UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

2009.1

OTIMIZAÇÃO E EFICIÊNCIA DE ALGORITMOS DE ROTULAÇÃO DE COMPONENTES CONEXOS EM IMAGENS BINÁRIAS

TRABALHO DE GRADUAÇÃO

Aluno	Diêgo João Costa Santiago	{djcs@cin.ufpe.br}
Orientador	Tsang Ing Ren	{tir@cin.ufpe.br}

Recife, Junho de 2009

Assinaturas

Este Trabalho de Graduação é resultado dos esforços do aluno Diêgo João Costa Santiago, sob a orientação do professor Tsang Ing Ren, sob o título de "Otimização e Eficiência de Algoritmos de Rotulação de Componentes Conexos em Imagens Binárias". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Tsang Ing Ren
Orientador

_____

Diêgo João Costa Santiago
Aluno

"O mais importante não é saber. É nunca perder a capacidade de aprender."

Leonardo Boff

Agradecimentos

Aos meus pais, Maria do Carmo e Cezar Santiago, por tudo que conquistei.

A meu orientador, Tsang, grande companheiro nesse projeto, pela confiança depositada no meu trabalho, oportunidade e apoio.

Aos meus familiares, amigos e a todos aqueles que me apoiaram, incentivaram ou criticaram meu trabalho.

A Juliane Cristina Botelho de Oliveira Lima, que colaborou com a versão inicial desse projeto na disciplina de Processamento de Imagens.

Resumo

Rotulação de componentes conexos é uma etapa indispensável e de fundamental importância em aplicações na área de Visão Computacional e Reconhecimento de Padrões. Em muitos casos, a etapa de rotulação de componentes conexos em Reconhecimentos de Padrões é uma das que consomem mais tempo em relação às outras etapas. Por isso, rotulação de componentes continua sendo uma ativa área de estudo. Um dos fatores que possui maior impacto no desempenho da rotulação de componentes conexos é a escolha de um bom método para a extração de características da imagem. Este trabalho de graduação tem como objetivo o estudo da eficiência e otimização dos algoritmos de rotulação de componentes conexos, auxiliando na escolha dos métodos para determinados casos.

Palavras-chaves: Componentes Conexos, Visão Computacional, Algoritmos, Otimização.

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

Tecnicamente, imagens são formadas por componentes conexos que, por sua vez, são formados de *pixels* conectados. A conectividade entre *pixels* é um conceito importante usado no estabelecimento de bordas de objetos e componentes de regiões em uma imagem [4]. Rotulação de componentes conexos é o procedimento de atribuição de um único rótulo a cada objeto (ou componente conexo) da imagem [13]. O algoritmo transforma uma imagem binária em uma simbólica que representa os objetos conectados.

Rotulação de componentes conexos é uma etapa indispensável e de fundamental importância em aplicações na área de Visão Computacional e Reconhecimento de Padrões [12]. É muito importante capturar as características essenciais da imagem. Uma forma de fazer isso é extrair significantes regiões da imagem. Quando um objeto é extraído da imagem, é necessário identificá-lo [1]. Os algoritmos de rotulação de componentes conexos são úteis para identificar objetos. Análise e reconhecimento de documentos, em particular, se beneficiam bastante deste método. Ela é bastante utilizada para a extração de objetos de alta importância, como caracteres, imagens meios-tons e regiões de texto [1].

Algoritmos de rotulação de componentes conexos são também muito utilizados na Física Estatística relacionados a teoria da percolação [9]. A concepção de percolação tem sido associado com a permeabilidade de um fluido por um meio poroso [9]. A formação de clusters de moléculas idênticas em cristais pode ser a estrutura para a teoria da percolação [9]. Vários exemplos do fenômeno e suas aplicações são enumerados na literatura, entre ele: magnetização espontânea em diluição com características ferro-magnéticas, difusão de doenças num organismo, formação de polímeros gelatinosos, condutividade elétrica em condutores amorfos e em solução de metal-amônia [9]. Isto demonstra um pequeno exemplo de vários fenômenos de relacionados a percolação presentes na natureza.

Além de detectar a existência de percolação (o mesmo componente conexo do tipo A se estende do lado até o outro na imagem), um algoritmo de rotulação é de interesse para identificar cada um dos clusters em separado, juntamente com outras informações de interesse, tais como o tamanho médio, estrutura espacial ou tamanho máximo do componente conexo [7][8][11].

Vários algoritmos têm sido propostos até agora. Eles podem ser classificados em cinco classes: algoritmos *mult-pass*, algoritmos *two-pass*, algoritmos *one-pass*, algoritmos desenvolvidos para imagens representadas por árvores e algoritmos paralelos [12][13]. Três delas (*multi-pass*, *two-pass* e *one-pass*) são representativas e adequadas para computadores de arquitetura simples [12] e são diferenciadas pela quantidade de vezes que o algoritmo atravessa a imagem.

Em muitos casos, a etapa de rotulação de componentes conexos em Reconhecimentos de Padrões é uma das que consomem mais tempo em relação às

outras etapas. Por isso, rotulação de componentes continua sendo uma ativa área de estudo [13].

Um dos fatores que possui maior impacto no desempenho da rotulação de componentes conexos é a escolha de um bom método para a extração de características da imagem. Este trabalho de graduação tem como objetivo o estudo da eficiência e otimização dos algoritmos de rotulação de componentes conexos, auxiliando na escolha dos métodos para determinados casos. Os algoritmos estudados foram implementados para comparação das suas eficiências.

O presente documento está estruturado em sete capítulos. O capítulo 2 apresenta uma revisão bibliográfica de assuntos relacionados à rotulação de componentes conexos. No capítulo 3, serão descritos os algoritmos *multi-pass*. No capítulo 4, descritos os algoritmos *two-pass*. No capítulo 5, os algoritmos *one-pass*. No capítulo 6, serão mostrados resultados da implementação, comparando os tempos de execução dos diferentes algoritmos implementados. O capítulo 7 é a conclusão do trabalho.

2. Componentes Conexos em Imagens Binárias

Apresenta-se neste capítulo uma revisão bibliográfica de assuntos relacionados à rotulação de componentes conexos. Inicialmente, apresenta-se a representação de imagens. Em seguida, o conceito de componentes conexos e a estrutura básica de um algoritmo de rotulação de componentes conexos.

2.1 Representação de Imagens

Segundo Gonzales e Woods [4], "o termo imagem refere-se à função bidimensional de intensidade da luz f(x,y), onde x e y denotam as coordenadas espaciais e o valor de f em qualquer ponto(x,y) é proporcional ao brilho da imagem naquele ponto".

Existem várias maneiras de representar digitalmente imagens. A mais utilizada é a que representa a imagem na forma de uma matriz, em que em cada posição da mesma representa um ponto da imagem e onde é armazenado o nível de cinza daquele ponto. Para o menor elemento de uma imagem digital, ou seja, para cada posição da matriz é dado o nome de *pixel*. Cada *pixel* é representado por um byte, podendo assim assumir 256 valores diferentes.

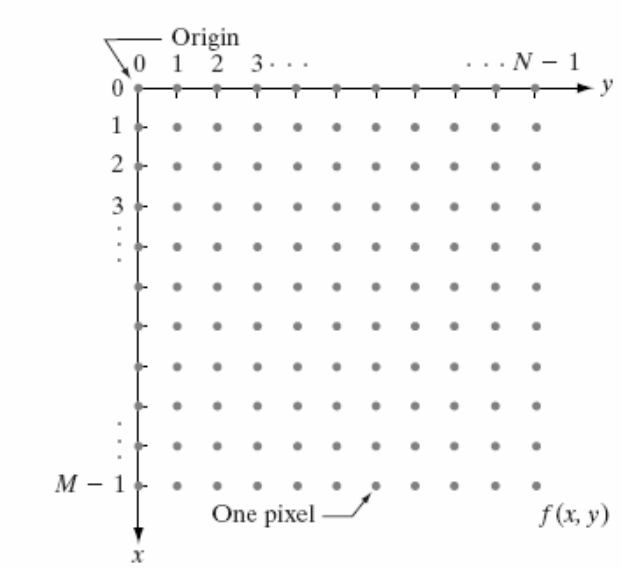

Figura 1. Representação de uma imagem

A figura acima ilustra a representação descrita e que é utilizada neste trabalho. Também é exibida a orientação de sistema de coordenadas mais utilizado

nos algoritmos de processamento de imagem, com a origem do sistema no canto superior esquerdo da matriz [4].

2.2 Componentes Conexos

A conectividade entre *pixels* é um conceito importante usado no estabelecimento de bordas de objetos e componentes de regiões em uma imagem. Para determinar se dois *pixels* estão conectados, é preciso definir a noção de vizinhança e se seus níveis de cinza obedecem algum critério de similaridade. Considerando estar trabalhando com imagens binárias, que contém apenas dois tipos de *pixels* (*object pixel*, valor 1, e *background pixel*, valor 0), dois *pixels* estão conectados se eles possuem pelo menos o mesmo valor, obedecendo assim o critério de similaridade [4].

Existem duas formas comuns de definirmos conectividade para imagens bidimensionais (2D): conectividade-de-4 e conectividade-de-8 [4][13]. Dois *pixels p* e *q* estão conectados de 4 se obedecem ao critério de similaridade e se são vizinhos horizontais ou verticais, e estão conectados de 8 se obedecem ao critério de similaridade e são vizinhos horizontais, verticais ou diagonais. Os vizinhos horizontais de um *pixel* de coordenada (x, y) são os pixels de coordenadas $(x+1, y)$ e $(x-1, y)$, os verticais são de coordenadas $(x, y+1)$ e $(x, y-1)$ e os diagonais são de coordenadas $(x+1, y+1)$, $(x+1, y-1)$, $(x-1, y+1)$ e $(x-1, y-1)$. Nesse projeto, trabalharemos com conectividade-de-8 [4].

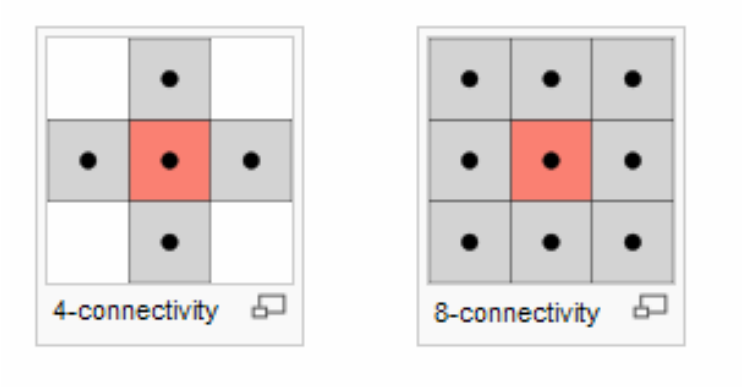

Figura 2. Conectividade entre pixels.

Componente conexo de uma imagem é o conjunto de *pixels* da imagem que estão conectados entre si, ou seja, dado um *pixel p* do componente conexo, existe pelo menos um caminho de *p* a todos os outros *pixels* do componente conexo. Um caminho de um *pixel p* com coordenadas (x, y) a um *pixel* com coordenadas (s, t) é uma seqüência de pixels distintos com coordenadas (x_0, y_0), (x_1, y_1),..., (x_n, y_n) em que $(x_0, y_0) = (x, y)$ e $(x_n, y_n) = (s, t)$ e (x_i, y_i) está conectado a (x_{i-1}, y_{i-1}) [4].

2.3 Algoritmos de Rotulação de Componentes Conexos

Rotulação de componentes conexos é o procedimento de atribuição de um único *label* a cada objeto (ou componente conexo) da imagem [13]. O algoritmo transforma uma imagem binária em uma simbólica que representa os objetos conectados.

Basicamente, os algoritmos examinam a imagem, designando *labels* provisórios para cada *object pixel*. Se essa varredura (*scanning*) for feita da esquerda para a direita e de cima para baixo denomina-se *forward scan*, enquanto se for feito da direita para a esquerda e de baixo para cima, *backward scan* [13].

Cada vez que um *pixel* é examinado, seus vizinhos são examinados para determinar o *label* apropriado. Observando a figura abaixo, se estamos examinando o *pixel e*, os vizinhos a serem examinados são os *pixels a, b, c* e *d*. Esses serão os *pixels* examinados, pois já possuem *pixels* provisórios determinados em etapas anteriores. A partir de agora, usaremos nesse documento as notações *pixels a*, *b*, *c* e *d* para representar os vizinhos e *e* para representar o *pixel* corrente sendo examinado com a distribuição mostrada na figura abaixo [13].

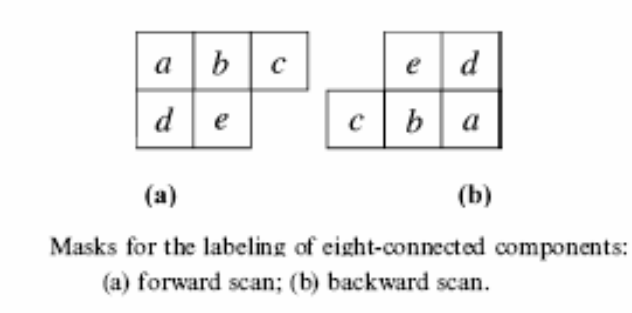

Figura 3. Vizinhos visitados na varredura.

Existem várias abordagens para rotulação de componentes conexos. Considerando que não abordaremos algoritmos paralelos e que as imagens são representadas por matrizes (arrays bidimensionais), podemos classificar esses algoritmos em algoritmos *multi-pass*, *two-pass* e *one-pass* [13]. Esses algoritmos serão descritos nos capítulos a seguir.

3. Algoritmos Multi-pass

Apresenta-se neste capítulo os algoritmos de rotulação *multi-pass*. Os algoritmos *multi-pass* examinam a imagem em *forward scanning* e *backward scanning* alternativamente para propagar *labels* provisórios até que não aconteçam mudanças nos *labels*. Esses algoritmos não obtêm necessariamente como resultados *labels* em ordem seqüencial [12].

Os algoritmos dessa classe são relativamente fáceis de implementar em hardware porque são baseados apenas em operações seqüenciais e não requer um algoritmo para resolver equivalência entre *labels*. Entretanto, ele precisa de um grande número de passos, porém o tempo de execução não é bem definido teoricamente, pois depende da complexidade dos componentes conexos na imagem [12].

3.1 Algoritmo Convencional

O algoritmo básico de varredura (*scanning*) pode ser apresentado como se segue. Para cada *object pixel p*, são examinados os quatros *pixels* dos vizinhos (conforme descrito na secção 2.3). Se nenhum desses vizinhos é um *object pixel*, um novo *label* é designado, caso contrário, o label do *object pixel p* é o menor *label* dos vizinhos. O *label* de um *background pixel* é 0 [13].

No primeiro passo,

$$L[e] \leftarrow \begin{cases} 0, & I[e] = 0 \\ l, (l \leftarrow l + 1), & I[i] = 0, \forall i, \\ \min_{i \mid I(i)=1} (L[i]) & \textit{caso contrário} \end{cases}$$

Onde *i* são os elementos vizinhos (*a*, *b*, *c* e *d*), *e* é o pixel corrente sendo examinado, *L* é o array de *labels* e *I* o array de *pixels*, sendo *I[x,y] =0* para *background pixels* e *I[x,y] =1* para *objects pixels*.

Nos passos seguintes, quando todos os *labels* provisórios já foram descobertos,

$$L[e] \leftarrow \min_{i \mid I(i)=1} (L[i]) \; se \; I[i] = 1 \; e \; I[e] = 1$$

O *forward scanning* e o *backward scanning* se repetem até que não haja mudanças nos *labels*.

3.2 Scan Plus Connection Table

O *Scan plus Connection Table* (SCT4) é um algoritmo *multi-pass* que otimiza o algoritmo básico de *scanning* usando uma *Connection Table*. A *Connection Table* usada no SCT4 é uma estrutura de dados mínima, um *array* unidimensional, que mantém a informação equivalente dos *labels* e produz o correto *label* final [12][13].

No primeiro passo,

$$L[e] \leftarrow \begin{cases} 0, & I[e] = 0 \\ l,(T[l] \leftarrow l, l \leftarrow l+1), & I[i] = 0, \forall i, \\ \min_{i|I(i)=1}(T[L[i]]),(T[L[i]] \leftarrow L[e]) & \textit{caso contrário} \end{cases}$$

Onde T representa o *array* unidimensional do *Connection Table* [12].

Nos passos seguintes,

$$L[e] \leftarrow \min_{i|I(i)=1}(T[L[i]]), T[L[i]] \leftarrow L[e] \textit{ se } I[i] = 1$$

O SCT4 é mais rápido que o algoritmo básico, pois propaga mais rapidamente as informações de equivalência dos *labels*. Chamamos essa básica versão do SCT de SCT4, pois acessa os quatros vizinhos na máscara [13].

He et al. [5] considera esse algoritmo como um híbrido de algoritmos *multi-pass* e *two-pass*, pois utiliza um array unidimensional para armazenar e resolver *labels* equivalentes. Isso é característico de algoritmos *two-pass* e não é usado no *multi-pass* convecional. Porém, consideramos esse algoritmo como um *multi-pass* pelo critério de classificação, que é o número de vezes que atravessa a imagem.

3.3 Árvores de Decisão

Uma árvore de decisão é uma maneira de expressar, em forma de árvore, um conjunto de condições que é necessário ocorrer para que um determinado conjunto de ações deva ser executado. Cada nós internos da árvore representam uma decisão e cada folha representa uma ação a ser executada.

O SCT1 é um algoritmo *multi-pass* que também usa *Connection Table* e otimiza o SCT4 usando árvores de decisão. O SCT1 não necessariamente examina todos os vizinhos. Esse algoritmo utiliza a noção de vizinhança entre os vizinhos para reduzir o número de possibilidades de operações. O *pixel b* é o primeiro *pixel* a ser examinado, pois todos os outros *pixels* da máscara são vizinhos do *pixel b*. Caso o *pixel b* seja um *object pixel*, o *label* de b será copiado para o *label* de *e*. Caso contrário, os outros ramos das árvores devem ser analisados [13].

Existem quatro possíveis árvores de decisão para esse algoritmo. Duas dessas árvores são mostradas na figura 4 (figura 4b e 4c). As outras duas árvores

podem ser obtidas através das árvores da figura trocando a ordem de análise dos *pixels a* e *d*. Nesse projeto, usamos na implementação a árvore da figura 4b [13].

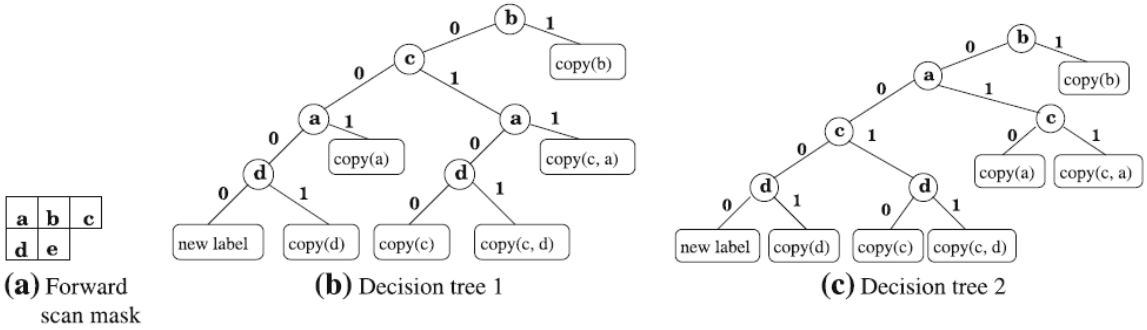

Figura 4. As árvores de decisão usadas na varredura de vizinhos conectados-de-8.

Percebe-se também através da figura 4 que existem três operações definidas pelas árvores de decisão. A primeira, a função *copy* de um argumento contém apenas um comando, por exemplo, *copy(a)* possui o comando *L[e]←T[L[a]]*, ou seja, o *label* do *pixel e* será o label equivalente do *pixel a*.

A segunda, a função *copy* de dois argumentos contém três comandos, por exemplo, *copy(c,a)* possui os comandos *L[e]←min(T[L[c]], T[L[a]]), T[L[c]]←L[e]* e *T[L[a]]←L[e]*. Ou seja, o *label* do *pixel e* será o menor dos *labels* dos *pixels c* e *a*. Observa-se que o *pixel b* só aparece na função de *copy* com um argumento devido ao fato de ser vizinhos dos outros *pixels* e que a ação *copy(a,d)* não existe pois *a* e *d* são vizinhos.

A terceira operação, executada quando todos os *pixels* da máscara são *background pixels*, cria um novo label da mesma forma que os outros algoritmos, *L[e]←l, T[l]←l* e *l←l*+1. Essa operação é executada apenas no primeiro *scan*, quando o *pixel* ainda não está rotulado [13].

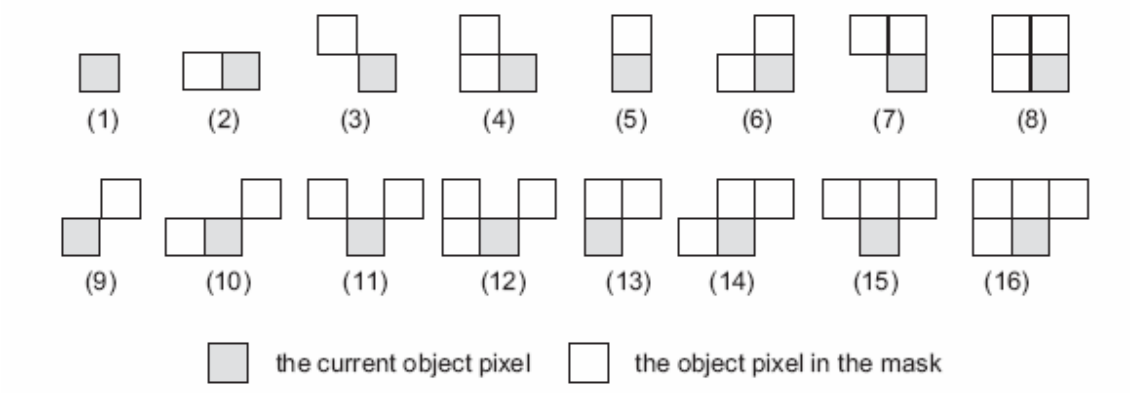

Figura 5. Os seis possíveis casos de vizinhos para o corrente *pixel*

Analisando a árvore da figura 4b, a que usamos na implementação, percebemos que oito casos (casos 5, 6, 7, 8, 13, 14, 15 e 16) executam a ação da folha

copy(b) que realiza apenas uma comparação, dois casos (casos 3 e 4), o *copy(a)* e dois casos (casos 11 e 12), o *copy(c,a)*, que realizam três comparações. As outras folhas correspondem cada uma a um caso e precisa de quatro comparações. Assim sendo, (8*1+4*3+4*4)/16=2,25. A mesma média é obtida analisando as outras árvores.

O SCT1 é mais rápido do que o SCT4, pois realiza menos comparações com os vizinhos. Em média, o SCT1 compara 2,25 vezes, enquanto o SCT4 sempre compara 4 vezes. Logo, o SCT1 é 1,78 (4/2,25) vezes mais rápido que o SCT4.

4. Algoritmos Two-pass

Apresenta-se neste capítulo os algoritmos de rotulação *two-pass*. Os algoritmos *two-pass* examinam a imagem apenas duas vezes. A primeira vez, na fase de varredura (*scanning phase*), *labels* provisórios são designados para os *objects pixels* e a informação de equivalência entre os *labels* provisórios é armazenada em alguma estrutura de dados, geralmente um *array* unidimensional ou bidimensional. Após ou durante essa primeira vez, as equivalências dos *labels* são resolvidas por algum algoritmo durante a fase de análise (*analysis phase*). A segunda vez, na fase de rotulação (*labeling phase*), os *labels* provisórios são substituídos pelos *labels* equivalente usando a informação equivalente [12][13].

O tempo de execução dos algoritmos dessa classe pode também depender da complexidade dos componentes conexos na imagem devido ao uso de algoritmos para determinação da equivalência entre os *labels*, que por sua vez, depende da estrutura de dados que representa a equivalência [12]. Uma das mais eficientes estruturas de dados para representar a informação de equivalência é *union-fnd data structure* (conjuntos disjuntos) que será abordado em breve.

4.1 Conjuntos Disjuntos

Nos algoritmos de dois passos, é necessário armazenar a relação de equivalência em uma estrutura de dados. A estrutura de dados mais adequada para isso é Conjuntos Disjuntos. Durante a *scanning phase*, é construída uma estrutura de Conjuntos Disjuntos para armazenar a informação de equivalência entre os *labels* provisórios. Após a *scanning phase*, essa estrutura é analisada e os *labels* finais determinados para cada *label* provisório [13].

Uma estrutura de dados Conjuntos Disjuntos é uma coleção $S=\{S_1, \ldots, S_k\}$ de conjuntos dinâmicos disjuntos, ou seja, sem interseção entre si. Cada conjunto S_k é identificado por um representante, que é um membro do conjunto. Tipicamente não importa quem é o representante, apenas que ele seja consistente [2]. Nos algoritmos de rotulação de componentes conexos, geralmente esse representante é o menor *label* do conjunto [13].

Existem apenas três operações em estruturas de conjuntos disjuntos: *make-set*, *union* e *find-set*. *Make-set(x)* cria um novo conjunto cujo único elemento é representado por x, onde x não pode pertencer a outro conjunto da coleção. *Union(x, y)* executa a união dos conjuntos que contêm x e y, digamos S_x e S_y, em um conjunto único, onde $S_x \cap S_y = 0$ e o representante de $S = S_x \cup S_y$ é um elemento de S. *Find-set(x)* retorna um ponteiro para o representante (único) do conjunto que contém x [2][13].

4.1.1. Representação de Conjuntos Disjuntos por Listas Encadeadas

Uma maneira simples de implementar uma estrutura de dados Conjuntos Disjuntos consiste em representar cada conjunto como uma lista encadeada. O primeiro elemento da lista é o representante e todos os elementos referenciam o representante, como mostrado na figura abaixo [2].

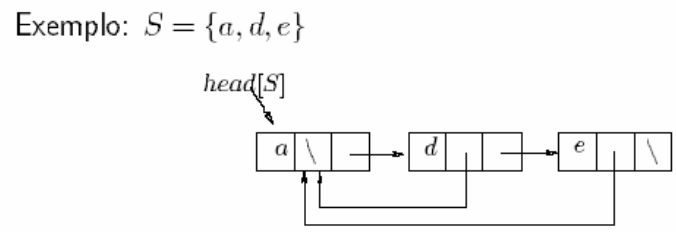

Figura 6. Uma lista encandeada representando um conjunto dos Conjuntos Disjuntos.

Com essa representação de lista encadeada, com cada elemento referenciando o representante, as operações de *make-set* e *find-set* são fáceis, realizando-se em tempo constante. Porém essa maneira não é muito eficiente devido à operação de união entre dois conjuntos. A operação união, *union(x, y)*, adiciona a lista de *x* no fim da lista de *y*. O representante do conjunto é o elemento que era originalmente o representante de *y*. Todavia, temos de atualizar o ponteiro para o representante de todos os elementos de *x* [2]. A figura abaixo mostra a realização de uma união entre dois conjuntos.

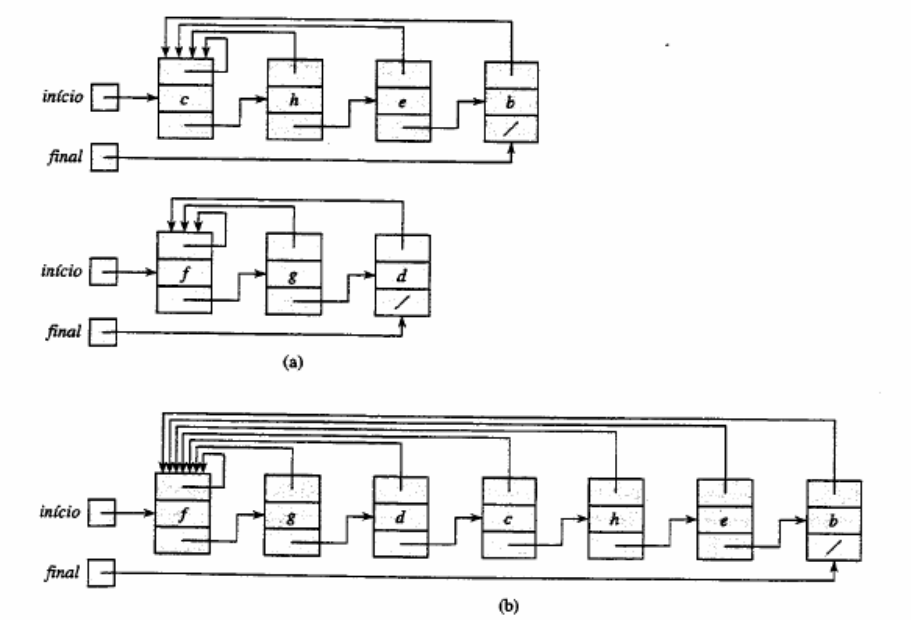

Figura 7. Operação de união entre dois Conjuntos Disjuntos representados por listas encadeadas.

A eficiência da união pode melhorar se usarmos a heurística de união ponderada ou união por tamanho. Nesse caso, adicionamos a lista de menor comprimento no final da lista de comprimento maior. Com isso, precisamos atualizar o representante dos elementos da lista de menor tamanho, ou seja, menos operações [2]. Para isso, é necessário armazenar o comprimento da lista, que no *make-set* será 1 e é a soma dos comprimentos dos dois conjuntos na *union*.

Na maioria dos casos, os nós de listas encadeadas estão espalhados randomicamente pela memória. Uma operação *union* caminha pela lista e atravessa a memória de maneira imprevisível. Isto tipicamente é lento. Sugere-se, então, representar listas encadeadas usando *arrays* que se baseia em localização seqüencial na memória [5][6][13].

Na representação de Conjuntos Disjuntos com listas encadeadas usando *arrays*, cada elemento é indexado no *array*, que armazena o representante do conjunto, conforme é visto na figura abaixo. Ainda são necessários mais dois *arrays*, um para armazenar o próximo elemento a ele na lista e o outro para armazenar o final da lista [5][6]. Um quarto *array* é necessário se for implementado a heurística ponderada para armazenar o comprimento da lista.

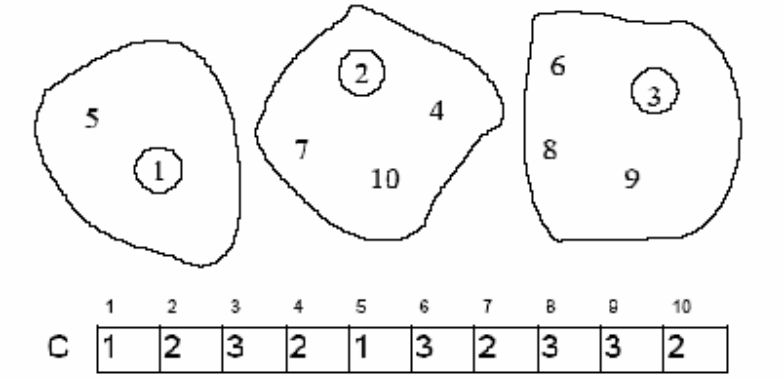

Figura 8. Representação de Conjuntos Disjunto com listas encadeadas através de arrays

4.1.2. Florestas de Conjuntos Disjuntos

Em uma implementação mais rápida da estrutura de dados Conjuntos Disjuntos, representamos conjuntos por meio de árvores denominados Florestas. A raiz da árvore contém o representante do conjunto e o filho referencia para o pai, como mostrado na figura abaixo. Em Florestas de Conjuntos Disjuntos, *make-set* cria uma árvore com um único vértice. *Find-set* segue os ponteiros para os pais até atingir a raiz. *Union* faz a raiz de uma árvore apontar para a raiz da outra. Para isso, o *union* precisa de dois *find-sets*, um para cada conjunto a fim de encontrar o seu representante [13].

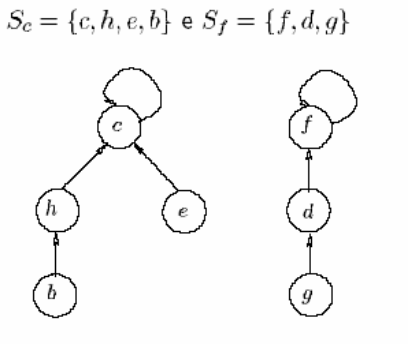

Figura 9. Uma Floresta.

Visto que o custo da operação *union* em Florestas depende da operação *find-set,* refinamentos nas operações *find-sets* são muito importantes na redução do custo do algoritmo. Compressão de caminho (*path compression*) é uma abordagem muito utilizada e eficiente para reduzir o custo de manipulação de Florestas. Usando compressão de caminho, após o *find-set* todos os nós visitados durante o *find-set* passam a referenciar a raiz, seu representante. Nos próximos *find*-sets, precisaríamos de apenas um passo para encontrar o representante. Um exemplo de um *find-set* com compressão de caminho é mostrado na figura abaixo, quando é feito um *find-set* em *p* [3][13].

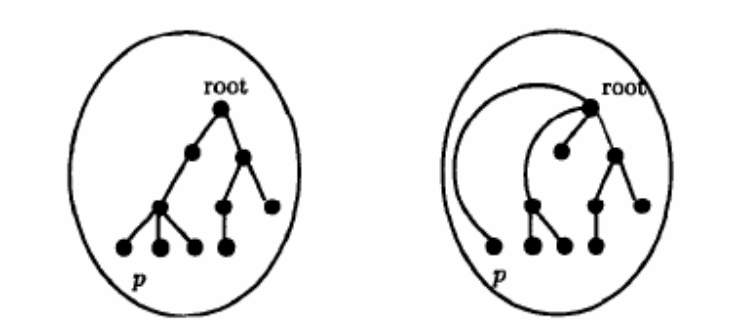

Figura 10. Resultado de um *Find*-set no elemento *p* usando Compressão de Caminho.

Assim como nas listas encadeadas, na maioria dos casos, os nós de árvores enraizadas baseadas em ponteiros estão espalhados randomicamente pela memória. Uma operação *find-set* segue os ponteiros até a raiz (o representante) e atravessa a memória de maneira imprevisível. Isto tipicamente é lento. Sugere-se, então, representar Florestas usando *arrays* que se baseia em localização seqüencial na memória. A figura abaixo mostra um exemplo de uso de *arrays* para representar Florestas. O conteúdo da posição *i* do *array P* representa a posição onde está o pai desse elemento no *array P* [13].

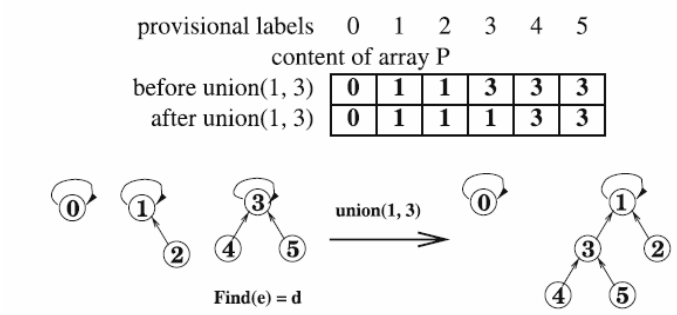

Figura 11. Representação de Florestas através de Conjuntos Disjuntos.

4.2 Scan Plus Union-Find

A forma natural de representar ligações em árvores é através de ponteiros. O *Scan plus Union-Find* (SUF) é um algoritmo de dois passos que usa árvores de decisão no *scanning* e Florestas com árvores enraizadas baseadas em ponteiros para resolver o problema de equivalência de *labels*.

Nesse projeto, apresentamos duas abordagens para o SUF: o SUF0 e o SUF1. No SUF1, após o *scan* de cada linha, ocorre uma operação de *flatten* nessa linha. A operação de *flatten* em um *pixel* faz com que a informação de equivalência desse *pixel* referencie diretamente o representante do seu conjunto. SUF0 não possui essa operação de *flatten* e por isso executa mais rapidamente do que o SUF1 [3][13].

Algumas adaptações precisam ser realizadas nas operações das árvores de decisão para os algoritmos *two-pass*. A operação *copy* de um argumento copia o *label* do vizinho, por exemplo, *copy(a)* é o comando *L[e]←L[a]*. A operação *copy* de dois argumentos faz a união dos conjuntos desses argumentos, por exemplo, *copy(c,a)←union(P,L[c],L[a])* [13].

4.3 Scan Plus Array-Based Union-Find

O *Scan Plus Array-Based Union-Find* (SAUF) é um algoritmo de dois passos que usa árvores de decisão no *scanning* e Florestas com árvores enraizadas baseadas em *arrays* para resolver o problema de equivalência de *labels* [13].

4.4 Run-Based Two-Scan

O *Run-Based Two-Scan* (RTS) é um algoritmo de dois passos que usa árvores de decisão no *scanning* e listas encadeadas baseadas em *arrays* para resolver o problema de equivalência de *labels* [6].

Nesse projeto, apresentamos duas abordagens para o RTS: o RTS0 e o RTS1. O RTS0 é o RTS original, proposto pelos autores, He et al. [5][6]. Nesse algoritmo, a união é realizada adicionando a lista com o representante que possui o maior *label* no final da lista do representante que possui o menor *label*. No RTS1, realiza-se a heurística de união ponderada, ou seja, adiciona-se a lista com o menor comprimento no final da lista de maior comprimento. O RTS0 comporta-se aproximadamente como o RTS1, já que os *labels* com menores valores são criados primeiro e geralmente possuem maior comprimento, mas isso não é garantido em todas as situações [5].

4.5 Hoshen-Kopelman

O principal objetivo desse algoritmo é determinar o tamanho dos componentes conexos em apenas um passo. Para a rotulação de cada objeto com um único *label*, é necessário um segundo passo [8].

Esse algoritmo não usa árvore de decisão. Assim como o SCT4, no *scanning*, ele examina todos os seus quatro vizinhos já rotulados (*labels a*, *b*, *c* e *d*). Abaixo, encontra-se o algoritmo [9].

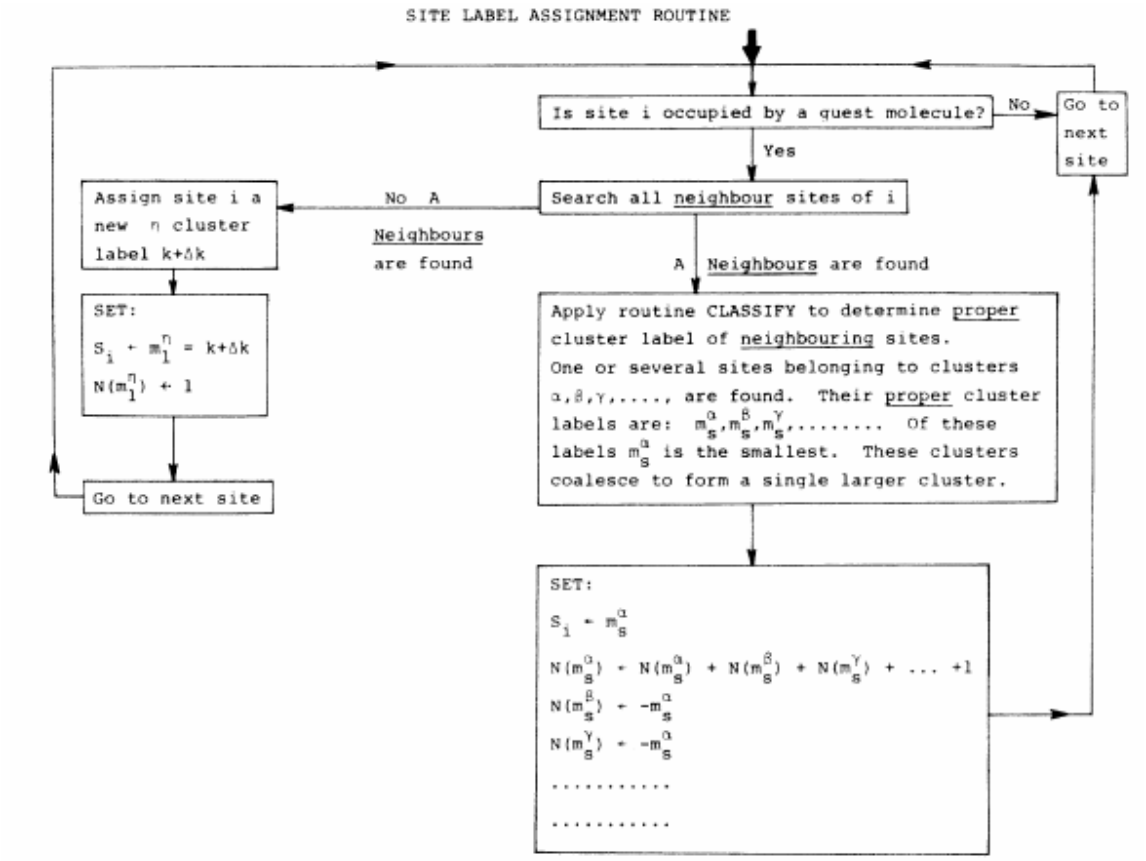

Figura 12. Algoritmo Hoshen-Kopelman

Hoshen-Kopelman utiliza florestas para a resolução da equivalência dos *labels* [7][8], porém a representação da floresta não é a mesma que foi descrita na primeira secção desse capítulo. A floresta é representada através de um *array* unidimensional N. Os representantes, as raízes das árvores, são os elementos *k* tal que N(*k*) é positivo. Esses *labels k* são chamados de *propers labels* ou *labels* corretos. Esse número N(*k*) representa o tamanho, o número de *pixels*, do componente conexo. Os índices *k* em que N(*k*) são negativos representam um índice indireto para um *label* válido. O simétrico desse valor, -N(*k*), é o pai do nó na árvore [7][8][9].

O *make-set* dessa representação de florestas de Hoshen-Kopelman cria um novo label *k* com N(*k*) igual a 1, 1 *pixel* no componente. O *find-set* é chamado por Hoshen-Kopelman de *classify*. Ele examina o *array* na posição passada. Se o valor de N(*k*) é um número positivo, ele encontrou o *proper label*, representante do conjunto. Senão ele continua procurando o *proper label* examinando o pai do *label k*, que é - N(*k*). A rotina *classify* encontra-se na figura abaixo [9].

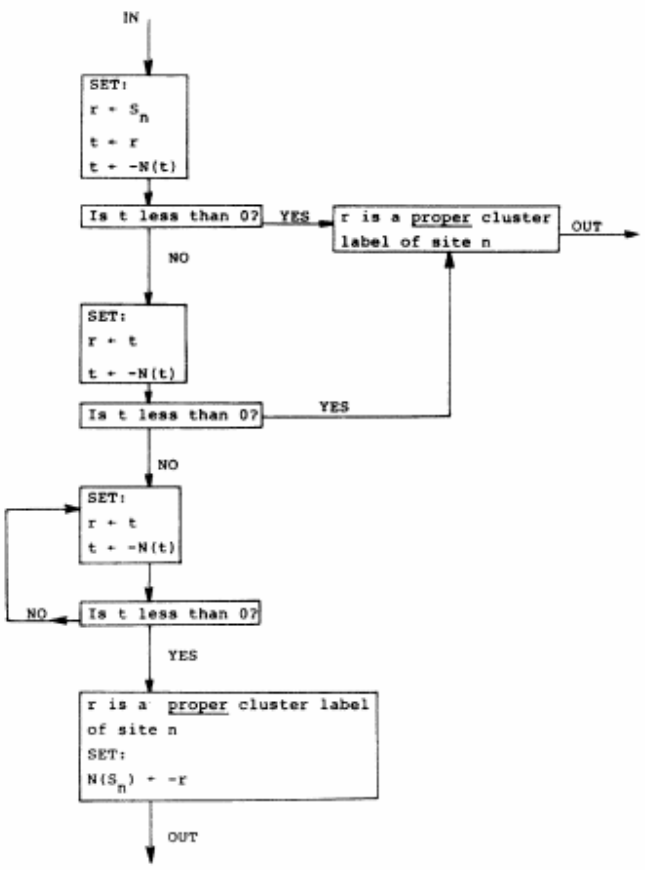

Figura 13. Rotina *classify* do Hoshen-Kopelman

Para realizar o *union* de dois rótulos, assim como no *union* da representação convencional devem-se conhecer os representantes dos dois conjuntos. Considerando que os representantes desses dois conjuntos são r e p, assim como acontece na união convencional, a raiz de um conjunto aponta para a raiz do outro conjunto, $N(p) = -r$. Mas antes, o número de elementos no conjunto deve ser atualizado, $N(r) = N(r) + N(p)$ [9].

Nos mesmos pontos em que o número de elementos, mais conhecido como tamanho ou área do objeto (componente conexo), é calculado, podem-se calcular outras características do objeto, como momentos, centro geométrico, largura e comprimento máximos, entre outros. Hoshen em [7] e [8] e Martín-Herrero et al. em [11] mostram como extrair essas características do objeto.

5. Algoritmos One-pass

Apresenta-se neste capítulo os algoritmos de rotulação *one-pass*. Os algoritmos *one-pass* examinam a imagem apenas uma vez. Porém isso não significa que cada *pixel* é examinado uma vez, mas sim que atravessa a imagem apenas uma vez. Tipicamente acontecem acessos irregulares aos *pixels* da imagem [13]. Esses algoritmos não requisitam uma fase de rotulação (*labeling phase*), uma vez que um *label* é designado a um pixel, ele não é mudado [1].

5.1 Countor Tracing

O mais eficiente algoritmo de *one-pass* é o *Contour Tracing* (CT). Esse algoritmo é baseado no princípio que um componente é determinado pelo seu contorno, assim como um polígono é determinado pelos seus vértices. Obtém como produto os *labels* dos componentes conexos em ordem seqüencial [1].

Conceitualmente, o CT pode ser divido em quatro etapas ilustradas na figura 14. Quando um contorno externo é encontrado (*pixel A* na figura 14a) pela primeira vez, ou seja, um *object pixel* sem *label* cujo vizinho acima é um *background pixel*, o contorno é traçado por completo e o mesmo *label* designado para todos os pontos do contorno. Quando um *pixel* do contorno externo já rotulado é encontrado (*pixel A'* na figura 14b), segue-se a linha até um *pixel* de contorno interno, marcando os *pixels* com o mesmo *label* do pixel de contorno encontrado. Quando um contorno interno é encontrado (*pixel B* na figura 14c) pela primeira vez, ou seja, um *object pixel* sem *label* cujo vizinho abaixo é um *background pixel*, o contorno é traçado por completo e o mesmo *label* designado para todos os pontos do contorno. Quando um *pixel* do contorno interno já rotulado é encontrado (*pixel B'* na figura 14d), segue-se a linha até um *pixel* de contorno externo, marcando os *pixels* com o mesmo *label* do pixel de contorno encontrado [1].

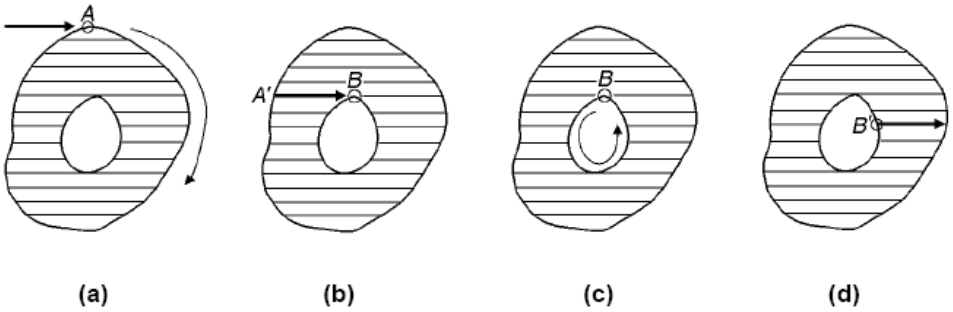

Figura 14. As quatro etapas conceituais do algoritmo de *Contour Tracing*.

5.2 Algoritmos Recursivos

Uma das técnicas para rotular objetos em imagens binárias é o algoritmo de rotulação de componentes conexos recursivo clássico. A função recursiva marca o corrente *pixel* com o corrente *label* e examina toda a sua vizinhança a procura de *object pixels* conectados. Se qualquer *object pixel* é encontrado na vizinhança, a função chama a si mesma para repetir o procedimento nesse *object pixel*. Um simples passo através da imagem, no qual a função é invocada para cada *object pixel* da imagem, rotula cada objeto da imagem com consecutivos *labels* [10][11]. O algoritmo recursivo clássico pode ser visto na figura abaixo.

```
MAIN
    set Label = 0
    for all i, j
        if site_{i,j} = −1
            add 1 to Label
            Label_Site(i, j)
        end of if
    end of for
end of MAIN

LABEL_SITE(i, j)
    set site_{i,j} = Label
    for each neighbour site_{m,n}
        if site_{m,n} = −1 do Label_Site(m,n)
    end of for
end of LABEL_SITE
```

Figura 15. Algoritmo Recursivo Clássico

O algoritmo atravessa seqüencialmente a imagem até que encontra um *object pixel*. Então, ele incrementa o corrente *label* e chama a rotina auxiliar recursiva LABEL_SITE que marca o corrente *pixel* com o corrente *label* e procura por *objects pixels* ainda não rotulados em sua vizinhança. Se encontrar, a rotina LABEL_SITE chama a si própria, rotula o *object pixel* com o mesmo *label*, o corrente *label*, e continua a procurar *objects pixels* ainda não rotulados em sua vizinhança. Se não encontra, a rotina volta para onde ela foi chamada. A função recursiva é chamada apenas uma vez para cada *object* pixel na imagem. Cada vez que a rotina LABEL_SITE é chamada no MAIN, um objeto é rotulado por inteiro [10][11].

Os algoritmos recursivos permitem extrair características dos objetos tais como área, centro geométrico, momentos, em paralelo com a rotulação dos objetos em apenas um passo através da imagem. Já nos algoritmos de dois passos, segundo Martín-Herrero, essa caracterização é um pouco mais complicada [10].

O que limita a eficiência e até a execução dos algoritmos recursivos é a *stack system*. Os algoritmos recursivos requerem uma pilha muito grande para armazenar as variáveis locais e os estados das variáveis para um grande número de chamadas recursivas consecutivas quando executado para grandes objetos. Uma solução é converter o algoritmo recursivo em iterativo. Martín-Herrero propõe um híbrido entre essas duas abordagens [10]. No seu algoritmo (MH), a iteração é usada para rotular *pixels* na mesma linha e a recursão é usada para mudar de linha. Seu algoritmo pode ser visto na figura abaixo [10].

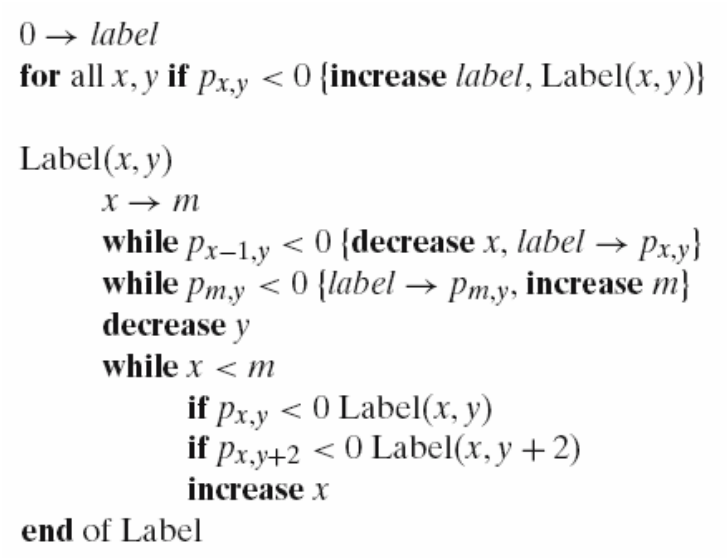

Figura 16. Algoritmo Híbrido de Martín-Herrero

Assim como o algoritmo recursivo clássico, Martín-Herrero rotula um objeto por inteiro com o corrente *label*, quando a função auxiliar recursiva é chamada. *Pixels* são rotulados em *bursts*, grupos de adjacentes *objects pixels* na mesma linha. O algoritmo inicialmente examina a linha em *backward scan*, decrementando x até que encontra o primeiro *pixel* do *burst*. Ele armazena essa posição em x. Em seguida, examina a linha em *forward scan*, incrementando m até encontrar o último *pixel* do *burst*. Ele armazena essa posição em m. Com o início e o fim do *burst* armazenado em x e m respectivamente, ele chama recursivamente o algoritmo para a linha de cima e para a linha de baixo.

O algoritmo da figura acima rotula a imagem usando conectividade-de-4. Para usarmos conectividade-de-8, o tipo que usamos nesse trabalho, a variável x deve ser decrementada e a variável m incrementada. Porém, para reduzir o número de operações no algoritmo isso pode ser realizado substituindo

while $p_{x-1,y} < 0$ *{decrease x, label* $\rightarrow p_{x,y}$ *}* por
while $p_{x,y} < 0$ *{ label* $\rightarrow p_{x,y}$*, decrease x }* e
$x \rightarrow m$ por $x+1 \rightarrow m$ e
while $x < m$ por *while* $x \leq m$.

5.3 Teoria dos Grafos

Grafo é uma estrutura de dados representada por um conjunto de pontos, chamados de vértices, ligados por conjunto de retas chamadas de arestas. Uma imagem pode ser representada por um grafo, considerando que os *pixels* são os vértices e as relações de adjacência entre os *pixels* são as arestas.

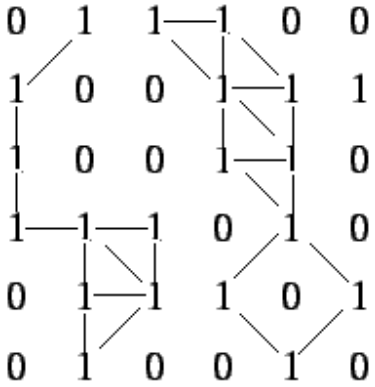

Figura 17. Imagem representada por grafo, onde as linhas são as arestas e os *pixels* são os vértices.

A representação de uma imagem como um grafo permite explorar algoritmos eficientes em grafos. Em particular, estamos interessados em conectividade para rotulação de componentes conexos. Cada grafo numa imagem representa um objeto ou componente conexo. Para rotulá-lo, é necessário fazer uma pesquisa no grafo. Pesquisar um grafo consiste em acompanhar sistematicamente as arestas do grafo de modo a alcançar todos vértices do grafo [2]. As formas mais comuns de pesquisar em grafos são a busca em largura e a busca em profundidade.

A busca em largura é um dos algoritmos mais simples de se pesquisar em grafos. Ela explora sistematicamente a partir de um vértice v as arestas até descobrir cada vértice acessível a partir de v. Segundo Cormen et al., "essa busca recebe esse nome porque expande a fronteira entre vértices descobertos e não descobertos uniformemente ao longo da extensão da fronteira, isto é, o algoritmo descobre todos os vértices a distância k a parti de v , antes de descobrir quaisquer vértice a distância $k + 1$" [2].

Para implementar a busca em largura, é necessária uma estrutura de dados FIFO (*First In First Out*). Filas seguem o padrão FIFO, usando apenas as funções *queue* (inserir no fim) e *dequeue* (remover do começo). Para isso, precisa armazenar o *head* (início da fila) e o *tail* (fim da fila). A fila foi implementada usando *array* para melhor eficiência com acessos seqüenciais. O *head* é o índice do *array* que está o primeiro elemento e o *tail* o índice do último. Inicialmente, o *head* e o *tail* são 0 e a lista encontra-se vazia se o *head* for igual ao *tail*. O *queue* é implementado

incrementando o *tail* (*tail = (tail +1) mod tamanho_do_array*) e o *dequeue* é implementado incrementando o *head* (*head = (head +1) mod tamanho_do_array*).

A busca em profundidade explora o grafo o "mais fundo" sempre que possível. Na busca em profundidade, as arestas são exploradas a partir do vértice *v* mais recentemente descoberto. Quando todas as arestas do vértice são exploradas, a busca retrocede para explorar as arestas que deixam o vértice a partir do qual *v* foi descoberto. Esse processo continua até descobrirem-se todos os vértices acessíveis a partir do vértice inicial.

Para implementar a busca em profundidade, é necessária uma estrutura de dados LIFO (*Last In First Out*). Pilhas seguem o padrão LIFO, usando apenas as funções *push* (inserir no começo) e *pop* (remover do começo). Para isso precisa armazenar o *top* (topo da pilha). A pilha também foi implementada usando *array*. O *top* é o índice do último elemento inserido no array. Inicialmente, esse valor é 0 e a pilha estará vazia quando esse valor for 0. O *push* é implementado incrementando o *top* e o *pop* decrementando o *top*.

Para fim de comparações, são apenas apresentados os resultados da busca em largura (BL). Abaixo, encontra-se a descrição do algoritmo.

```
Para todo pixel p de um object pixel, tal que label(p) = 0
    label(p) ← novoLabel e insira p na FILA F
    Enquanto a fila F não estiver vazia
          Remova p da fila F
          Para todo q vizinho de p, tal que label(q) = 0
            label(q) ← novoLabel e insira na FILA F
    incremente novoLabel
```

Figura 18. Algoritmo de Busca em Largura para Rotulação de Componentes Conexos.

6. Experimentos

Apresenta-se neste capítulo a metodologia utilizada para avaliação da implementação. Foi enfatizada a relação de tempo de execução para a verificação de qual algoritmo obtém melhor desempenho.

6.1 Metodologia

Todos os algoritmos usados na nossa comparação foram implementados em C, compilados e executados nas mesmas condições. Todos os resultados apresentados nessa seção foram obtidos pela média de 100 execuções em um Intel Celeron CPU 2.13GHz, 896 MB de RAM, *Windows XP Black Edition*. A média dos desvios padrões dessas execuções também são apresentados.

Para avaliar o desempenho dos algoritmos de rotulação foram utilizados um conjunto de imagens artificiais randômicas e oito conjuntos de imagens reais: satélite, digitais, textos, faces, médica, microscópica, textura e natural. Os conjuntos de teste de imagens reais estão descritos na próxima secção. Todas essas imagens coloridas foram convertidas para tons de cinza, tiveram seu tamanho padronizado para 512x512 *pixels* e foram transformadas em imagens binárias, aplicando o algoritmo de limiarização de Otsu [14].

(a) (b) (c)

Figura 19. Etapa de pré-processamento. A imagem (a) equivale à imagem colorida, (b) a imagem em tons de cinza e (c) a imagem binarizada.

Para avaliar o desempenho dos algoritmos com a variação do tamanho e da densidade da imagem foram gerados 10 conjuntos imagens diferentes com distribuição uniforme, cada conjunto com 6 diferentes tamanhos (32x32, 64x64, 128x128, 256x256, 512x512 e 1024x1024). Para cada tamanho, foram geradas 41 imagens binárias, variando o *threshold* de 0 a 1, com intervalos de 0.025), totalizando 2460 imagens.

6.2 Conjuntos de Teste

Satélite: Esse conjunto é composto por 38 imagens de satélite e outras imagens aéreas que podem ser encontradas no Signal & Image Processing Institute of University of Southern California[1].

Figura 20. Exemplo de Imagem do Conjunto de Teste **Satélite.**

Digitais: Esse conjunto é composto por 80 imagens de impressões digitais que podem ser encontradas no Fingerprint Verification Competition[2].

Figura 21. Exemplo de Imagem do Conjunto de Teste **Impressões Digitais.**

Textos: Esse conjunto é composto por 8 imagens de textos escritos e com imagens que podem ser encontrados no Signal Analysis and Machine Perception Laboratory of The Ohio State University[3].

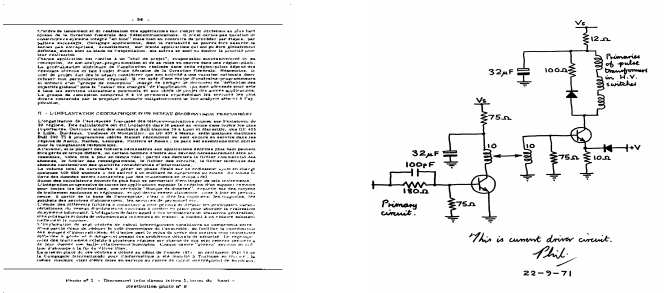

Figura 22. Dois Exemplos de Imagens do Conjunto de Teste **Textos.**

[1] http://sipi.usc.edu/database/database.cgi?volume=aerials
[2] http://bias.csr.unibo.it/fvc2000/db2.asp
[3] http://sampl.ece.ohio-state.edu/data/stills/ccitt/index.htm

Faces: Esse conjunto é formado pelas 30 primeiras faces com diferentes expressões e posicionamentos das 240 presentes no Active Appearance Models of Technical University of Denmark[4].

Figura 23. Exemplo de Imagem do Conjunto de Teste **Faces.**

Médica: Esse conjunto é formado por 100 imagens médicas de radiografias de diferentes órgãos do corpo humano e pode ser obtida no Medical Image Database of Uniformed Services University[5].

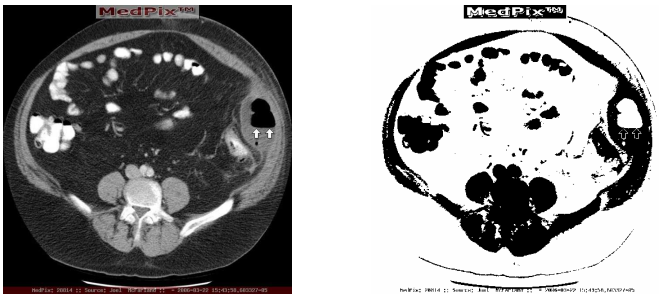

Figura 24. Exemplo de Imagem do Conjunto de Teste **Médica.**

Microscópica: Esse conjunto é formado por 22 imagens de algas microscópicas que podem ser obtidas no Image Processing and Vision Modeling Group do Instituto de Optica (CSIC)[6].

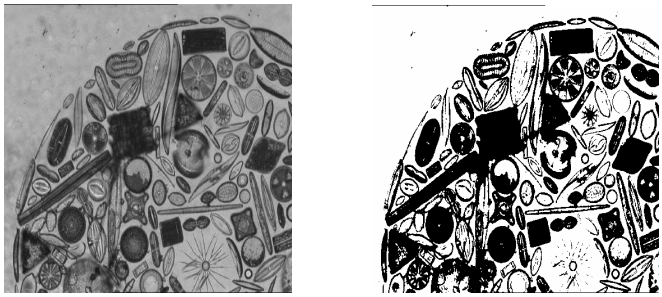

Figura 25. Exemplo de Imagem do Conjunto de Teste **Microscópica.**

[4] http://www2.imm.dtu.dk/~aam/datasets/imm_face_db.zip
[5] http://rad.usuhs.mil/medpix/
[6] http://www.iv.optica.csic.es/page44/page48/files/images/

Textura: Esse conjunto é formado por 64 imagens e podem ser encontradas no Signal & Image Processing Institute of University of Southern California[7].

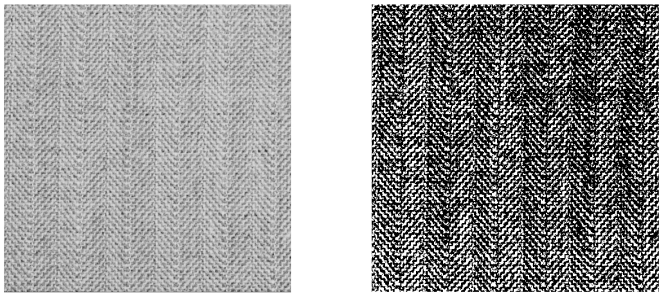

Figura 26. Exemplo de Imagem do Conjunto de Teste **Textura.**

Natural: Esse conjunto é formado por 85 imagens usadas como imagens padrões da maioria dos algoritmos de Processamento de Imagens, sendo que 41 delas podem ser encontradas no Image Processing and Vision Modeling Group do Instituto de Optica (CSIC)[8] e 44 no Signal & Image Processing Institute of University of Southern California [9].

Figura 27. Exemplo de Imagem do Conjunto de Teste **Natural.**

Figura 28. Exemplo de Imagem do Conjunto de Teste **Natural.**

[7] http://sipi.usc.edu/database/database.cgi?volume=textures
[8] http://www.iv.optica.csic.es/page44/files/images
[9] http://sipi.usc.edu/database/database.cgi?volume=misc

6.3 Resultados

Os algoritmos implementados foram aplicados a cada imagem dos conjuntos de testes por 100 vezes consecutivas para a obtenção do tempo médio de execução dos mesmos com o seu desvio padrão. Esses tempos médios são apresentados para cada conjunto de teste nas tabelas dessa secção, seguido do desvio padrão médio entre parênteses. O conjunto de teste *Reais* representa a junção de todas as dos oito conjuntos de imagens reais (satélite, texto, faces, digital, médica, natural, microscópica e textura). Os algoritmos implementados foram descritos nas secções anteriores.

Tabela 1. Lista de Algoritmos

	Sigla	Secção
Scan Plus Connection Table 4	SCT4	3.2
Scan Plus Connection Table 1	SCT1	3.3
Scan Plus Union-Find 0	SUF0	4.2
Scan Plus Union-Find 1	SUF1	4.2
Scan Plus Array-Based Union-Find	SAUF	4.3
Run-Based Two-Scan 0	RTS0	4.4
Run-Based Two-Scan 1	RTS1	4.4
Hoshen-Kopelman	HK	4.5
Contour Tracing	CT	5.1
Híbrido de Martín-Herrero	MH	5.2
Busca em Largura	BL	5.3

A Tabela 2 apresenta os resultados obtidos pelos algoritmos de múltiplas passagens. Verificou se que, em média, o desempenho do SCT1 é melhor que o SCT4 em média 2,30 vezes para imagens reais e 2,32 vezes para imagens randômicas. Isso ocorre porque o SCT1 não precisa varrer todos os *pixels* para definir o rótulo do objeto, utilizando árvores de decisão.

Tabela 2. Eficiência da Árvore de Decisão

	SCT4	SCT1	**SCT4/SCT1**
Satélite	89,18(2,77)	37,14(1,64)	**2,40**
Texto	25,81(0,73)	16,10(0,70)	**1,60**
Faces	41,26(1,00)	19,71(0,85)	**2,09**

Digital	69,39(1,93)	32,74(1,66)	**2,12**
Médica	60,09(1,76)	25,92(1,19)	**2,32**
Natural	80,19(2,02)	32,85(1,48)	**2,44**
Microscópica	50,71(1,69)	24,29(1,13)	**2,09**
Textura	98,93(2,19)	41,39(1,90)	**2,39**
Reais	71,72(1,92)	31,17(1,45)	**2,30**
Randômica	92,87(2,18)	40,08(1,53)	**2,32**

A Tabela 3 apresenta os resultados obtidos pelos algoritmos de duas passagens que utilizam florestas para o *union-find*. Verificou-se que o SAUF possui o melhor desempenho dos três. Esse ganho se deve a utilização de *array* (levando em consideração a proximidade da informação), em contrapartida os demais utilizaram ponteiro. O SUF0 é mais eficiente que o SUF1 porque o segundo executa uma operação a mais para cada *pixel* (*flatten*).

Tabela 3. Eficiência do Uso de Arrays

	SUF0	SUF1	**SUF1/SUF0**	SAUF	**SUF0/SAUF**
Satélite	18,40(1,14)	24,07(1,18)	**1,31**	9,45(0,75)	**1,95**
Texto	10,15(0,44)	12,53(1,02)	**1,23**	6,36(0,51)	**1,60**
Faces	11,59(0,48)	14,91(0,71)	**1,29**	6,89(0,64)	**1,68**
Digital	16,46(0,87)	21,91(1,23)	**1,33**	8,69(0,81)	**1,89**
Médica	14,99(0,81)	20,22(1,15)	**1,35**	7,93(0,72)	**1,89**
Natural	18,19(1,07)	24,71(1,25)	**1,36**	9,07(0,68)	**2,01**
Microscópica	13,58(0,67)	17,38(0,86)	**1,28**	7,62(0,65)	**1,78**
Textura	20,91(1,16)	28,24(1,52)	**1,35**	10,49(0,73)	**1,99**
Reais	16,68(0,91)	22,29(1,19)	**1,34**	8,70(0,72)	**1,92**
Randômica	22,24(1,20)	29,02(1,27)	**1,31**	11,41(0,61)	**1,95**

A Tabela 4 apresenta os resultados obtidos pelos algoritmos de duas passagens que utilizam árvores de decisão. Verificou-se que os algoritmos SAUF, RTS0 e RTS1 possuem desempenho semelhante. Isso faz refletir que as condições em que os *labels* com valores maiores e que possuem conjuntos de maior tamanho são raras, fazendo com que RTS0 e RTS1 tenham eficiência semelhante. Também verificou-se que a eficiência da utilização de listas ligadas é semelhante a utilização de florestas para o *union-find*, já possuem custos semelhantes quando são aplicadas as heurísticas de otimização [2]. Isso faz com que o RTS e o SAUF tenham

eficiências semelhantes. Porém era esperado que o RTS tivesse uma eficiência um pouco melhor do que a do SAUF pois o algoritmo de rotulação geralmente realiza mais operações de *find-set* do que de *union*. Lembrando que o *find-set* tem custo constante para listas ligadas.

Tabela 4. Eficiência da Utilização de Listas Encadeadas

	RTS0	RTS1	**RTS0/RTS1**	SAUF	**SAUF/RTS**
Satélite	9,37(0,80)	9,39(0,72)	**1,00**	9,45(0,75)	**1,01**
Texto	6,33(0,64)	6,35(0,62)	**1,00**	6,36(0,51)	**1,00**
Faces	6,83(0,53)	6,83(0,62)	**1,00**	6,89(0,64)	**1,01**
Digital	8,54(0,80)	8,52(0,68)	**1,00**	8,69(0,81)	**1,02**
Médica	7,87(0,78)	7,84(0,72)	**1,00**	7,93(0,72)	**1,01**
Natural	9,01(0,77)	9,00(0,73)	**1,00**	9,07(0,68)	**1,01**
Microscópica	7,52(0,55)	7,52(0,66)	**1,00**	7,62(0,65)	**1,01**
Textura	10,37(0,69)	10,41(0,73)	**1,00**	10,49(0,73)	**1,01**
Reais	8,61(0,74)	8,60(0,70)	**1,00**	8,70(0,72)	**1,01**
Randômica	11,54(0,85)	11,79(0,59)	**0,98**	11,41(0,61)	**0,99**

A Tabela 5 apresenta uma síntese dos resultados dos algoritmos de duas passagens, comparando o HK com o RTS, algoritmo que utiliza árvore de decisão onde se verificou melhor desempenho. Verificou se que, em média, o desempenho do RTS é melhor que o HK em média 2,07 vezes para imagens reais e 1,88 vezes para imagens randômicas. Isso ocorre porque os algoritmos de árvores de decisão não precisam varrer todos os *pixels* para definir o rótulo do objeto.

Tabela 5. Eficiência do Hoshen-Kopelman

	HK	RTS	**HK/RTS**
Satélite	20,01(0,94)	9,39(0,72)	**2,13**
Texto	7,44(0,66)	6,35(0,62)	**1,17**
Faces	10,94(0,55)	6,83(0,62)	**1,60**
Digital	17,33(0,84)	8,52(0,68)	**2,04**
Médica	15,60(0,84)	7,84(0,72)	**1,99**
Natural	20,04(1,01)	9,00(0,73)	**2,23**
Microscópica	13,01(0,66)	7,52(0,66)	**1,73**
Textura	23,83(1,23)	10,41(0,73)	**2,29**

Reais	17,80(0,91)	8,60(0,70)	**2,07**
Randômica	21,75(1,31)	11,54(0,85)	**1,88**

A Tabela 6 apresenta os resultados para o CT, comparando com o RTS, que apresentou o melhor desempenho entre os algoritmos de duas passagens. Foi identificado que o CT possui melhor desempenho em imagens cujos objetos são bem definidos como nas imagens de faces, enquanto que o RTS é mais eficiente em imagens randômicas ou que possuem ruídos.

Tabela 6. Eficiência do Contour Tracing

	CT	RTS	**RTS/CT**
Satélite	7,51(0,83)	9,39(0,72)	**1,25**
Texto	4,77(0,34)	6,35(0,62)	**1,33**
Faces	2,92(0,20)	6,83(0,62)	**2,34**
Digital	6,19(0,48)	8,52(0,68)	**1,38**
Médica	3,37(0,25)	7,84(0,72)	**2,33**
Natural	5,09(0,40)	9,00(0,73)	**1,77**
Microscópica	5,42(0,49)	7,52(0,66)	**1,39**
Textura	9,35(0,70)	10,41(0,73)	**1,11**
Reais	5,61(0,45)	8,60(0,70)	**1,53**
Randômica	14,60(0,67)	11,54(0,85)	**0,79**

A Tabela 7 apresenta os resultados para o MH, comparando com CT e com o RTS. Foi identificado que o MH possui desempenho semelhante ao RTS, sendo o RTS um pouco melhor para imagens reais e o MH um pouco melhor para imagens randômicas. Então, assim como o RTS, o CT possui melhor desempenho para imagens de objetos bem definidos , enquanto o MH para imagens randômicas.

Tabela 7. Eficiência do Martín-Herrero

	MH	CT	**MH/CT**	RTS	**RTS/MH**
Satélite	8,58(0,68)	7,51(0,83)	**1,14**	9,39(0,72)	**1,10**
Texto	4,34(0,54)	4,77(0,34)	**0,91**	6,35(0,62)	**1,46**
Faces	4,46(0,30)	2,92(0,20)	**1,53**	6,83(0,62)	**1,53**
Digital	7,31(0,54)	6,19(0,48)	**1,18**	8,52(0,68)	**1,16**
Médica	5,81(0,49)	3,37(0,25)	**1,72**	7,84(0,72)	**1,35**

Natural	7,56(0,81)	5,09(0,40)	**1,49**	9,00(0,73)	**1,19**
Microscópica	6,21(0,73)	5,42(0,49)	**1,14**	7,52(0,66)	**1,21**
Textura	10,40(0,87)	9,35(0,70)	**1,11**	10,41(0,73)	**1,00**
Reais	7,27(0,63)	5,61(0,45)	**1,30**	8,60(0,70)	**1,18**
Randômica	11,99(0,19)	14,60(0,67)	**0,82**	11,54(0,85)	**0,96**

A Tabela 8 apresenta os resultados para os algoritmos de Teoria dos Grafos, particularmente o de Busca em Largura. Verificou-se que o seu desempenho foi pior do que os do CT e do MH. Os algoritmos de Teoria dos Grafos não obtiveram resultados esperados. Esperava-se que eles tivessem desempenho melhor ou semelhante ao do MH, pois transforma um problema recursivo em iterativo, enquanto o MH é um híbrido recursivo-iterativo. O desempenho ruim se deve ao fato de ter que criar e gerenciar estruturas de dados armazenadas na memória virtual, *heap*.

Tabela 8. Eficiência dos Algoritmos de Teoria dos Grafos.

	BL	MH	**MH/BL**	CT	**CT/BL**
Satélite	21,65(1,31)	8,58(0,68)	**0,40**	7,51(0,83)	**0,35**
Texto	7,31(0,62)	4,34(0,54)	**0,59**	4,77(0,34)	**0,65**
Faces	10,33(0,62)	4,46(0,30)	**0,43**	2,92(0,20)	**0,28**
Digital	18,40(0,87)	7,31(0,54)	**0,40**	6,19(0,48)	**0,34**
Médica	15,43(0,83)	5,81(0,49)	**0,38**	3,37(0,25)	**0,22**
Natural	20,90(1,14)	7,56(0,81)	**0,36**	5,09(0,40)	**0,24**
Microscópica	13,34(0,96)	6,21(0,73)	**0,47**	5,42(0,49)	**0,41**
Textura	26,01(1,32)	10,40(0,87)	**0,40**	9,35(0,70)	**0,36**
Reais	18,58(1,00)	7,27(0,63)	**0,39**	5,61(0,45)	**0,30**
Randômica	25,16(1,20)	11,99(0,19)	**0,48**	14,60(0,67)	**0,58**

As figuras 29 e 30 apresentam gráficos com a relação entre o tempo de execução dos algoritmos e o número de *pixels* na imagem. Para os testes, foram utilizadas as imagens randômicas geradas conforme descrito na secção 1 deste capítulo. Esses gráficos confirmam o fato dos algoritmos serem lineares em relação ao número de *pixels* na imagem. Verifica-se ainda que o SCT4 é o que apresenta crescimento assintótico mais rápido, enquanto o SAUF, o RTS0 e o RTS1 possuem crescimento mais lento. A figura 30 apresenta os algoritmos de melhores desempenhos, nela, o RTS0 e o RTS1 foram excluídos por possuírem resultados semelhantes aos do SAUF.

Não foi possível a execução do algoritmo de Martín-Herrero para as imagens com 1024x1024 *pixels* de tamanho. O *stack system* não é suficiente. O próprio Martín-Herrero propõe dividir a imagem em várias sub-imagens, aplicar o algoritmo para essas sub-imagens e resolver as equivalências nas bordas, como pode ser visto em [11].

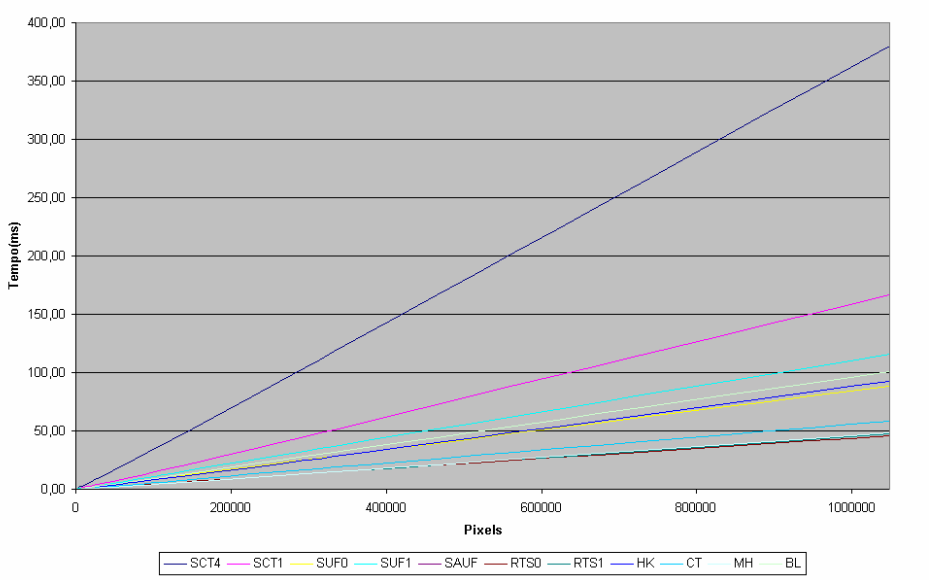

Figura 29. Relação do tempo de execução dos algoritmos e o número de pixels na imagem.

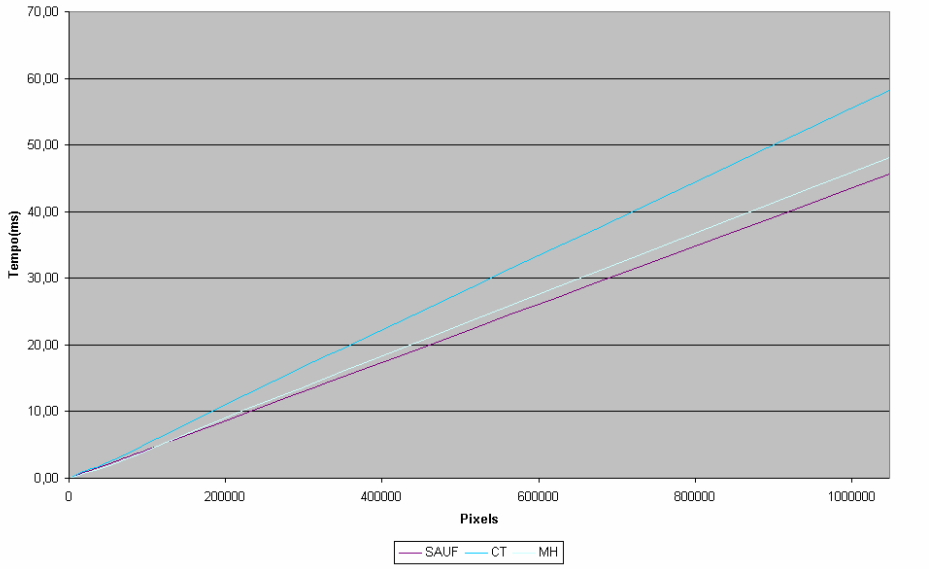

Figura 30. Relação do tempo de execução dos algoritmos e o número de pixels na imagem para os algoritmos de melhor desempenho.

As figuras 31 e 32 apresentam gráficos com a relação entre o tempo de execução dos algoritmos e a densidade da imagem. Uma imagem densa significa

uma imagem com objetos maiores e, portanto, menos objetos. Já uma imagem de baixa densidade apresenta objetos menores e com muitos objetos espalhados. Verificou-se que os algoritmos de rotulação de duas passagens são menos influenciados pela densidade do que o CT, que apresenta tempo médio máximo na densidade de 0.5, quando o tamanho dos objetos e a sua quantidade influenciam da mesma forma o desempenho do algoritmo. Isso acontece porque o CT tem seu desempenho piorado para grandes objetos e para imagens com muitos objetos espalhados, pois aumenta a quantidade de acessos aleatórios.

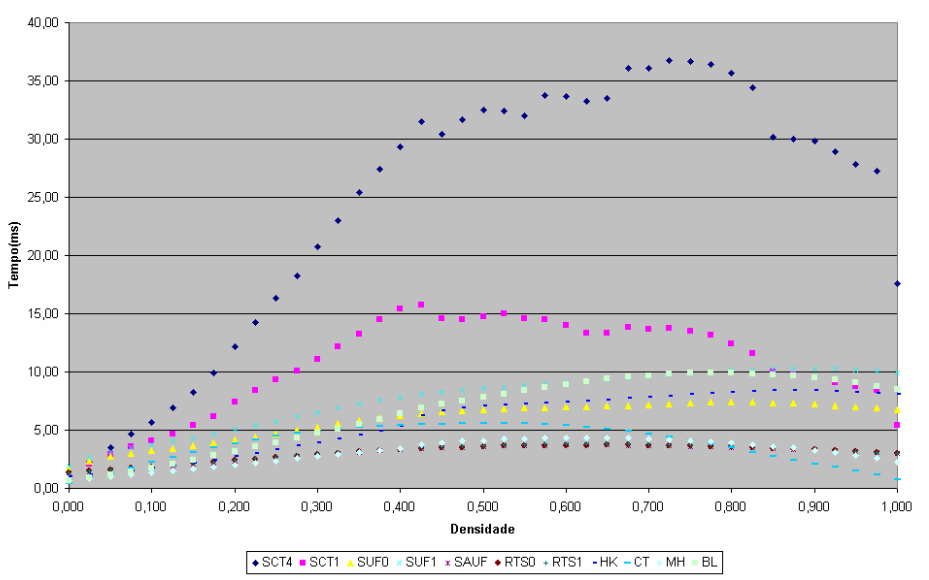

Figura 31. Relação do tempo de execução dos algoritmos e a densidade da imagem.

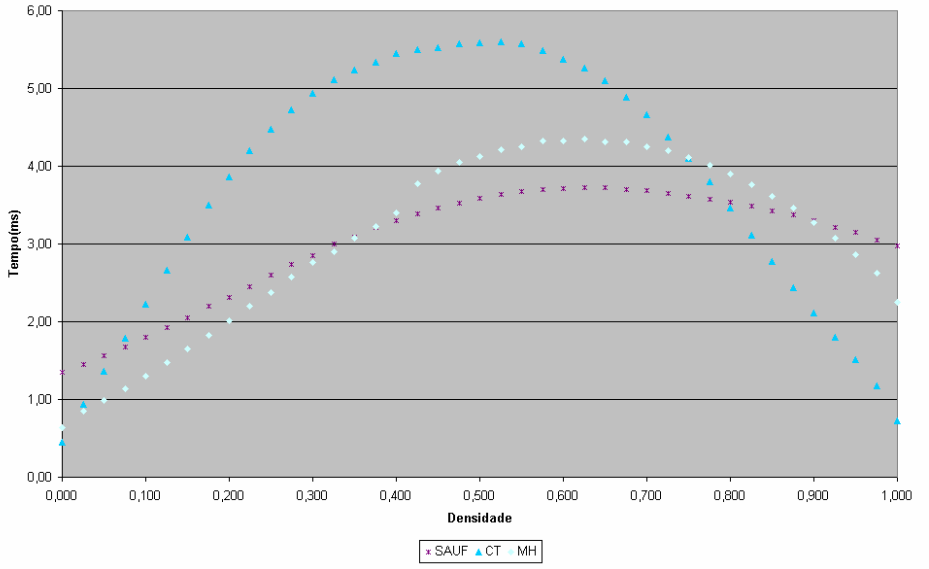

Figura 32. Relação do tempo de execução dos algoritmos e a densidade da imagem para os algoritmos de melhor desempenho.

7. Conclusão

Foram estudadas a eficiência e otimização dos algoritmos de rotulação de componentes conexos. Os algoritmos estudados foram implementados para comparação das suas eficiências. O SAUF, o MH e o CT foram os algoritmos que possuíram obtiveram melhores resultados. Sendo cada um melhor aplicado a diferentes situações. O CT é melhor quando aplicados a imagens que possuem o cenário bem definido pois trabalha com o contorno dos objetos. Enquanto que o SAUF e o MH são melhores quando aplicados a imagens aleatórias ou que possuem ruídos pois trabalham com a informação de vizinhança.

Foram estudadas duas abordagens para otimizar o tempo de execução de algoritmos de rotulação de componentes conexos. A primeira foi reduzir o tempo gasto na fase de varredura dos *pixels* da imagem através de árvores de decisão. Ela aproveita a noção de vizinhança para reduzir a quantidade de *pixels* visitados. Essa abordagem obteve melhorias em algoritmos de múltiplas passagens. A segunda foi a redução do tempo necessário para manipular a informação equivalente utilizando conjuntos disjuntos baseado em *arrays*, aproveitando-se do acesso seqüencial.

Muito trabalho ainda precisa ser feito para melhor entendermos o desempenho dos algoritmos. Por exemplo, os algoritmos de Teoria dos Grafos, que não apresentaram o desempenho esperado, precisam ser mais bem estudados para se descobrir gargalos de implementação. Técnicas alternativas para os algoritmos recursivos ainda precisam ser melhores estudadas.

Para trabalhos futuros, estudaremos a aplicação de árvores de decisão no algoritmo de Hoshen-Kopelman para reduzir o seu tempo de execução. Também será muito útil, incluir nos outros algoritmos o cálculo de extratores de características (área, massa, momentos, centro geométrico, entre outros) dos objetos que já são calculados no Hoshen-Kopelman.

Por fim, poderemos testar os algoritmos em outras configurações. Além disso, implementar o algoritmo em diferentes linguagens de programação (Java, C, C++, C#, Pyhton e MATLAB são as que tenho domínio) e comparar a sua eficiência com algoritmos já implementados nessas linguagens.

Referências

[1] CHANG F, CHEN C-J, LU C-J (2004) A linear-time component-labeling algorithm using contour tracing technique. Comput Vis Image Underst 93(2):206-220

[2] CORMEN T H, LEISERSON C E, RIVEST R L, STEIN C. Introduction to Algorithms - Second Edition. MIT Press, 2001

[3] FIORIO C, GUSTEDT J (1996) Two linear time union-find strategies for image processing. Theor Comput Sci 154(2):165-181

[4] GONZALEZ R C, WOODS R E. Processamento de imagens digitais. São Paulo: Edgard Blucher, 2000. 509 p.

[5] HE L, CHAO Y, SUZUKI K, WU K (2008) Fast connected-component labeling. Pattern Recognition 42:1977-1987

[6] HE L, CHAO Y, SUZUKI K (2008) A run-based two-scan labeling algorithm. IEEE Transactions on Image Processing 17(5):749-756

[7] HOSHEN J (1999) The application of the enhanced Hoshen-Kopelman algorithm for processing unbounded images. IEEE Transactions on Image Processing 8(3):421-425

[8] HOSHEN J (1998) On the application of the enhanced Hoshen-Kopelman algorithm for image analysis. Pattern Recognition Letters 19:575-584

[9] HOSHEN J, KOPELMAN R (1976) Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm. Physical Review B 14:3438-3445

[10] MARTÍN-HERRERO J (2007) Hybrid object labeling in digital images. Machine Vision and Applications 18:1-15

[11] MARTÍN-HERRERO J, PEÓN-FERNÁNDEZ J (2000) Alternative techniques for cluster labeling on percolation theory. J. Phys A: Math. Gen 33: 1827-1840

[12] SUZUKI K, HORIBA I, SUGIE N (2003) Linear-time connected-component labeling based on sequential local operations. Comput Vis Image Underst 89(1):1-23

[13] WU K, OTTO E, SUZIKI K (2008) Optimizing two-pass connected-component labeling algorithms. Pattern Analysis and Applications 12:117-135

[14] OTSU N (1979) A threshold selection method from gray-level histograms. IEEE Transactions Syst. Man Cybernet. 9:62-66