detecção de mínimos regionais varre a imagem inteira apenas uma vez, independente do número destes detectados, e o algoritmo em si tem seu tempo de desempenho atrelado à fila de prioridade, independente da estrutura da imagem. Da mesma forma, o algoritmo Código de Corrente efetua a rotulação de forma independente da estrutura da imagem, sendo que as outras etapas não são influenciadas pela filtragem realizada. Os dados na Tab. 4.6 devem ser interpretados como uma aceleração no desempenho resultante da redução de mínimos regionais. Valores negativos indicam melhora (tempos de execução menores), enquanto valores positivos indicam piora. A segunda parte da análise de desempenho é realizada tendo como entrada as imagens originais e filtradas pré-processadas para remoção de zonas planas. Desta forma, incluem-se os algoritmos *union-find* e *watershed cut*, que necessitam desta etapa para produzir resultados equivalentes aos outros. No en-

tanto, a remoção de zonas planas também altera o desempenho da maioria dos algoritmos, pois estes não realizam mais o tratamento interno destas, quando explícito. O procedimento utilizado para os testes foi o mesmo do anterior, sendo que o pré-processamento da imagem não foi mensurado, pois este não faz parte do foco deste estudo. Assim como nas imagens sem pré-processamento, os tempos medidos revelam o algoritmo Componentes Conexos como a implementação Python mais eficiente, e o algoritmo Código de Corrente com o pior desempenho, para todos os tamanhos de imagem testados. O pré-processamento para remoção de zonas planas não influenciou na variabilidade nos tempos de execução entre imagens do mesmo tamanho. Desta forma, procede-se com o cálculo das médias por tamanho da imagem, apresentadas nas tabelas 4.7 e 4.8, assim como o desempenho relativo ao algoritmo Componentes Conexos. Realizando o mesmo comparativo do teste anterior, verifica-se na tabela 4.9 a perda de desempenho quando realizada a filtragem na imagem. Analisando-se esta tabela, nota-se que a variação que ocorre em imagens sem pré-processamento, especialmente em relação ao algoritmo Componentes Conexos, é eliminada, tornando o desempenho deste, e também dos outros algoritmos, praticamente estável em relação a estrutura da imagem, dado que o pré-processamento a modifica de forma a otimizar a classe de algoritmos que opera por busca em profundidade. Para concluir este teste, o gráfico apresentado na Fig. 4.13 apresenta o desempenho médio dos algoritmos sobre imagens pré-processadas sem filtragem. Nota-se que não há diferenças grandes de desempenho, no entanto o pré-processamento permitiu ao algoritmo Componentes Conexos aumentar seu desempenho em relação aos outros.

A influência real do pré-processamento sobre o comportamento dos algoritmos é analisada na Tab. 4.10, onde calcula-se a diferença percentual entre o desempenho dos algoritmos sem e com a remoção de zonas planas, para as imagens sem filtragem. Nota-se a pequena influência disto sobre os algoritmos por busca em largura, dado que possíveis reduções são ocasionadas por menos uso de memória por filas, e aumentos no tempo (valores negativos) são resultados de mais níveis de cinza nas filas de prioridade, causando uma tabela de espalhamento maior, utilizada para armazenar cada nível, ocupando mais memória.

Em relação aos algoritmos por busca em profundidade, a diferença de desempenho está relacionada a estratégia utilizada para percorrer os caminhos calculados, se isto é feito recursivamente (Tobogã Invariante), ou através de uma pilha (Código de Corrente) ou fila (Componentes Conexos, Caminhos Mínimos). Claramente, o uso de uma fila traz melhores resultados, visto que os caminhos são percorridos uma vez para identificação do rótulo e utiliza-se a fila então para aplicar estes rótulos nos *pixels* do caminho. Na Tab. 4.10 interpreta-se os valores como o ganho de desempenho relacionado à remoção de zonas planas.

### 4.3.1 Análise de Complexidade

Para possibilitar uma avaliação adequada dos algoritmos e das medidas de desempenho apresentadas nesta seção, deve-se ter como parâmetro também a complexidade à qual as implementações dos algoritmos estão submetidas. No entanto, esta análise não é trivial, pois a maioria dos algoritmos possui diversas etapas com complexidades distintas. Assim, consideram-se como fatores o número total de *pixels*  $n$ , o número de *pixels* em zonas planas  $n_2$  e o número de *pixels* em mínimos regionais  $n_3$ . O tamanho da vizinhança, por ser significativamente menor que o número de *pixels* é descartado da análise de complexidade. A respeito dos algoritmos em que o número de arestas de um grafo é relevante, este é representado por  $e$ .

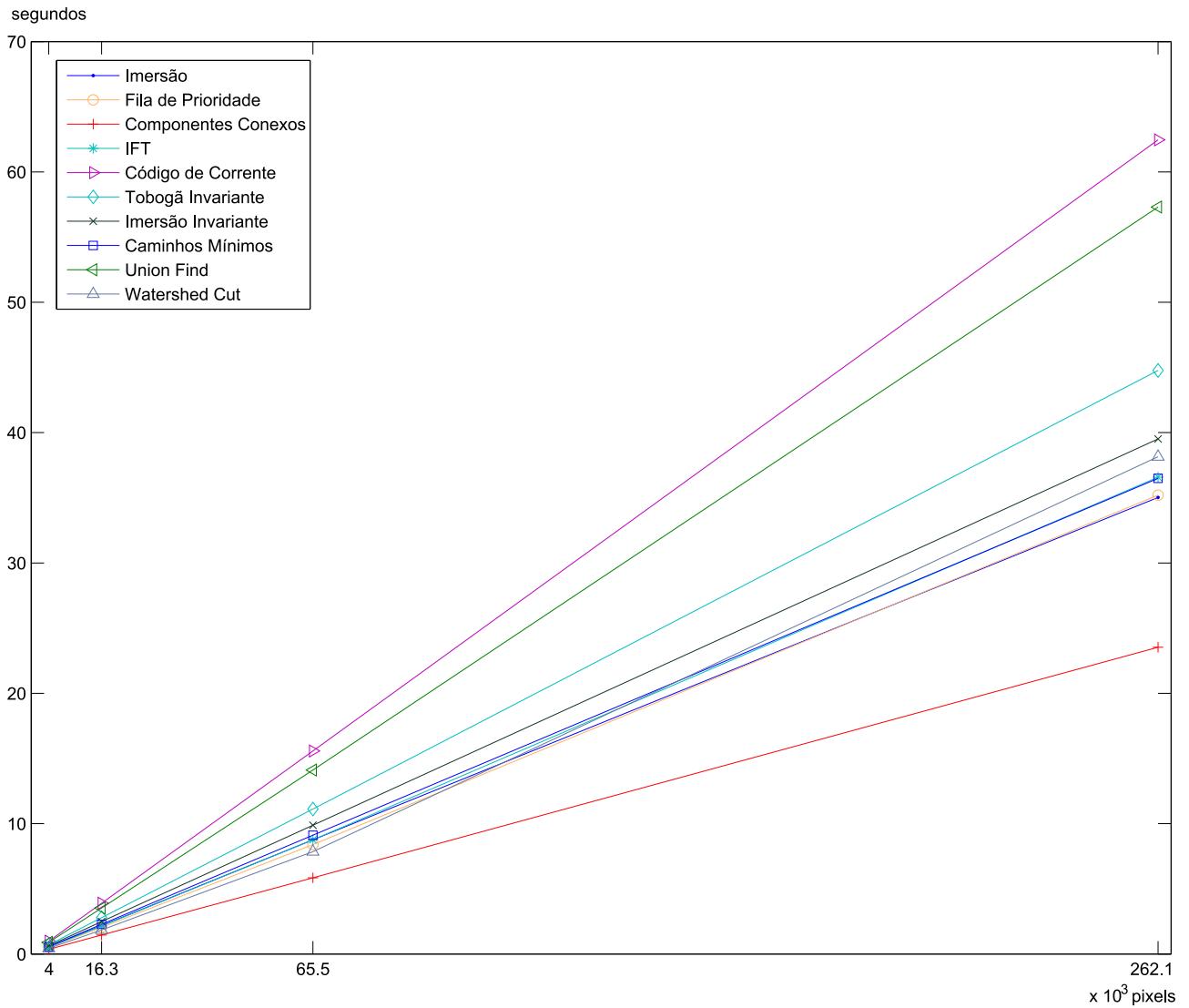


Fig. 4.13: Gráfico comparativo do tempo de execução médio dos algoritmos em segundos por número de *pixels*, para imagens sem zonas planas e sem filtragem

O algoritmo de imersão é considerado linear, utilizando-se uma estrutura de dados especial para ordenação e acesso aos elementos ordenados em cada nível de cinza, são realizadas em média 5 acessos a cada *pixel* [6]. O algoritmo fila de prioridade tem como maior característica o uso da estrutura de fila de prioridade com remoção mínima. Implementada utilizando uma *heap*, esta estrutura obtém complexidade  $O(\log n)$  nas operações de inserção e remoção. Assim, dado que cada *pixel* é inserido e removido uma vez da fila, e é necessária uma varredura adicional para detecção dos mínimos regionais, este algoritmo tem complexidade  $O(2n \log n)$ . O algoritmo componentes conexos é avaliado em cada uma de suas etapas individualmente, com complexidade  $O(4n + n_2 + n_3 \log n_3)$  [29]. A respeito do algoritmo Union-Find, sua complexidade não é avaliada como um todo na literatura, no entanto, este requer duas varreduras para execução do *lower completion*, uma varredura para cons-

trução do grafo acíclico direcionado, e o processamento de uma operação FIND em cada *pixel*, que pode ser desdobrada em múltiplas chamadas, devido a natureza do algoritmo. O algoritmo IFT pode ser visto como uma implementação do algoritmo de Dijkstra com custo máximo, que tem complexidade  $O((n + e) \log n)$  [26]. Entretanto, sua implementação é degenerada no algoritmo de fila de prioridade, onde o número de arestas efetivamente visitadas é igual ao número de *pixels* na imagem. Neste algoritmo também é realizada uma varredura adicional para detecção dos mínimos. A inclusão de condições adicionais verificadas para cada *pixel*, estendendo-o no cálculo da zona de empate não altera sua complexidade. No algoritmo código de corrente, a remoção da operação FIND, faz com que, baseado no algoritmo componentes conexos, sua complexidade seja reduzida para  $O(4n + n^2)$  [23]. O algoritmo caminhos mínimos por sua vez ao reduzir o número total de varreduras da imagem, altera a sua complexidade para  $O(2n + n_2 - n_3)$  [25]. Por último, o algoritmo watershed cut varre a imagem duas vezes apenas, atingindo então complexidade linear [9].

Tomando estas informações como base para análise dos gráficos das Figs. 4.12 e 4.13, verifica-se que as implementações e imagens utilizadas mascararam estes resultados, criando um comportamento quase linear para todos os algoritmos testados. Esta característica pode ser atribuída à linguagem Python, cujo desempenho é limitado pela capacidade de seu intepretador de refletir o real comportamento esperado de algoritmos fortemente iterativos e também aos tamanhos das imagens utilizadas e suas características. Ou seja, devido à escolha da linguagem Python para implementação dos algoritmos em um nível alto de abstração, incorreu-se na perda de desempenho geral, e descaracterização da função limitante destes. Além disso, o *overhead* das estruturas de dados faz com que os algoritmos que menos dependam destas para realizar suas iterações se tornem mais rápidos, como por exemplo o algoritmo componentes conexos se beneficiando desta característica e código de corrente sendo prejudicado.

### 4.3.2 Considerações

Nesta seção foram apresentados resultados de um teste de desempenho do tempo de execução dos algoritmos de transformada *watershed* considerando a execução sobre imagens com e sem pré-processamento para remoção de zonas planas. Deve-se ressaltar que os resultados obtidos aqui são influenciados pela implementação em linguagem Python, que não favorece algoritmos com intensa repetição de dados. Também deve-se considerar a carga causada pelo uso de estruturas de dados, as quais otimizadas podem causar sensíveis diferenças de desempenho. No entanto, o uso desta avaliação pode guiar projetos de algoritmos e implementações a utilizar combinações de técnicas para melhorar o tempo de execução.

Foi mostrado também que nos resultados para a linguagem Python a operação de pré-processamento não traz aumento significativo do desempenho, considerando o seu custo. No entanto, alguns algoritmos dependem desta para produzir resultados consistentes. Entretanto, é claro que os resultados obtidos aqui podem ser diferentes dos obtidos ao utilizar outras linguagens, por exemplo C/C++, devido a natureza das linguagens interpretadas, como Python, de apresentar perda de velocidade em iterações sobre grandes conjuntos de dados, e descaracterizar a função limitante.

## 4.4 Análise de Paralelismo

O cálculo da transformada *watershed* é muito utilizado em tarefas de segmentação de imagens em processos de visão computacional. Tais processos têm - de maneira geral - requisitos de velocidade limitantes, onde deseja-se sempre diminuir os tempos de execução para obter resultados mais rapidamente ou adicionar etapas mais complexas ao processamento. Desta forma, desde a introdução da transformada *watershed* estudam-se técnicas de paralelização de modo a obter ganhos em velocidade. O foco inicial destes trabalhos estava em sistema de processamento de imagem típicos, onde a segmentação representaria uma etapa demorada [46, 30, 44], e algoritmos paralelos poderiam melhorar o desempenho destes sistemas. Neste tópico, Roerdink e Meijster produziram uma extensa revisão bibliográfica [12]. Mais recentemente, devido a evolução dos processadores e a redução significativa do tempo de processamento dos algoritmos sequenciais, o foco dos algoritmos paralelos - apesar de ainda ser a aceleração - passou a ser em sistemas com requisitos de processamento mais restritos, tais como de navegação autônoma, onde deseja-se atingir taxas de processamento de até 30 quadros por segundo. Neste sentido, algoritmos sequenciais ainda não são suficientemente rápidos, e outras estratégias são procuradas [47, 48].

Os primeiros trabalhos de Moga *et al.* [44] e Meijster e Roerdink [46] buscam definir o problema e abordagens para este, baseando-se em algoritmos já existentes. Desta forma, cria-se de fato uma arquitetura e técnicas para gerenciar rótulos e divisões de blocos, necessárias ao paralelismo. Estas abordagens dependem de um algoritmo sequencial, executado em cada bloco, e posteriormente uma etapa de fusão das regiões e unificação de rótulos. Bieniek *et al.* [30] segue esta mesma linha, porém introduz um novo algoritmo sequencial, mais adequado à divisão da imagem em blocos. Uma característica comum às arquiteturas paralelas propostas é a necessidade de detecção e rotulação dos mínimos regionais previamente ao *watershed* em si, feito isto para unificação dos rótulos.

Galilée *et al.* [48] introduz um algoritmo especificamente para arquiteturas paralelas, sem a necessidade da detecção de mínimos regionais, e utilizando comunicação por mensagens para otimizar o processamento entre os vários núcleos. Dessa forma, o algoritmo paralelo de fato realiza o gerenciamento de mensagens e estados, propagando rótulos e distâncias, especialmente em zonas planas, que aguardam a chegada dos dados de vizinhos até a rotulação de toda a imagem. A abordagem de Trieu e Maruyama [47] baseia-se no algoritmo Código de Corrente, de Sun, Yang e Ren [23], modificando-o para arquiteturas FPGA, sem o uso de estruturas de dados como filas e pilhas, realizando a sincronização dos dados até estabilização destes, varrendo a imagem em sentido *raster* e *anti-raster* até a convergência do processo.

Em conjunto com o trabalho desta dissertação, propôs-se um algoritmo de transformada *watershed* apropriado ao uso em processadores gráficos *many-core* de propósito genérico, chamados de GPUs [49]. Estes processadores operam em arquitetura SIMD (*Single Instruction Multiple Data*), requerendo algoritmos especializados e projetados para tal fim. Entretanto, a aplicação da transformada *watershed* inteiramente paralela, como projetado inicialmente, não se mostrou a melhor opção comparando-se os tempos de execução para os passos individuais necessários. Desta forma, aproveitando-se de uma característica do modelo CUDA de programação para GPUs que permite alternar o processamento entre GPU e CPU, desenvolveu-se um algoritmo híbrido, utilizando as etapas mais rápidas em cada caso [50].

A divergência de abordagens paralelas para a transformada *watershed* ressalta que este não é um problema trivial. A evolução de novas máquinas paralelas e arquiteturas diferenciadas torna esta ta-

refa mais complicada, requerendo novos algoritmos. Entretanto, verifica-se que entre as abordagens existentes para paralelização da transformada *watershed*, todas são baseadas em algoritmos sequenciais, carregando problemas de sincronização destes. Assim, identifica-se a necessidade de novas abordagens para algoritmos paralelos de transformada *watershed*, utilizando conceitos e técnicas diferentes das utilizadas em algoritmos sequenciais.

Ano	Algoritmo	Definição	Expansão	Endereçamento	Rotulação	Mínimos	Estruturas de Dados
1991	Inmersão Vincent e Soille	Flooding-WT (1)	Largura	--	--	Ordenação	Fila / Ordenação
1993	Fila de Prioridade Beucher e Meyer	IFT-WT	Largura	--	--	Entrada	Fila Hierárquica
1994	Dijkstra-Moore de Caminhos Mínimos de Meyer(2)	TD-WT	Largura	--	--	Entrada	--
1994	Hill-Climbing de Meyer (2)	TD-WT	Largura	--	--	Entrada	--
1994	Berge de Caminhos Mínimos de Meyer (2)	TD-WT	Aleatório	--	--	Entrada	--
1998	Componentes Conexos de Bieniek e Moga	LC-WT	Profundidade	Absoluto	Compressão	Arrowing	Fila
1998	Union-Find de Meijster e Roerdink	TD-WT	Profundidade	Absoluto	Compressão	Arrowing	Fila / Grafo
2000	IFT de Lotufo e Falcão	IFT-WT	Largura	--	--	Entrada	Fila Hierárquica
2005	Código de Corrente de Sun, Yang e Ren	LC-WT	Profundidade	Relativo	Rotulação	Arrowing	Fila / Pilha
2005	Zona de Empate de Audigier, Lotufo e Couprie	TZ-IFT-WT	Largura	--	--	Entrada	Fila Hierárquica
2005	Tobogã Invariante a Ordem de Lin et al. (3)	TD-WT	Profundidade	Absoluto	Rotulação	Arrowing	Fila / Grafo
2005	Inmersão Invariante a Ordem de Lin et al. (3)	TD-WT	Largura	--	--	Ordenação	Fila / Ordenação
2006	Caminhos Mínimos de Osma-Ruiz et al.	LC-WT	Profundidade	Relativo	Rotulação	Arrowing	Fila
2008	Watershed-Cut de Cousty et al.	WC-WT	Profundidade	Conjunto	Rotulação	Arrowing	Conjunto / Grafo

Tab. 4.2: Resumo das características dos algoritmos estudados

	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
lena	432	1855	7844	36628
lena filtrado	273	1051	3654	16923
baboon	600	2654	10798	37372
baboon filtrado	488	2133	8978	30674
camera	454	1914	8594	24709
camera filtrado	208	845	3640	6804
peppers	442	1661	7337	33000
peppers filtrado	320	1039	3257	22222

Tab. 4.3: Mínimos regionais por imagens e por tamanhos

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	0.55	2.21	8.80	35.41	1.43	1.40	1.38	1.33
Fila de Prioridade	0.50	2.05	8.32	35.03	1.30	1.29	1.30	1.32
Componentes Conexos	0.39	1.58	6.37	26.60				
IFT	0.54	2.14	8.64	36.30	1.39	1.35	1.36	1.36
Código de Corrente	0.98	3.92	15.56	62.87	2.55	2.48	2.44	2.36
Tobogã Invariante	0.68	2.71	10.69	43.11	1.77	1.72	1.68	1.62
Imersão Invariante	0.62	2.44	9.58	38.01	1.60	1.54	1.50	1.43
Caminhos Mínimos	0.59	2.35	9.40	39.14	1.52	1.48	1.48	1.47

Tab. 4.4: Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens com zonas planas não filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	0.56	2.29	9.17	36.85	1.25	1.20	1.16	1.15
Fila de Prioridade	0.51	2.05	8.37	34.55	1.13	1.07	1.06	1.08
Componentes Conexos	0.45	1.91	7.91	31.96				
IFT	0.53	2.15	8.71	35.91	1.18	1.12	1.10	1.12
Código de Corrente	0.98	3.94	15.63	62.89	2.18	2.06	1.98	1.97
Tobogã Invariante	0.66	2.60	10.24	41.98	1.46	1.36	1.29	1.31
Imersão Invariante	0.59	2.30	8.95	35.60	1.30	1.20	1.13	1.11
Caminhos Mínimos	0.62	2.53	10.40	42.43	1.37	1.32	1.32	1.33

Tab. 4.5: Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens com zonas planas filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	2%	4%	4%	4%
Fila de Prioridade	1%	0%	1%	-1%
Componentes Conexos	17%	21%	24%	20%
IFT	-1%	1%	1%	-1%
Código de Corrente	0%	0%	0%	0%
Tobogã Invariante	-4%	-4%	-4%	-3%
Imersão Invariante	-5%	-6%	-7%	-6%
Caminhos Mínimos	5%	8%	11%	8%

Tab. 4.6: Perda de desempenho percentual para os algoritmos, por tamanho de imagem, entre as imagens com zonas planas filtradas e não filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	0.55	2.18	8.76	35.03	1.51	1.50	1.50	1.49
Fila de Prioridade	0.50	2.05	8.40	35.21	1.38	1.41	1.43	1.50
Componentes Conexos	0.36	1.46	5.85	23.54				
Union-Find	0.89	3.53	14.11	57.30	2.44	2.43	2.41	2.43
IFT	0.53	2.14	8.76	36.58	1.44	1.47	1.50	1.55
Código de Corrente	0.98	3.90	15.58	62.46	2.68	2.68	2.66	2.65
Tobogã Invariante	0.70	2.79	11.12	44.77	1.91	1.92	1.90	1.90
Imersão Invariante	0.63	2.49	9.88	39.52	1.71	1.71	1.69	1.68
Caminhos Mínimos	0.56	2.28	9.12	36.49	1.54	1.57	1.56	1.55
Watershed Cut	0.44	1.85	7.85	38.15	1.21	1.27	1.34	1.62

Tab. 4.7: Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens sem zonas planas não filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	0.55	2.20	8.80	35.35	1.51	1.51	1.53	1.54
Fila de Prioridade	0.50	2.03	8.37	34.28	1.37	1.40	1.46	1.49
Componentes Conexos	0.36	1.45	5.74	22.99				
Union-Find	0.89	3.56	14.25	58.04	2.43	2.45	2.48	2.52
IFT	0.53	2.13	8.75	35.75	1.44	1.47	1.52	1.56
Código de Corrente	0.97	3.92	15.67	62.44	2.67	2.70	2.73	2.72
Tobogã Invariante	0.70	2.84	11.50	46.48	1.92	1.96	2.00	2.02
Imersão Invariante	0.63	2.51	10.05	40.00	1.72	1.73	1.75	1.74
Caminhos Mínimos	0.57	2.28	9.15	36.58	1.56	1.57	1.59	1.59
Watershed Cut	0.45	1.84	7.77	40.84	1.22	1.27	1.35	1.78

Tab. 4.8: Média geral dos tempos de execução em segundos por tamanho de imagem e algoritmo, e desempenho relativo, para imagens sem zonas planas e filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	0%	1%	0%	1%
Fila de Prioridade	-1%	-1%	0%	-3%
Componentes Conexos	0%	0%	-2%	-2%
Union-Find	0%	1%	1%	1%
IFT	0%	0%	0%	-2%
Código de Corrente	0%	0%	1%	0%
Tobogã Invariante	1%	2%	3%	4%
Imersão Invariante	0%	1%	2%	1%
Caminhos Mínimos	1%	0%	0%	0%
Watershed Cut	1%	0%	-1%	7%

Tab. 4.9: Perda de desempenho percentual para os algoritmos, por tamanho de imagem, entre as imagens sem zonas planas filtradas e não filtradas

<b>Algoritmos/Tamanhos</b>	<b>64</b>	<b>128</b>	<b>256</b>	<b>512</b>
Imersão	1%	1%	1%	1%
Fila de Prioridade	0%	0%	-1%	-1%
Componentes Conexos	6%	9%	9%	13%
IFT	2%	0%	-1%	-1%
Código de Corrente	1%	0%	0%	1%
Tobogã Invariante	-2%	-3%	-4%	-4%
Imersão Invariante	-1%	-2%	-3%	-4%
Caminhos Mínimos	4%	3%	3%	7%

Tab. 4.10: Comparação de desempenho para os algoritmos por tamanho de imagem, entre as imagens sem filtragem com e sem zonas planas



# Capítulo 5

## Considerações Finais

### 5.1 Conclusões

A transformada *watershed*, desde sua introdução no processamento morfológico de imagens até o presente momento, tem sido alvo constante de estudos, buscando atingir algoritmos mais rápidos e formalizações que os representem consistentemente, além de fornecer uma ferramenta de segmentação útil e robusta para aplicações diversas. No entanto, apesar de prover uma interface simples, a transformada *watershed* pode ser definida por ao menos seis formas, e implementada por 14 algoritmos. Tamanha diversidade inspirou este trabalho, buscando investigar as razões para tal e a evolução existente no período. Para isto foram necessárias ferramentas básicas de notação de algoritmos e da compreensão das definições existentes, seu comportamento, resultados e relações, baseado no trabalho anterior de Romaric focado nas relações entre as diferentes definições [13].

Este trabalho buscou criar um panorama geral da transformada *watershed*, avaliando 14 algoritmos existentes na literatura: Imersão [6], Fila de Prioridade [11], Dijkstra-Moore de Caminhos Mínimos, Hill-Climbing e Berge de Caminhos Mínimos [7], Componentes Conexos [18], Union-Find [22], IFT [19], Código de Corrente [23], Zona de Empate [20], Tobogã e Imersão Invariante a Ordem [24], Caminhos Mínimos [25] e Watershed Cut [9]. Tal estudo permitiu validar os resultados em relação às definições existentes, corrigindo-os quando necessário, provendo pseudocódigo com notação uniformizada, facilitando a comparação, compreensão e implementação destes. Cada algoritmo foi avaliado buscando:

- Definição base;
- Comportamento de varredura na imagem;
- Características de projeto;
- Descrição do funcionamento;
- Uso de estruturas de dados; e
- Detalhes de implementação.

Assim se obteve uma coleção de algoritmos e informações padronizadas, fornecendo meios para classificar e generalizar as abordagens analisadas. A comparação entre os algoritmos se seguiu dividida em análise de resultados, comparativa, de desempenho e de paralelismo. A análise de resultados avaliou os métodos utilizados para tratamento de zonas planas e que influenciam na transformada e a influência das seis definições em aplicações práticas onde o *watershed* é parte do processo. A análise comparativa agrupou, classificou e comparou os algoritmos através das características de projeto e comportamento obtidos na avaliação individual. A análise de desempenho utilizou implementações prototipadas para obter medidas de tempo de execução dos algoritmos em linguagem Python, ideal para representação de pseudocódigo por permitir abstrações de programação e assim se aproximar do primeiro, sem perdas de funcionalidade. Por último, a análise de paralelismo buscou identificar os algoritmos com capacidades para tal e revisar também os trabalhos existentes na literatura, destacando as dificuldades encontradas neste tipo de abordagem.

Com este estudo, se obteve uma coleção de algoritmos compondo uma revisão bibliográfica atualizada na área, compreendendo 14 algoritmos, utilizando pseudocódigo com notação uniforme e provendo implementações operacionais na mesma linha, aproximando as abordagens para representação de algoritmos. Entre os algoritmos analisados, foram efetuadas correções em 3 destes: Tobogã e Imersão Invariantes a Ordem de Lin *et al.* [24] e Caminhos Mínimos de Osma-Ruiz *et al.* [25]. Nos dois primeiros casos, a correção teve o intuito de tornar os algoritmos aderentes à definição TD-WT, com poucas modificações para isto. No terceiro caso, a correção implica apenas em corretude de especificação para eliminação de comportamentos incorretos. A identificação de propriedades dos algoritmos permitiu classificá-los de acordo com o comportamento de sua varredura e compará-los com abordagens clássicas em computação, a saber: busca em largura e busca em profundidade. Esta classificação também permitiu generalizar os algoritmos e descrever - utilizando mais a noção de custos de caminhos e conectividade em detrimento das noções intuitivas - a transformada *watershed*. Entre as outras propriedades de interesse, pode-se, por inspeção dos algoritmos em alguns casos, identificar qual a definição implementada por cada um destes, representando claramente as opções e requerimentos de cada um.

Ao se vislumbrar a coleção obtida, algumas conclusões são imediatas: tem-se maior compreensão da literatura do assunto ao agrupar as abordagens e identificar características nestas, auxiliando na escolha de um algoritmo para um sistema ou mesmo para projetos de novos algoritmos e implementações; a diversidade de algoritmos permite supor que não há limitação na arquitetura destes em relação a qual definição implementam, com exemplos amparando tal suposição, como por exemplo abordagens em largura e profundidade para a definição TD-WT. Desta última conclusão, pode-se constatar também a independência dos algoritmos em relação às definições, podendo-se supor que qualquer abordagem pode implementar qualquer definição, assumindo-se assim que pode existir, por exemplo um algoritmo em profundidade que implementa a definição IFT-WT.

Conforme mencionado na introdução, este trabalho está disponibilizado no ambiente Adessowiki, provendo código-fonte e pseudocódigo de referência para a implementação dos algoritmos estudados aqui. Deseja-se que este torne-se uma fonte de consulta dinâmica, provendo programas eficientes para cada um dos algoritmos, conjugando código-fonte em diversas linguagens e estilos com a especificação dos algoritmos e sua documentação. De forma a avançar neste sentido, o trabalho é colocado no ambiente de forma a permitir a colaboração de outros autores e assim ser constantemente atualizado em relação à transformada *watershed*.

## 5.2 Trabalhos Futuros

A continuação deste trabalho é prevista em quatro linhas, interligadas porém distintas. Planeja-se estudar mais profundamente as abordagens de paralelismo, especialmente mas não limitado, em algoritmos morfológicos, como a transformada *watershed*, tomando proveito das tecnologias recentes que permitem se obter novamente ganhos de desempenho significativos, como os processadores gráficos *many-core* (GPU) e as CPUs *multi-core*, e com isso desenvolver novas estratégias, tanto para projeto, quanto para análise, teste e implementação de tais algoritmos.

Entre os pontos pouco focados neste trabalho, um deles é a análise de desempenho dos algoritmos, onde foi utilizada a linguagem Python por motivos estéticos mas que impactou em uma análise não muito aprofundada. Neste sentido, planeja-se produzir implementações eficientes em linguagem C/C++ dos algoritmos mais promissores em termos de velocidade entre os estudados e analisar novamente o desempenho destes, provendo à comunidade científica amparo na escolha também neste ponto, além de implementações públicas com código-fonte. Uma consequência quase direta de tal análise também é aprofundar os impactos dos diferentes algoritmos e definições nas aplicações práticas, permitindo realizar comparativos com aplicações de grande porte e com quantidades maiores de imagens, inviáveis utilizando as implementações Python.

Em paralelo, neste trabalho apenas os algoritmos de transformada *watershed* clássicos foram estudados, sendo possível também estender a análise para outros tipos, como o *watershed* hierárquico, estocástico, entre outros. No entanto, tais classes são representadas por poucos algoritmos, não apresentando os mesmos desafios encontrados na abordagem original.



# Referências Bibliográficas

- [1] Rafael C. González and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2 edition, 2002.
- [2] J. Clerk Maxwell. On hills and dales. *Philosophical Magazine*, II:233–240, December 1870.
- [3] H. Digabel and C. Lantuéjoul. Iterative algorithms. In J.-L. Chermant, editor, *Proc. Second European Symp. Quantitative Analysis of Microstructures in Material Science, Biology and Medicine*, pages 85–99, Stuttgart, Germany, 1978. Riederer Verlag.
- [4] S. Beucher and C. Lantuéjoul. Use of watersheds in contour detection. In *International Workshop on Image Processing: Real-time Edge and Motion Detection/Estimation*, Rennes, France, September 1979.
- [5] F. Meyer and S. Beucher. Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1):21–46, September 1990.
- [6] L. Vincent and P. Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.
- [7] F. Meyer. Topographic distance and watershed lines. *Signal Processing*, 38(1):113–125, 1994.
- [8] A. X. Falcão, J. Stolfi, and R. A. Lotufo. The image foresting transform: theory, algorithms, and applications. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(1):19–29, 2004.
- [9] J. Cousty, G. Bertrand, L. Najman, and M. Couprise. Watershed cuts: Minimum spanning forests and the drop of water principle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(8):1362–1374, 2009.
- [10] D. Hagyard, M. Razaz, and P. Atkin. Analysis of watershed algorithms for greyscale images. In *Proceedings of International Conference on Image Processing (1996)*, volume 3, pages 41–44, 1996.
- [11] S. Beucher and F. Meyer. *Mathematical morphology in image processing*, chapter The Morphological Approach to Segmentation: The Watershed Transformation. Optical Engineering. M. Dekker, New York, 1993.

- [12] J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms and parallelization strategies. *Fundam. Inf.*, 41(1-2):187–228, 2000.
- [13] R. Audigier. *Zona de empate: o elo entre definições da transformada de watershed e entre elas e a segmentação via conexidade nebulosa*. PhD thesis, FEEC/Unicamp, Campinas, SP, July 2007.
- [14] Romaric Audigier and Roberto de A. Lotufo. Duality between the watershed by image foresting transform and the fuzzy connectedness segmentation approaches. In *Computer Graphics and Image Processing, Brazilian Symposium on*, pages 53–60, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
- [15] Guido van Rossum and Fred Drake L. *Python Reference Manual*. Python Software Foundation, 2009. <http://docs.python.org/ref/ref.html>.
- [16] Roberto A. Lotufo, Rubens C. Machado, André Körbes, and Rafael G. Ramos. Adessowiki on-line collaborative scientific programming platform. In *WikiSym '09: Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, pages 1–6, New York, NY, USA, 2009. ACM.
- [17] SDC Information Systems. *SDC Morphology Toolbox for Python Documentation*. SDC Information Systems, Naperville, IL, USA, 2009. <http://www.mmorph.com/pymorphpro/morph/index.html>.
- [18] A. Bieniek and A. Moga. A connected component approach to the watershed segmentation. In *ISMM '98: Proceedings of the fourth international symposium on Mathematical morphology and its applications to image and signal processing*, pages 215–222, Norwell, MA, USA, 1998. Kluwer Academic Publishers.
- [19] R. Lotufo and A. Falcão. The ordered queue and the optimality of the watershed approaches. In *Proceedings of the 5th International Symposium on Mathematical Morphology and its Applications to Image and Signal Processing*, volume 18, pages 341–350. Kluwer Academic Publishers, June 2000.
- [20] R. Audigier, R. Lotufo, and M. Couprise. The tie-zone watershed: Definition, algorithm and applications. In *Proceedings of IEEE International Conference on Image Processing (ICIP'05)*, volume 2, pages 654–657, 2005.
- [21] R. Audigier and R. A. Lotufo. Watershed by image foresting transform, tie-zone, and theoretical relationships with other watershed definitions. In *ISMM'2007 Proceedings*, volume 1, São José dos Campos, October 2007. Universidade de São Paulo (USP), Instituto Nacional de Pesquisas Espaciais (INPE).
- [22] A. Meijster and J. B. T. M. Roerdink. A disjoint set algorithm for the watershed transform. In *Proc. IX European Signal Processing Conf EUSIPCO '98*, pages 1665–1668, 1998.
- [23] Han Sun, Jingyu Yang, and Mingwu Ren. A fast watershed algorithm based on chain code and its application in image segmentation. *Pattern Recognition Letters*, 26(9):1266–1274, 2005.

- [24] Y. Lin, Y. Tsai, Y. Hung, and Z. Shih. Comparison between immersion-based and toboggan-based watershed image segmentation. *IEEE Transactions on Image Processing*, 15(3):632–640, 2006.
- [25] V. Osma-Ruiz, J. I. Godino-Llorente, N. Sáenz-Lechón, and P. Gómez-Vilda. An improved watershed algorithm based on efficient computation of shortest paths. *Pattern Recognition*, 40(3):1078–1090, 2007.
- [26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge , Massachusetts, 2 edition, 2001.
- [27] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.
- [28] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1 edition, 1999.
- [29] A. Bieniek and A. Moga. An efficient watershed algorithm based on connected components. *Pattern Recognition*, 33(6):907–916, 2000.
- [30] A. Bieniek, H. Burkhardt, H. Marschner, M. Nölle, and G. Schreiber. A parallel watershed algorithm. In *Proceedings of 10th Scandinavian Conference on Image Analysis (SCIA97)*, pages 237–244, 1997.
- [31] Romaric Audigier and Roberto Lotufo. Uniquely-determined thinning of the tie-zone watershed based on label frequency. *J. Math. Imaging Vis.*, 27(2):157–173, 2007.
- [32] Gilles Bertrand. On topological watersheds. *J. Math. Imaging Vis.*, 22(2-3):217–230, 2005.
- [33] Jean Cousty, Gilles Bertrand, Laurent Najman, and Michel Couprise. Watershed cuts: Thinnings, shortest-path forests and topological watersheds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009, to appear.
- [34] W.S. Rasband. Imagej. Technical report, U. S. National Institutes of Health, Bethesda, Maryland, USA, 1997–2009. <http://rsb.info.nih.gov/ij/>.
- [35] F. Meyer. Color image segmentation. In *Image Processing and its Applications*, 1992, pages 303–306. IEEE Computer Society, 1992.
- [36] Open computer vision library, 2009. <http://sourceforge.net/projects/opencvlibrary/>.
- [37] Roland Levillain, Thierry Géraud, and Laurent Najman. Milena: Write generic morphological algorithms once, run on many kinds of images. In *Proceedings of the 9th International Symposium on Mathematical Morphology (ISMM)*, Groningen, The Netherlands, Aug. 2009. <http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/>.
- [38] Epita Research and Development Laboratory. Olena project, 2009. <http://olena.lrde.epita.fr>.

- [39] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
- [40] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.
- [41] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications*. 1976.
- [42] Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, 1969.
- [43] C. Berge. *The Theory of graphs and its applications*. John Wiley & Sons, Inc., New York, 1 edition, 1962.
- [44] Alina Moga, Timo Viero, Bogdan Dobrin, and Moncef Gabbouj. Implementation of a distributed watershed algorithm. In J. Serra and P. Soille, editors, *Mathematical morphology and its applications to image processing*, volume 2, pages 281–288. Kluwer Academic Publishers, 1994.
- [45] F. Meyer. Minimum spanning forests for morphological segmentation. In J. Serra and P. Soille, editors, *Mathematical morphology and its applications to image processing*, volume 2, pages 77–87. Kluwer Academic Publishers, 1994.
- [46] A. Meijster and J. B. T. M. Roerdink. *A Proposal for the Implementation of a Parallel Watershed Algorithm - CAIP'95*, volume 970 of *Lecture Notes in Computer Science*, pages 790–795. Springer Berlin / Heidelberg, 1995.
- [47] Dang Ba Khac Trieu and Tsutomu Maruyama. Real-time image segmentation based on a parallel and pipelined watershed algorithm. *Journal of Real-Time Image Processing*, 2(4):319–329, December 2007.
- [48] Bruno Galilée, Franck Mamalet, Marc Renaudin, and Pierre-Yves Coulon. Parallel asynchronous watershed algorithm-architecture. *IEEE Transactions on Parallel and Distributed Systems*, 18(1):44–56, 2007.
- [49] André Körbes, Giovani Bernardes Vitor, Janito Vaqueiro Ferreira, and Roberto de Alencar Lotufo. A proposal for a parallel watershed transform algorithm for real-time segmentation. In *Proceedings of Workshop de Visão Computacional WVC'2009*, São Paulo, Brazil, Sep. 2009. Available on [http://iris.sel.eesc.usp.br/wvc2009/WVC2009\\_CD.rar](http://iris.sel.eesc.usp.br/wvc2009/WVC2009_CD.rar).
- [50] Giovani Bernardes Vitor, Janito Vaqueiro Ferreira, and André Körbes. Fast image segmentation by watershed transform on graphical hardware. In *Proceedings of the 30ºCILAMCE*, Armação dos Búzios, Brazil, Nov. 2009.

## Apêndice A

# Framework de Processamento de Imagens

Em algoritmos de processamento de imagens, a busca por eficiência geralmente leva a programas desenvolvidos utilizando o nível de abstração mais baixo possível, de modo a se obter instruções mais eficientes e programas mais otimizados. No entanto, uma consequência direta desta abordagem é a obtenção de códigos-fonte de difícil compreensão e manutenção. Entretanto neste trabalho buscou-se prezar pela legibilidade de código e ao mesmo tempo se obter implementações funcionais próximas das especificações dos algoritmos em pseudocódigo. Assim, optou-se por desenvolver um *framework* de base, contemplando os problemas mais comuns e abstraindo-os do código que efetivamente produz os resultados dos algoritmos estudados.

Considerou-se para efeito de projeto deste *framework* que algumas funcionalidades deveriam ser encapsuladas, entre elas o tratamento de dimensionalidade para processamento de imagens de qualquer formato e tamanho; o tratamento da borda, considerando a necessidade de uso de vizinhanças, onde *pixels* externos à imagem original não devem ser visitados, sendo comportamento padrão na morfologia matemática; e o tratamento de vizinhança, encapsulando o cálculo de deslocamentos no domínio da imagem. Algumas restrições adotadas para este desenvolvimento devem ser comentadas: (1) exige-se que uma relação de vizinhança seja simétrica, de forma que o cálculo da borda da imagem também o seja; (2) as relações de vizinhança são dadas como uma lista de deslocamentos em cada dimensão da imagem; (3) as bordas têm valor constante determinado pelo infinito positivo da linguagem Python. Assim, tem-se o primeiro bloco de código criando as constantes de vizinhanças 4 e 8, e definindo a constante de borda e uma função que verifica o valor de um *pixel* e identifica se este é uma borda ou não. Esta função é necessária apenas por questões de compatibilidade e legibilidade.

```
# constants
# neighbourhood
N4 = array ([[ -1 ,0],[0 ,1],[1 ,0],[0 , -1]])
N8 = array ([[ -1 ,0],[0 ,1],[1 ,0],[0 , -1],[-1 , -1],[-1 ,1],[1 ,1],[1 , -1]])

# values
inf = 1e400
BORDER = inf

def isBorder(v):
```

```
""" Tests if the value is a border value """
return v == BORDER
```

Uma restrição importante diz respeito à forma de varredura da imagem, de modo a converter esta em um vetor unidimensional. Este processo deve ser padronizado, ordenado, variando-se da última dimensão para a primeira, ou seja, em uma imagem 2D, isso implica que para cada linha serão varridas suas colunas. A Fig. A.1 exemplifica este processo, realizando a varredura de uma imagem com 3 linhas e duas colunas, e transformando-a em um vetor de 6 posições. Padronizando a forma de varredura, pode-se então prever as posições em que os *pixels* estarão a partir do tamanho da imagem original.

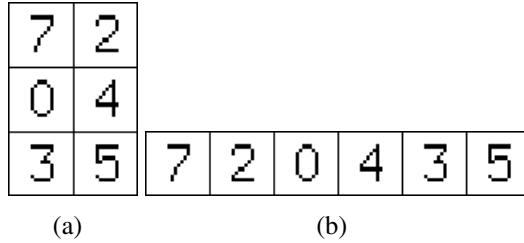


Fig. A.1: Exemplo de transformação de (a) imagem em (b) vetor unidimensional

Utilizando esta transformação de uma imagem com dimensões arbitrárias para apenas uma dimensão, pode-se abstrair o problema da dimensionalidade. No entanto, o tratamento da imagem em relação às bordas deve ser cuidadoso, de forma a criar estas áreas e impedir que o processamento escape do domínio da imagem. Em resumo, tais cuidados são tomados por uma operação de redimensionamento da imagem (*padding*) onde cria-se a borda com o valor constante definido anteriormente, pela operação de transformação em vetor unidimensional, e pela operação de recorte da imagem (*crop*) onde a partir do vetor de trabalho, restaura-se a imagem para suas dimensões originais, eliminando a borda. Além de funcionar como principal ponto de entrada do *framework*, a classe **wsImage** apresentada abaixo tem nestas operações suas funcionalidades fundamentais.

```
# common classes for the algorithms
class wsImage():
    """ Class for storing the images and controlling entry and exit points """

    def __init__(self, array):
        """ Constructor for storing the N-D input image """
        self.input = array
        self.work = None
        self.label = None
        self.output = None
```

```

def begin(self , offsets):
    """ Prepare the image for processing """
    from numpy import zeros , ravel

    if len(self.input.shape) != offsets.shape[1]:
        raise Exception("Image shape does not fit offsets dimensions")

    # initialise the neighbour
    self.neighbour = wsNeighbour(offsets)
    # pad the input image
    self.work = self.\_pad()
    # store the padded shape
    self.workshape = self.work.shape
    # ravel the padded image (make 1-D)
    self.work = ravel(self.work)
    # make a zeroed copy of it
    self.label = zeros(self.work.shape)
    # initialise the output
    self.output = None
    # initialise the shape of the image
    self.neighbour.setImageShape(self.workshape)
    self.neighbour.setImage(self.work)
    # initialise the domain object
    D = wsDomain(self.work.size)
    D.setImage(self.work)

    # returns the neighbourhood relation , the working image , the label
    # image and the domain of the image
    return self.neighbour.N, self.work, self.label , D

def \_pad(self):
    """ Pads the N-D image with the BORDER constant as necessary for
    containing all the offsets from the list """
    from numpy import zeros
    # generate the newshape by iterating through the original and adding
    # the extra space needed at each dimension
    newshape = tuple(map(lambda orig , d: orig + (d-1) , self.input.shape ,
                        self.neighbour.shape))
    # generate the slicing list
    slicing = map(lambda orig , d: slice((d-1)/2 , (d-1)/2 + orig) ,
                  self.input.shape , self.neighbour.shape)
    # create the padded image
    workimage = zeros(newshape)
    workimage[:] = BORDER
    workimage[slicing] = self.input
    return workimage

def \_crop(self):
    """ Reshape and crop the label image """
    return self.crop(self.label)

```

```

def crop(self, x):
    """ Reshape and crops a N-D image to the original size (same as
    self.input) """
    from numpy import reshape
    # generate the slicing list
    slicing = map(lambda orig, d: slice((d-1)/2, (d-1)/2 + orig),
                  self.input.shape, self.neighbour.shape)
    # reshape the label image to the original shape
    temp = reshape(x, self.workshape)
    # crop the temp image
    return temp[slicing]

def end(self):
    return self.\_crop().astype('int32')

def makeWorkCopy(self, default=0):
    """ Make a copy of the work image filled with the value and type of
    parameter default """
    copied = self.work.copy()
    copied = copied.astype(type(default))
    copied.fill(default)
    return copied

```

Através da classe **wsImage** apresentada, a qual encapsula o problema da dimensionalidade prioritariamente, tem-se acesso ao controle de domínio da imagem, encapsulado pela classe **wsDomain**. Um problema recorrente em algoritmos de processamento de imagens, especialmente em morfologia onde trata-se de conjuntos limitados, é o tratamento de borda. No caso dos algoritmos morfológicos, a vizinhança fora da imagem é recortada, ou seja, o domínio de processamento é limitado apenas ao domínio da imagem, não sendo utilizadas estratégias como valores constantes ou repetição cíclica, tal como no processamento da transformada de Fourier. Desta forma, deseja-se abstrair este cálculo, de forma que na programação de um algoritmo não seja necessário verificar se os endereços sendo visitados extrapolam o domínio da imagem. A programação desta classe foi realizada através de objetos iteráveis, onde a cada nova iteração verifica-se qual o próximo endereço válido a ser visitado, considerando-se um espaço pré-definido de endereços possíveis. Um endereço é dito válido se na imagem tomada para cálculo de domínio este não representar uma borda produzida pela operação de *padding*. A Fig. A.2 apresenta a imagem com *padding* de bordas para uma vizinhança 4 ou 8, e sua vetorização, onde os *pixels* em branco indicam a borda e serão descartados na visitação do domínio.

É importante ressaltar na Fig. A.2 os *pixels* de borda intermediários, os quais são transparentemente descartados pela classe **wsDomain**, não permitindo que o programa os processe e incorra em erros. Esse tratamento interno aumenta significativamente a legibilidade e usabilidade deste *framework*, permitindo que varreduras pela imagem sejam realizadas apenas iterando-se sobre um conjunto de endereços.

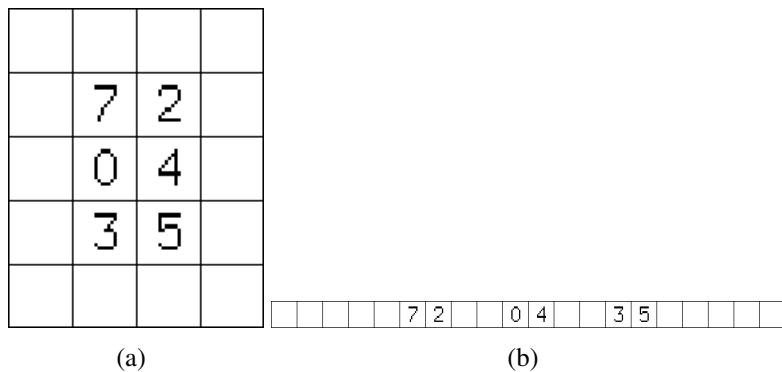


Fig. A.2: Exemplo de tratamento das bordas na transformação de (a) imagem com borda em (b) vetor com borda, onde estes pixels são intercalados

```

class wsDomain:
    def \_\_init\_\_(self , length):
        self.inner = xrange(length)
        self.count = 0

    def next(self):
        if self.count >= len(self.inner):
            self.count = 0
            raise StopIteration

        while isBorder(self.im[ self.inner[ self.count ] ]):
            self.count += 1
            if self.count >= len(self.inner):
                self.count = 0
                raise StopIteration

        c = self.count
        self.count = c + 1
        return self.inner[c]

    def \_\_getitem\_\_(self , item):
        return self.inner[item]

    def \_\_iter\_\_(self):
        return self

    def \_\_len\_\_(self):
        return len(self.inner)

    def setImage(self , im):
        self.im = im

```

A classe **wsDomain** fornece funcionalidade essencial aos algoritmos, permitindo acesso transpa-

rente aos *pixels* que compõe exclusivamente o domínio da imagem, sem tratamentos especiais para bordas. De mesma importância é o acesso à vizinhança, comumente denotado em pseudocódigo como  $N(p)$ . Com o desenvolvimento da classe **wsNeighbour** e em especial o método **N** busca-se obter um código-fonte que remeta diretamente à notação de pseudocódigo. Em particular, esta classe deve lidar com 2 restrições: (1) realizar o cálculo dos deslocamentos fornecidos em qualquer dimensão para o vetor unidimensional; e (2) impedir que a vizinhança avance sobre a borda da imagem. A primeira restrição é resolvida pela programação da Eq. A.1, dada de acordo com a padronização da varredura explicada anteriormente e exemplificada na Fig. A.1. Dado um deslocamento N-dimensional  $\langle d_N, d_{N-1}, \dots, d_2, d_1 \rangle$ , a transformação para um deslocamento unidimensional  $r$  em uma imagem com tamanhos  $\langle s_N, s_{N-1}, \dots, s_2, s_1 \rangle$  é dada por:

$$r = \sum_{i=1}^N (d_i \prod_{j=1}^{i-1} s_j) \quad (\text{A.1})$$

Para uma dada vizinhança, e.g. vizinhança-4, são dadas 4 listas de deslocamento, e calcula-se 4 valores de  $r$ , correspondendo aos deslocamentos que devem ser calculados sobre um ponto qualquer no domínio da imagem. Este cálculo só é realizado uma vez, pois não depende do *pixel*, sendo então apenas somado a coordenada deste para localização dos vizinhos. Para uma imagem de tamanho (3, 10, 15) por exemplo, e um deslocamento (-1, 1, 2), a Eq. A.1 se expande da seguinte forma:

$$r = (2) + (1 \cdot 15) + (-1 \cdot 10 \cdot 15) = -133 \quad (\text{A.2})$$

Ao calcular a posição do vizinho (-1, 1, 2) do *pixel* de endereço unidimensional 260 será apenas somado o deslocamento -133 a este, resultando na posição 127. Este endereço será então verificado se corresponde a uma borda da imagem, e, caso contrário, será incluído na lista de vizinhos possíveis do *pixel* 260. A classe **wsNeighbour** encapsula este comportamento, permitindo ao método **N** operar de forma rápida e transparente em relação a dimensionalidade e bordas.

```
class wsNeighbour():
    """ Class for neighbourhood processing """

    def __init__(self, offsets):
        """ Constructor for the list of offsets in N-D (neighbours)
            offsets must be a m x N matrix, where m is the number of
            offsets (neighbours) and N is the dimensions of the image """
        self.offsets = array(offsets)
        self._shape()
        self.s = None

    def _shape(self):
        """ Calculates the shape of the offsets """
        N = self.offsets.shape[1]
        self.shape = []
```

```

for i in range(N):
    dmax = max(self.offsets[:, i])
    dmin = min(self.offsets[:, i])
    if abs(dmax) != abs(dmin):
        raise Exception("Offsets must be symmetrical")
    d = dmax - dmin + 1
    # make the dimension always odd
    if d % 2 == 0:
        d += 1
    self.shape.append(d)
self.shape = tuple(self.shape)

def setImageShape(self, imshape):
    """ Set the image shape and calculates the offsets in 1-D """
    self.s = imshape
    if len(self.s) != self.offsets.shape[1]:
        raise Exception("Image shape does not fit offsets dimensions")

    # calculate the offsets in 1-D
    # the process occurs like this:
    # each offset is multiplied by the multiplication of the values of
        the next components of the shape of the image and summed:
    # example:
    # shape: (3, 10, 15)
    # offset: [1, 1, 2]
    # offset in 1-D: (1 * 10 * 15) + (1 * 15) + (2)
    #
    # of course, the offsets must follow the order of the shape (Nth-D,
        ..., 3rd-D, 2nd-D, 1st-D), that is usually
    # (time, channel, row, column) or in grayscale images (time, row,
        column) or simple 2-D images (row, column)

    # LONG VERSION
    # self.roffsets = []
    # for offset in self.offsets:
        # roffset = 0
        # for i in range(len(offset)):
            # n = offset[i]
            # roffset += n * reduce(lambda x,y: x*y, self.s[(i+1):], 1)
        # self.roffsets.append(roffset)

    # SHORT VERSION (using map and reduce)
    self.roffsets = map(
        lambda offset: sum(
            map(lambda n, i:
                n * reduce(lambda x,y: x*y, self.s[(i+1):], 1),
                offset, range(len(offset))
            )
        ),
        self.offsets
    )

```

```

def setImage(self , im):
    """ Set the working image to query for border values on
    neighbourhood calculation """
    self.im = im

def N(self , pixel):
    """ Returns the list of indexes of neighbours of pixel in 1-D """
    if not self.s:
        raise Exception("Set the image shape first!")

    # calculate the coordinates of the neighbours based on the offsets
    # in 1-D
    n = map(lambda c: c + pixel , self.roffsets)
    r = list()

    for i in n:
        if isBorder(self.im[i]):
            continue

        r.append(i)

    return r

def query(self , c):
    """ Lookup on the roffsets for the index of the offset c """
    for i in range(len(self.roffsets)):
        if self.roffsets[i] == c:
            return i
    return -1

def addOffset(self , p, index):
    """ Adds the offset of the desired index to the value p """
    if index < 0 or index >= len(self.roffsets):
        return None
    else:
        c = self.roffsets[index]
        return p + c

```

Além das três classes apresentadas aqui, foram desenvolvidas interfaces para estruturas de dados comuns em algoritmos, como filas, pilhas e filas de prioridade. Estas interfaces foram construídas de modo a manter transparente a implementação interna, que, neste caso, foi realizada utilizando listas e dicionários embarcados na linguagem Python.